TOWARDS SYSTEMATIC REQUIREMENTS REUSE

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2014

By
James Naish
School of Computer Science

Contents

Word Count: 71547

List of Tables

List of Figures

Abstract

TOWARDS SYSTEMATIC REQUIREMENTS REUSE
James Naish
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2014

Reuse has often been claimed in the software engineering literature to improve the quality and reduce the cost of software. Motivated by the idea that these gains can be multiplied if reuse can be achieved earlier in the software life-cycle, a subset of the requirements engineering literature has focused, since the inception of the field, on investigating approaches to reuse at the requirements level. A wide array of different approaches now exist within this space. However, these approaches offer varying degrees of generality and utility. Generality is important because it enables a requirements engineer to utilise the same reuse library across multiple projects. Utility is important because it is a measure of the extent to which effort is reduced by utilising a reuse approach.

This thesis presents Reuse-Oriented Requirements Engineering (RORE): a systematic framework to support the production of requirements models by reuse. RORE aims to improve on existing requirements-reuse approaches in respect of the generality-utility trade-off. RORE seeks to do this by bringing together the strengths of two existing requirements-level reuse approaches: The Domain Theory and Problem-Oriented Software Engineering (POSE - a refinement of Jackson's Problem Frames Approach). This thesis evaluates RORE with respect to both generality and utility, and compares RORE against both frameworks. The major conclusion of the thesis is that while RORE improves on each framework in respect of some, but not all, evaluation metrics, RORE does succeed in offering a level of generality which compares favourably to existing highly general approaches, and without significantly reducing the utility of the approach.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

This thesis has been made possible by an extensive supporting cast, but a few names in particular are worthy of a credit. First and foremost I am, of course, indebted to Dr. Liping Zhao, my supervisor, without whom this thesis would simply not have been possible. Through this process, her guidance and support have been invaluable, and her patience has been most deeply appreciated. My thanks must also go to Professor Alistair Sutcliffe for setting me back on a rational path on more than one occasion.

Beyond the academic support I have received throughout my PhD, I must also thank my parents. Without their support in all the usual parental ways, and more, this thesis simply could not have been completed, so: thanks Mum and Dad!

And finally (and I can assure the following names that this is on paper only), it would be criminal of me not briefly to acknowledge the moral support and much needed good humour which has been provided at critical moments, and in no particular order, by: Lars, Wing, Cheree, Habz, Skendha and Susie. Love and thanks to each of you.

Chapter 1

Introduction

1.1 Motivation: Towards Systematic Requirements Reuse

In software engineering, reuse is desirable because it provides a more efficient implementation of the engineering process than is possible without reuse. Furthermore, by the nature of reuse, reusable artefacts can be tried and tested over many applications and so the quality of an artefact can be improved if it is constructed from reusable artefacts. Most of the reuse literature produced to date has emphasised code- or design-level reuse [Kru92, FK05, Gam95]. However, Cheng and Atlee have argued that requirements engineering could benefit substantially from reuse [CA07]. They argue that reuse at the requirements level would facilitate the realisation of a more systematic approach to requirements engineering. Several methodologies to support requirements engineering have been proposed (and accepted) in the literature [DFvL91, DVLF93, Yu93, Yu97, Jac01b]. However, these methodologies often lack detailed methodological guidance to support low-level tasks such as requirements elicitation and analysis [NE00, CA07, SPZ06]. Reuse can fill this gap by providing a means for the dissemination of both domain and procedural knowledge.

There is, however, another reason to seek an effective approach to requirements-level reuse: the gains of reuse can likely be increased if reuse is achieved earlier in the software engineering process [Sut02]. Sadraei et al [SABP07] offer a review of reuse studies, and conclude that between 5-25% of the effort of a software engineering project is spent on requirements engineering. However, most software design and coding errors are traceable to errors at the requirements level [BMU75, Gla98], and these errors are the most costly to fix. Improving the quality of requirements artefacts, therefore, could lead to significant reductions in the time and money spent on fixing errors

during the software engineering process. Furthermore, some approaches (such as generative software development [Cza05] and software product lines [CN01]) associate requirements-space abstractions with solution-space abstractions allowing a mapping from the former to the latter. Where such approaches are possible, greater levels of reuse can be achieved across the software development process as a whole, and greater levels of reuse than would be possible at the code-level alone. The productivity gains which can be achieved through reuse are, then, greater when reuse can be achieved at the requirements level.

Reuse has been a background theme throughout the history of requirements engineering. However, in recent years the RE community has started to make a concerted effort to address this question [CA07]. In particular, the community has established a Requirements Patterns workshop (RePa) to address the question of requirements-level reuse, which has been co-located since its inception in 2011 with the official RE conference [PMCS11]. This development, in conjunction with the sheer proportion of requirements reuse literature which advocates patterns-based approaches, indicates that the community has generally accepted patterns-based approaches as the preferred approach to requirements-level reuse.

Patterns-based approaches to reuse do offer important benefits. They provide a common language through which both domain and technical experts can discuss their ideas [AC98]. Patterns also support the reuse of simple but effective ideas which are well known, but often forgotten [Ris10]. Furthermore the sheer volume of patterns which have been proposed, even just in the requirements engineering literature, means that there is likely to exist a pattern to support most tasks and contexts which a requirements engineer comes across.

The extent of the patterns literature, however, is a double-edged sword. Agerbo and Cornils argued in 1998 [AC98] that the volume of patterns which had been proposed in the literature was already making the body of knowledge which is contained within those patterns unmanageable. This prediction appears today to have been prescient. Recent attempts to systematise patterns knowledge (outside the requirements literature, for this is where cutting-edge patterns research occurs) have either reduced the literature to small but coherent pattern languages (e.g. the Pattern-Oriented Software Architecture series[BHS07]), or else have run into significant, particularly organisational, challenges. Booch's "Handbook of Software Architecture" project [Boo05], for instance, has been criticised for organising patterns in a manner that is "ad hoc" and "limited" [HC11], and for equating commercial value with quality in the selection of

patterns for the handbook [HJN11]. Without an effective system to support the retrieval and application of patterns that are relevant to a particular context, reuse remains ad hoc and the full benefits that reuse can offer will not be realised.

Several approaches to requirements-level reuse have been proposed in the literature which address, indirectly, this limitation to the extent that they offer systems for reuse, rather than simply providing a set of reusable artefacts. One widely acknowledged paradigm for reuse is analogy [Fin88, Mai92, MVL97, CJ97]. Analogical reuse is an approach to artefact retrieval based on the structure-mapping theory of analogy proposed by Gentner [Gen83]. Analogical approaches to reuse, in line with Gentner's work, support the retrieval of reusable artefacts by matching the structure of an abstract artefact to that of a concrete scenario. Gentner's theory holds that multiple domains may share a similar structure, although the details of the entities which comprise that structure will vary [Gen83]. Because of this, analogical reuse approaches have been shown to be highly effective approaches to transferring reusable knowledge between application domains [Mai92, Sut02]. The approaches therefore provide a balance between generality across domains and systematicity in those tasks which they are designed to support. However, each of the analogical approaches that have been proposed have been designed to support the construction by reuse of a specific type of requirements model in the context of a specific requirements methodology [Mai92, MVL97, CJ97].

An alternative approach to reuse is proposed by Jackson [Jac01b] who posits a set of "Problem Frames". Like the domain abstractions posited by Sutcliffe and Maiden [SM98], Problem Frames are highly abstract and describe structural relationships between entities. The major difference, however, is that where a domain abstraction within the Domain Theory describes structural relationships between entities within a single domain [Sut02], Jackson's Problem Frames describe relationships between domains but do not describe the content of a domain [Jac01b]. Consequently, Problem Frames are highly abstract and so offer less utility than the Domain Theory. However, the Problem Frames approach has been extended to support the derivation of design and architectural specifications from the problem models which are composed from problem frames. Problem-Oriented Software Engineering (POSE) uses transformation patterns specified as sequents incrementally to transform problem models into solution models [HRJ08]. In common with the Problem Frames approach itself, POSE shares a high degree of domain generality. Furthermore, POSE is committed neither to a

specific set of transformation patterns nor to a specific schema for expressing knowledge. As such, POSE supports a higher degree of task and method generality than the Domain Theory.

1.2 The Research Problem

In order fully to reap the benefits of reuse, it is desirable that requirements engineers have available an approach to reuse which balances the benefits that are provided by existing reuse approaches. A reuse approach to support requirements engineering should offer support for reuse across a range of application domains and throughout the engineering process. Reuse should be a central component of the engineering process, rather than an activity which occurs only when the requirements engineer remembers a relevant component. The approach should be capable of supporting the full gamut of engineering tasks from early stage requirements gathering to late-stage solution specification. The requirements engineer should be free to use the reuse approach with their own choice of requirements engineering methodology while still have access to a sufficient body of reusable knowledge. Accordingly, knowledge which is gained through a project undertaken using one requirements engineering methodology should be transferable to support engineers who are working with alternative methodologies.

However, like any system, reuse approaches are constrained by a range of different, often opposing, forces acting both on individual components of the system and on the system as a whole thereby affecting the properties of the approach. In the case of reuse approaches, such forces include:

- **Generality**: A measure of the range of domains, reuse contexts and requirements engineering methods to which a reuse approach can be applied;
- **Utility**: The effort reduction achieved by reusing a reusable artefact versus achieving the same goal without the aid of reuse;
- **Systematicity**: The extent to which reuse is a driving force, rather than an incidental occurrence, in the software development process, and to which such reuse is supported by a repeatable set of procedures and tools;
- **Practicality**: The ability of a reuse approach to satisfy the organisational, economic, legal and technical constraints imposed of a practical setting while still yielding utility.

It is the need to optimise the interplay of these different forces which significantly complicates the task of design-for-reuse and which has led to the limitations of existing approaches. Depending on the goals which a reuse designer aims to achieve, different forces need to be taken into consideration in the design of a reuse approach. This thesis advocates an approach to reuse which offers on the one hand generality with respect to domains, tasks and methodology, and, on the other hand, a high degree of systematicity and utility. Furthermore, this thesis advocates that approaches to reuse should achieve these goals while satisfying real-world practicality constraints, such as realistic computational memory and performance constraints.

Finding the right balance between these two sets of properties is a challenging design problem because the design tactics by which generality is often realised (such as reducing granularity or increasing abstraction) tend to impact adversely on the utility of an approach. The generality of a reuse approach is related to the likelihood that a reusable artefact can be identified from the approach to support any given context in which a requirements engineer might find themselves. Generality, therefore, is influenced by two important factors. Firstly, the generality of a reuse approach as a whole is influenced by the generality and number of the reusable artefacts within the system. Increasing the number of reusable artefacts within a library will increase the generality of a library as a whole. This is because whole because for any given use-context the chances that the library will contain a reusable artefact which satisfies that use context are increased if the number of artefacts also increases. Secondly, the generality of the system as a whole is also increased by an increase in the generality of individual artefacts within that system. This increase in generality, however, will be in proportion to the number of artefacts contained within the system: the more artefacts a library contains, the less impact a change in the generality of individual components will have. Alternatively, increasing the generality of several individual components can reduce the number of components needed to cover a given range of scenarios.

A number of design tactics exist which allow a system designer to do this but each inhibits, in some way, either utility or systematicity. Increasing the generality of a reuse approach by either increasing the abstraction or reducing the granularity of the artefacts which that approach comprises will reduce the utility of the approach. The abstraction of a component can be understood as the degree to which a reusable artefact relates to a specific set of concepts or processes. The granularity of a component, roughly, can be defined in terms of the number of such concepts or operations that are contained within the artefact [Sut02].

The entire process of software engineering is one of reifying a set of requirements in order to produce code, which is the most concrete expression of the software design. The engineering process thus ultimately increases the information content of the artefacts which it produces. Reuse is desirable because it is an efficient way of achieving this information gain. However, since abstraction and reduced granularity reduce the information value of a reusable artefact, these tactics also reduce the information gained when that artefact is utilised. Thus a greater degree of effort is required to reify the reusable artefacts, and to compose the artefacts into a new whole [Sut02].

This would suggest that it is desirable to reuse more concrete components and to increase generality by increasing the number of artefacts in the library. However, after a certain point a library will become so large that it is simply not feasible, even for modern machines, to execute in reasonable time search algorithms to retrieve reusable artefacts that are relevant to a particular use context. It is desirable, therefore, in designing for reuse to balance the application of abstraction, granularity, and artefact number when designing for generality.

1.3 Research Aims and Objectives

This thesis proposes a framework which is capable of providing systematic support for reuse across a range of application domains and requirements tasks. The framework for reuse which is presented in this thesis - Reuse-Oriented Requirements Engineering (RORE) - is a systematic approach to model-driven requirements engineering by reuse which aims:

> To find a better balance between generality and systematicity than do existing approaches while maintaining a high level of utility and ensuring that the approach remains practically feasible.

This thesis poses and investigates, as major aims of this research, three questions which are integral to the successful satisfaction of this overall goal:

1. What factors affect the generality, systematicity and utility of a reuse approach?
2. What design tactics can be identified to support the realisation of this aim?
3. How can a reuse framework be designed to balance generality, and systematicity and utility?

To address these questions, this thesis will seek to satisfy the following objectives:

1. To identify a set of design tactics which can be employed in the design of frameworks for requirements reuse to support the goal of balancing generality against utility and systematicity, while ensuring the approach remains practical;

2. To develop a theoretical framework to inform and guide the design of frameworks for requirements reuse;

3. To develop a prototype implementation of the proposed reuse framework.

1.4 Research Challenges

In developing the reuse and theoretical frameworks which are presented in this thesis, the following significant challenges were encountered.

Challenge One. Identifying design strategies which would optimise the balance between generality, systematicity and utility was a considerable technical challenge.

This was a challenge because reuse systems are constrained by numerous and competing forces, such as granularity, abstraction and utility. Many of the design tactics through which generality can be achieved directly undermine utility [Sut02]. Adjusting one parameter tends to affect other parameters, and so the research task facing this thesis amounts to a challenging optimisation problem.

Challenge Two. Developing a multi-disciplinary theoretical understanding of the forces underpinning reuse systems.

In constructing the theoretical framework which this thesis presents to support the design-for-reuse framework, a significant challenge was to identify relevant literature on which the framework could be founded. Maiden has advocated a multi-disciplinary approach to theorising about reuse and proposes cognitive science as one discipline which can contribute [Mai92]. Another useful starting point was Sutcliffe's framework [Sut02]. These approaches, however, are limited in terms of the explanations and predictive power which they offer. Fleshing out this understanding by supplementing the work of Maiden and Sutcliffe with additional literature was therefore a major challenge in approach the design problem discussed in this thesis.

Challenge Three. The need to identify a set of basic procedures on which the reuse approach presented herein could be grounded.

While some of the tasks which reuse frameworks support are common and well known (e.g. component retrieval), requirements engineering consists of other kinds of task (e.g. analytical tasks) which a truly systematic requirements-level reuse framework should also support. There was, therefore, a need to determine how best to partition requirements engineering activities into recurring abstract activities and how best to design those activities to provide a high degree of utility.

Challenge Four. To identify knowledge structures through which reusable knowledge can be expressed.

Because this thesis advocates the integration of a mix of reuse approaches, this was a non-trivial task. A range of formalisms for expressing knowledge have been proposed for expressing knowledge in general (e.g. description logics [Baa03] and production rules [And04, ABB$^+$04]). The ideal situation would be to identify a single formalism which could adequately express all of the reusable structures which the framework presented herein supports. To the best of this author's knowledge, no single formalism exists, however, which neatly expresses both declarative and procedural knowledge. In designing the reuse framework, therefore, this thesis carefully considered the right formalism to express each type of structure.

1.5 Research Contributions

This thesis aims to make three significant research contributions as follows.

Contribution One. This thesis presents a suite of design tactics which can be reused as heuristics to support design-for-reuse where balancing generality against utility is an important design goal.

Specifically, this thesis distinguishes for the first time in the requirements reuse literature between procedural and declarative reuse, and provides some analysis of the respective advantages and limitations of each reuse paradigm. This thesis provides evidence and argument which strongly suggest that procedural reuse is inherently more general without a significant detriment to utility than is declarative reuse.

Contribution Two. This thesis presents a theoretical framework to guide and inform the development of systems for requirements-level reuse in order to satisfy the goal of an effective balance between generality and utility.

The framework codifies the design tactics identified in Contribution One into a coherent framework. The framework identifies the procedures, components and knowledge structures which are required to provide a reuse-driven approach to requirements engineering. The framework can be reused by other designers-for-reuse in order to develop alternative instantiations to that presented in this thesis while retaining the basic properties of the approach with respect to generality and utility.

Contribution Three. The framework presented in this thesis effectively unifies and extends two existing approaches to requirements-level reuse (the Domain Theory [Sut02] and POSE [HRJ08]).

This thesis evaluates the strengths and limitations of both frameworks and shows how aspects of each framework can be integrated in order to produce a framework which is stronger with respect to the overall goal of balancing generality and utility. This thesis shows how these modifications can strengthen the analogical reuse of Sutcliffe and Maiden [Sut02, Mai92] with respect to generality, and the procedural reuse of POSE [HRJ08] with respect to utility.

1.6 Research Methodology

This thesis broadly adopts the Design Science Research Methodology (DSRM) described by Peffers et al [PTRC07] (see Figure 1.1):

The DSRM identifies six stages of design science research: Problem Identification; Objective Definition; Design and Development; Demonstration; Evaluation; Communication. Peffers et al note that the "process is structured in a nominally sequential order" but that in reality the process "may actually start at almost any step" [PTRC07]. This thesis begins with problem identification.

Problem Identification. An object-centred approach to design begins with a practical or research need. The first step, therefore, in such a project is to carefully define the problem. The need which this thesis attempts to fulfil is the need to provide requirements engineers with a single approach to reuse which they can use to undertake a wide range of different engineering activities and in a range of different domains.

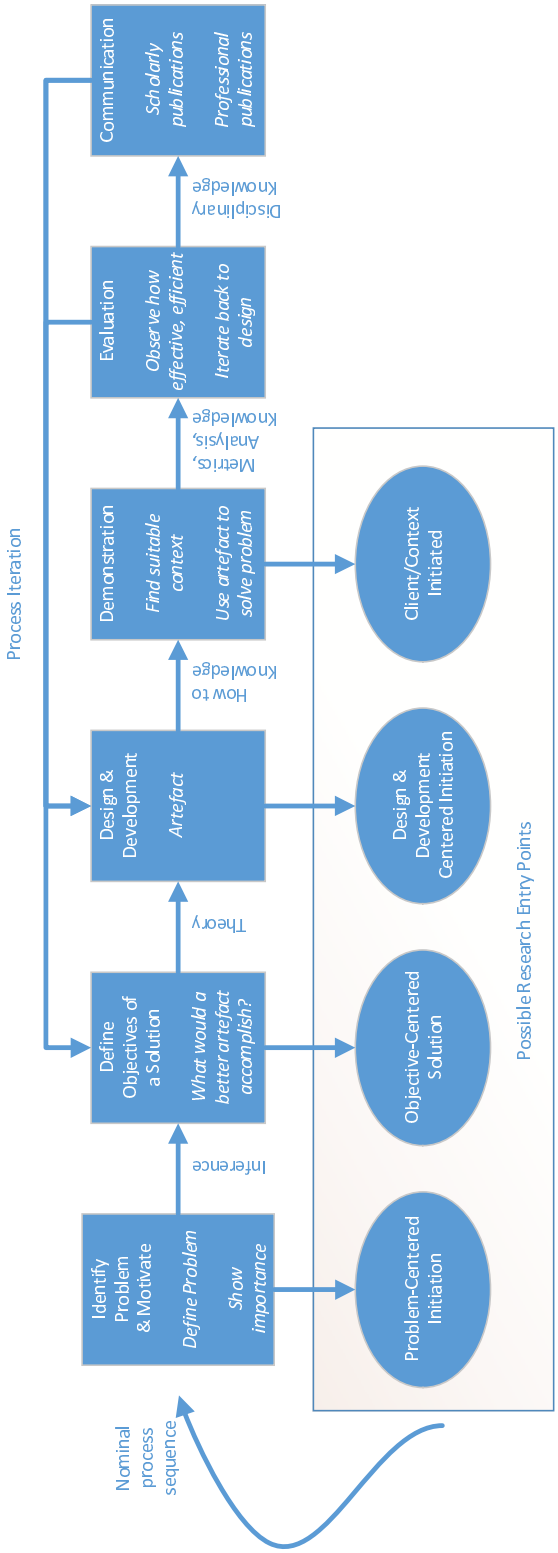

Figure 1.1: The Design Science Research Methodology described by Peffers et al [PTRC07]

Objective Definition. The next step was to formulate the aims and objectives of this research project. To achieve this, this thesis sought better to understand the extent to which existing approaches to requirements-level reuse satisfy this need and, where they do not, to understand why this need has historically been difficult to fulfil. A systematic review of requirements reuse literature was therefore undertaken. An early version of this survey is presented in [NZ11]. An exhaustive search was conducted through every issue of the Requirements Engineering and Transactions on Software Engineering journals, as well as the IEEE International Conferences on Requirements Engineering (REC) and Software Engineering (ICSE) dating back to 1990. Papers were included in the survey if they addressed reuse or artefact retrieval explicitly. Each paper was then analysed to evaluate the extent to which the approach which was discussed within supported the goals of generality, systematicity and utility. This analysis was aggregated in order to draw general conclusions about the generality, systematicity and utility of each approach.

Design and Development. The DSRM stipulates that the design step should be informed by theoretical knowledge about the problem to be solved. This is in line with Wieringa's advocacy of theoretically-informed approaches to design in the requirements engineering literature [Wie05]. To this end, this research sought to identify a set of design heuristics on which the design of the framework which this thesis presents could rationally be predicated. The theoretical framework described by Sutcliffe [Sut02] provided a starting point for this task. However, further reading in the fields of information theory [Sha01] and the physics of information [Lan61, Bek04, Bek03, TSU+10] provided further insights about the forces which underpin and act on systems for reuse.

The result of this analysis was the identification of four design heuristics which informed the overall structure of the framework. Each heuristic pointed to a basic set of components which needed to be present in order to produce a framework which satisfies the goals of this thesis. Once these basic components had been identified, the details of the framework were fleshed out in two steps:

1. Firstly, this research identified the reuse tasks and requirements engineering activities which the framework should support;

2. Secondly, this research identified the knowledge structures through which could be expressed the data required in order to enact these tasks.

The first of these steps was achieved through further analysis of the requirements engineering literate in order to identify commonly recurring task types. First, an initial set of task types was proposed based on the surveys of requirements engineering literature provided by Nuseibeh and Easterbrook [NE00] and by Cheng and Atlee [CA07]. These requirements tasks were validated further by determining the extent to which existing reuse approaches support each task [NZ11]. Each task was defined concretely in terms of informational properties such as the source and sink or the information which the task transforms and the type of transformation which is enacted. Tasks that were similar were merged into one task.

The second design step involved developing a knowledge model which defines formal knowledge structures through which reusable knowledge, and the requirements engineering methodologies which those knowledge structures support, can be defined. This task was driven heavily by the needs of the generic requirements tasks which make up the procedural guidance that is offered by the framework.

Demonstration. Within the DSRM, the Demonstration stage involves proving the concept of the research by its application to realistic examples. To support this task, a software implementation of the reuse framework was developed. The framework specification is sufficiently formal that the prototype implements the specification directly and so supports all of the activities which are supported by the framework: metamodeling, library specification and engineering-by-reuse. The tool was then applied to formalise two requirements engineering notations: a revised version of Sutcliffe and Maiden's OSM schema [SM98] and a novel representation for software system specifications which is partially based on the UML. Next a reuse library was specified to support the transformations of models expressed as OSMs into models expressed as ISMs. The resultant instantiation of the prototype tool was applied to produce software system specifications, predominantly by reuse, across three complex application domains: the Autopilot, File Transfer and Order Management domains.

Evaluation. The final stage of this thesis was to evaluate the approach. The evaluation of the work presented in this thesis is based on a combination of ethnographic [Hin08], qualitative and quantitative analytical methods. The reuse framework is critiqued on its own terms on the basis of data collected through the case study. In this section of the evaluation, the reuse framework is evaluated with respect to the four criteria which were introduced in Section 1.2 — Generality, Utility, Systematicity and

Practicality — in order to verify that the approach does indeed satisfy the aims and objectives of this thesis. Quantitative and qualitative measures are defined in order to provide as objective a framework as possible for the evaluation of the novel approach, and data is presented for each of these measures in order to support an assessment of the novel approach. This data includes performance profiling data, records of a self-validation exercise which involved the author of this research applying the tool, and a statistical analysis of quantitative data about the range of domains and tasks that were supported. In the second half of the evaluation, the conclusions of the analysis of the approach are used to contrast the framework presented in this thesis to the three major existing requirements reuse approaches: the Domain Theory; POSE; and Requirements Patterns. The novel approach — which is the subject of this thesis — is contrasted to these three existing approaches in terms of Generality, Utility and Systematicity. However, this thesis aims does not aim to improve on the practical feasibility of existing approaches — only to ensure that the novel approach is practical in its own right — and so the novel approach is not compared to existing approaches in terms of the Practicality measures.

Communication. To date, communication of this research has been achieved in two ways:

- This thesis disseminates the research which is presented herein;
- An early form of the analysis on which the design of RORE is based was presented at the First International Workshop on Requirements Engineering Patterns (see [NZ11]).

1.7 Overview of the Thesis

Figure 1.2 provides a roadmap to the remainder of this thesis which is organised as follows.

Chapter Two deepens and broadens the context for this thesis through a systematic review of requirements-level reuse and support for this task in major systematic approaches to requirements engineering. The systematic review first identifies the support for requirements-level reuse that is provided within the context of the four major requirements engineering methodologies, and then broadens the discussion to consider

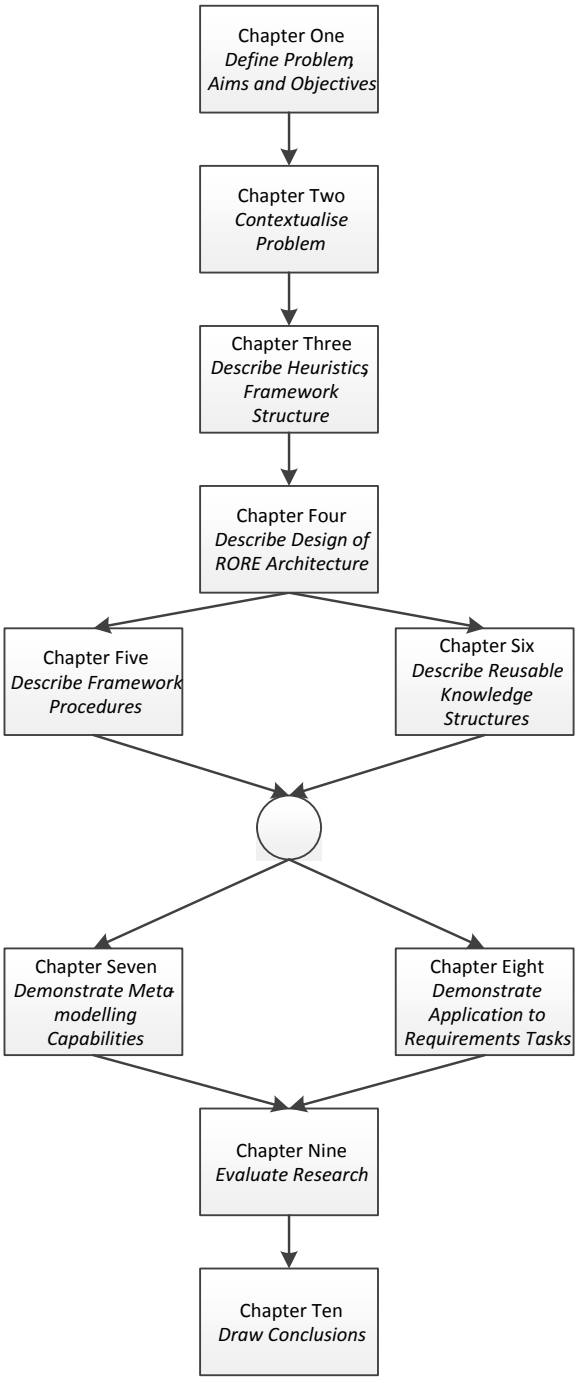

Figure 1.2: Overview of this thesis

less systematic, less widely accepted and less well-developed approaches to reuse. The survey ends with a discussion of retrieval algorithms to support reuse.

Chapter Three introduces the design heuristics on which the design of the RORE approach is predicated, and briefly describes how these were derived from Sutcliffe's conceptual framework for reuse [Sut02] and from the literature on information theory and physical information. The chapter then presents the RORE framework, briefly summarising its major components, and indicating how the design of these components has been influenced by the design heuristics. There are two major components to the RORE framework: a set of generic procedures for engineering new requirements artefacts by reuse, and a set of knowledge types which are used within RORE to express both reusable requirements engineering knowledge and concrete domain-specific knowledge.

Chapter Four presents the design of the RORE prototype. The prototype comprises two prototype tools. The first tool is a meta-modeling tool which supports the formalisation, through the RORE knowledge types, of requirements engineering modeling notations. The tool also supports the definition of reusable knowledge bases. The second tool supports requirements engineering itself by guiding users through the process of engineering new requirements artefacts based on the reuse of knowledge contained within a RORE knowledge base. Chapter Six presents both tools.

Chapter Five presents in detail the procedures which are defined by RORE for engineering requirements artefacts by reuse. Two broad classes of procedure are defined: requirements tasks and reuse procedures. Requirements tasks are high-level activities involving the manipulation and review of requirements artefacts. Reuse tasks are lower-level activities, from which requirements tasks are composed, which support the reuse of reusable knowledge.

Chapter Six drills down into the specifics of the knowledge typology through which requirements knowledge — both reusable and domain-specific — is expressed within RORE. RORE supports meta-modeling of requirements modeling notations, and the instantiation of those meta-models into concrete, domain-specific models. Chapter Five first introduces the knowledge structures by which meta-models are defined, and then describes the knowledge structures through which concrete models are specified.

Chapter Five also defines the knowledge structures through which various kinds of reusable knowledge are expressed in RORE.

Chapter Seven demonstrates the application of RORE to the first of these two activities: the meta-modeling task. The chapter demonstrates how RORE can be applied to engineer reusable knowledge bases for a specific requirements engineering activity: specifically, the task of producing software specifications from requirements models. In addition, Chapter Seven shows how RORE is used to formalise the reusable knowledge which supports that activities. Two notations — the Object System Model and Information System Model — are introduced and formalised. The former represents requirements models; the latter represents specifications. A manual transformation process is described and then formalised as a sequence of reusable knowledge structures

Chapter Eight demonstrates how RORE supports the second of these two activities: the engineering of new requirements artefacts. The chapter illustrates how RORE can be applied to utilise the reuse library that was constructed in the previous chapter in order to produce new requirements models for three different requirements scenarios.

Chapter Nine evaluates the demonstrative work that was presented in Chapters Six through Eight. The evaluation is structures into three parts. The first section provides a critique of the research methodology in order to determine the soundness of the research presented in this thesis. The second section critiques the RORE framework itself on the basis of the case study. The third section then applies this critique in order to contrast RORE with the Domain Theory, POSE and Requirements Pattern approaches.

Chapter Ten summarises the work presented in this thesis and identifies the main conclusions of this research.

Chapter 2

Reuse in Requirements Engineering

2.1 Introduction

This chapter reviews approaches to reuse in the requirements engineering literature. Although a large body of work has been presented on reuse in the requirements engineering literature, there is no consensus within the community as to how reuse can best be achieved at the requirements level. Within the requirements engineering literature a handful of major modeling approaches have been proposed, and some effort has been made to integrate reuse into these frameworks. However, only three requirements engineering approaches (Jackson's Problem Frames approach [Jac01b]; its offshoot, POSE [HRJ08]; and Sutcliffe and Maiden's Domain Theory [SM98, Sut02]) seek to make reuse a systematic driving force in requirements engineering.

Aside from these approaches, patterns as a means of reusing and disseminating requirements knowledge have been a particularly popular approach, as discussed by Naish and Zhao [NZ11]. Furthermore, the emergence of the Requirements Patterns (RePa) workshop series [PMCS11] is indicative of the growing significance of patterns-based research in requirements engineering. However, other authors have proposed alternative approaches based on use cases, domain-specific languages or feature modeling.

Finally a small, but important, area of research has been the investigation of mechanisms to support the retrieval of reusable artefacts from a library. Maiden has proposed the use of analogical reuse as a means of matching generic models across application domains [Mai92]. A wide range of recommender systems have been proposed both within, and without, requirements engineering [AT05] which seek to retrieve reuse

structures in a context-sensitive fashion. Various approaches to taxonomic and query-based retrieval have also been proposed, although research in these areas is less developed.

This chapter is structured as follows. Firstly, this chapter considers the major approaches to requirements engineering and the numerous attempts which have been made in each case to integrate reuse into those approaches. Secondly, this chapter reviews pattern-based approaches to requirements reuse, and considers different ways in which requirements pattern libraries have been organised. Thirdly, this chapter reviews alternatives to patterns as a means of reusing requirements knowledge. Finally, approaches to the retrieval of reusable requirements knowledge from libraries are reviewed in the final section of this chapter.

2.2 Reuse in Requirements Engineering Methods

2.2.1 Reuse in KAOS

KAOS is a formal approach to requirements engineering which was the earliest of the goal-oriented requirements engineering (GORE) methodologies [DFvL91, DVLF93]. A detailed overview of recent research into the KAOS methodology is given in [Lap05]. The basic ontological construct in KAOS is a goal: a statement of the intentions and motivations of actors within the problem domain [VL04]. Goals within a goal model are organised into goal abstraction hierarchies according to AND/OR relationships between goals and sub-goals [MVL97]. The goal model is constructing starting with high-level business goals, which may be vague and ill-defined, and decomposing these until a set of leaf goals has been identified for each top-level goal. Several types of sub-goal are distinguished [DVLF93]:

- Satisfaction goals are concerned with requests issued by agents;
- Information goals are concerned with informing agents of the states of objects;
- Robustness goals are concerned with recovery from exceptional or unexpected sequences of events;
- Consistency goals relate to consistency between the physical and automated components of the system;

- Safety goals restrict the conditions under which an agent state is considered "safe";
- Privacy goals restrict the conditions under which an agent state is observable.

Goals themselves are not operationalisable [DVLF93], in the sense that they cannot be directly enacted by the operations provided by individual agents. However, the aim of goal decomposition is to decompose goals to the extent that they *can* be operationalised [Lap05]. In order to model the operationalisation of a goal model, therefore, KAOS provides two other forms of model: an object model, which defines objects and their properties within the system; and an operational model which defines the services that are provided by agents, and which transform the objects in the object model [Lap05]. Collectively these models comprise objects which represent stateful "thing[s] of interest" which may evolve over time, and which can be "referenced by requirements". The object concept is specialised into entities, relationships, events and agents [DVLF93]. Entities are autonomous objects which exist independently of one another. Events are objects which exist only at instantaneous points in time. Relationships associate objects of all types with one another. Agents are responsible for enacting some action or activity. Actions are operations over objects which are defined in terms of mathematical relations [DFvL91].

Numerous authors have investigated reuse within the context of the KAOS framework. These solutions, however, have generally not yielded systematic approaches to reuse, have not been well integrated into the KAOS framework, and have not been thoroughly developed. Furthermore, the focus of these solutions has overwhelmingly been on the production of goal and other KAOS models, and so the task generality of the proposed solutions is highly limited. Darimont and Van Lamsweerde, for instance, propose a set of formalised refinement patterns to support the process of goal decomposition [DVL96]. Massonet and Van Lamsweerdee have investigated the application of analogical reuse to support the construction by reuse of KAOS specifications [MVL97]. The approach is effective for this task and exhibits the domain generality which arises from analogical reuse in general, but lacks both task and method generality. Semmak, Gnaho and Laleau propose an alternative approach [SGL08]. They extend the KAOS metamodel to support the modeling of variability. This approach does not support the construction of novel requirements artefacts by composition-driven reuse. Instead it supports the reuse of a requirements specification as a coarse-grained artefact by instantiating the specification to support a particular context. This solution supports a requirements family approach, and so opens the door for product-line

design.

2.2.2 Reuse in the NFR Framework

The NFR Framework was first proposed by Chung as a notation and methodological approach to modeling non-functional software requirements (NFRs) in general, and accuracy requirements in particular [Chu91]. The framework was designed in order to support the formal modeling of such requirements and the application of those requirements during design through the use of soft goal modeling [MCY99]. The framework defines a set of "goal sorts" for NFRs which correspond to different types of non-functional requirement (e.g., accuracy, security and reliability requirements). NFR goals can be associated with one another through link types which indicate both the decomposition of higher-level goals into lower-level goals, but also mutually satisficing or contradictory interactions between NFRs. Furthermore, the framework provides detailed methodological and heuristic guidance as to how NFR goal hierarchies can be refined in the first instance, and then evolved to be consistent [MCN92].

A recent development in the NFR Framework literature has been a small body of work which investigates the reuse of patterns and anti-patterns in order to support the production and refinement of NFR models. Supakkul et al have proposed a pattern-based approach to capturing and reusing knowledge about NFRs [SHC+10]. The approach advocates five classes of NFR pattern with each class playing its own role in the process of capturing and then formalising NFRS [SHOC09b]. Goal patterns support the clarification of NFR soft-goals. Problem patterns describe obstacles to realising NFRs. Causal attribution patterns model causes for those problems. Solution/means patterns supporting the capturing of solutions to those problems. Requirements patterns are used to specify NFRs themselves [SHC+10].

Other literature addressing reuse within the NFR framework has emphasised the identification and description of specific patterns and anti-patterns which can be applied using the NFR pattern approach. For instance, Serrano and Sampaoi do Prado Leite have focused on using the NFR Framework to capture knowledge about transparency requirements as patterns [SL11]. Cunha et al have extended this work by considering the challenges involved in using NFR patterns as a means of representing transparency requirements [CSdPLDW13]. Supakkul et al have defined patterns to support the modeling of security threats and vulnerability mitigation requirements [SHOC09a]. Supakkul has also begun documenting some of these patterns and anti-patterns [Sup13]. Work on the NFR Pattern approach, then, is small but growing.

2.2.3 Reuse in i* and TROPOS

i* is an agent-oriented approach to modeling which can be seen as a descendent of Chung's NFR framework. i* can be used to support requirements engineering [Yu97], but is also the modeling basis for the TROPOS requirements engineering methodology [CKM02]. Agent-oriented approaches to requirements engineering are desirable because they bring together functional, quality and procedural concerns around the central concept of an agent in order to support constantly evolving environments [Eri97]. i* provides support both for activities occurring prior to requirements gathering as well as for the requirements engineering process itself [Yu97]. During the earlier phases of requirements engineering, i* is used to model the current domain, and as requirements engineering progresses it can also be used to model the system to-be [Lap05].

The central ontological concept which underpins the i* framework is the actor [Yu97]. An actor is an intentional entity [Yu93]. The intentionality of an actor are modelled as knowledge, belief, goals, abilities and commitment. A particular feature of i* is that it attempts to model agents in a "rich organisational context" [Yu97]; that is, within a network of other actors on which agents may depend to realise their goals. Three types of actor are distinguished by i*: agents, roles and positions [Lap05]. Agents are individual entities which exhibit certain skills or capabilities. Roles are abstract sets of responsibilities and intentions. (An agent may play numerous roles). Positions are "socially recognised" [Lap05] sets of roles.

i* distinguishes two basic kinds of model [Lap05]. Strategic dependency models capture the external interactions between actors within an organisation. Strategic rationale models provide a window into the internal motivations and intentionality of individual actors [Yu97]. Components of a strategic dependency model are related by intentional dependencies [Yu93] of which four types are distinguished according to their subject matter: goal, soft-goal, task and resource. Elements of a strategic rationale model are related by decomposition and means-ends links. Means-ends links define alternative approaches to achieve a goal. Decomposition links specify sub-goals of a goal, as well as tasks and sub-tasks for realising those goals [Yu97].

This thesis identifies only a small number of authors who have considered reuse in the context of i*. Of these, the most comprehensive solution is that provided by Kolp, Giorgini and Mylopoulos [KGM03, KF07, Gio03] who have identified a collection of patterns to support the construction of i* organisational models. Their solution has been abstracted from practical experience and so has been carefully validated. However, their patterns emphasise only the modeling of organisational structures and so are

limited by task generality. Pavan, Maiden and Zhu have developed a pattern language which is expressed in i* for a submarine manuevering system. The language comprises four patterns and defines the relationships between these. However, it is limited with respect to domain generality as it is a domain-specific language. Finally, because i* supports the modeling of soft-goals it is a good candidate for expressing quality requirements. Quality reqirements are known to impact on the architecture of a software system. Pimentel et al. [PLC+12] have, therefore, developed a model-transformation based approach to deriving software architectural models from i* quality models. Their approach focuses specifically on the adaptive systems domain.

2.3 Pattern-based Reuse Approaches

2.3.1 The Domain Theory

The Domain Theory is rooted in Neil Maiden's PhD thesis [Mai92]. There are three significant components to the approach. The first is a meta-schema for expressing domain knowledge [SM98, Sut02]. The schema, which was refined further through the NATURE project [JBR+92], identifies a library of fact types through which knowledge can be expressed (see Figure 2.1).

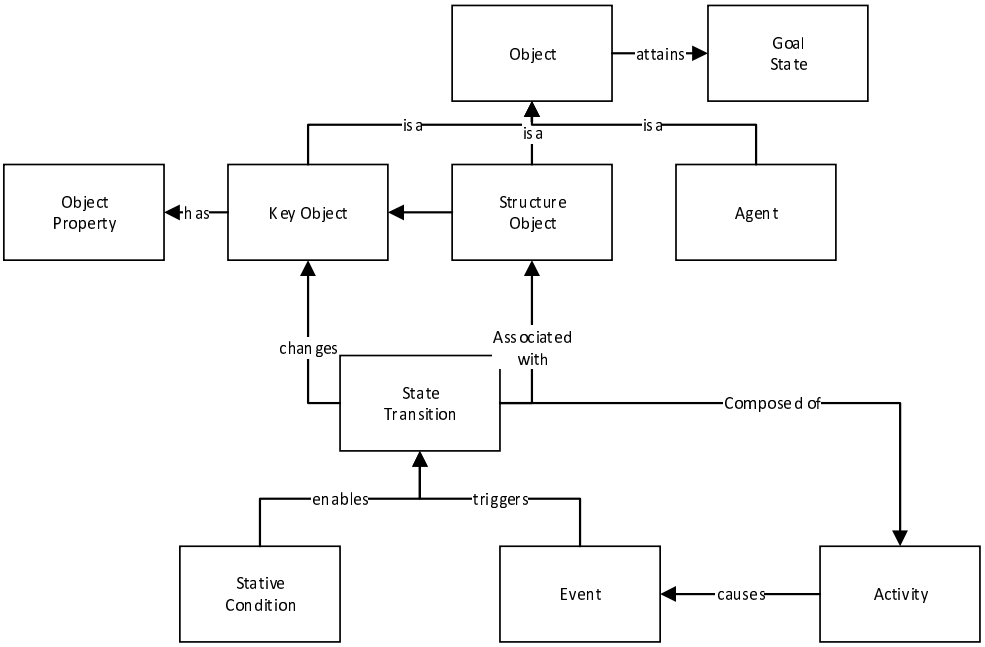

Figure 2.1: The Domain Theory's meta-schema for expressing domain knowledge as described in [SM98]

This meta-schema is instantiated in order to produce requirements models known as Object System Models (OSMs). Each OSM models an application domain in terms of a set of objects and interactions between those objects. Three types of objects are distinguished [SM98]. *Key objects* represent the subject matter of a domain; the objects which are transformed by the processes within that domain. Key objects are further specialised depending on whether they are physical, conceptual or financial. Physical objects are those which have a concrete presence in the material world. *Conceptual objects* are abstract and encode some informational value. Financial objects yield some exchange value in commercial transactions. *Structure (or container) objects* represent structural aspects of the application domain and provide context for key objects. *Agent objects* represent actors within a domain which are capable of interacting with key objects in order to change their state in some way. Key objects are stative objects whose state may be primary (defined by their relationship to a container object), or secondary (internal states defined by the properties of an object). Domain goals are expressed in terms of primary states and are realised by changes in a primary state. OSMs thus encode both the structure and behaviour of an application domain.

The Domain Theory also describes a library of generic OSMs. The library has evolved over more than a decade of research into the Domain Theory [Mai92, SM98, Sut02]. However, the version presented in [SM98] represents a reasonably stable version. The library is organised into a hierarchical structure (see Figure 2.2).

Each OSM represents a generic class of domains. These generic domain classes have been validated by Maiden et al who have determined that they conform to knowledge chunks which occur naturally in the minds of domain experts [MMS95]. This is significant because if an approach is based on reusable structures which neatly conform to the mental chunks of domains then the approach will be highly intuitive and thus offer a high degree of utility.

The Domain Theory orders fact types into a hierarchy according to the discriminatory power which each has: that is, the ability of the fact type to discriminate between two domain models. According to Sutcliffe [Sut02], this is in line with Rosch's theory of natural categories [RMG+76] which holds that chunks within the human mind are organised into categories such that those factors which most distinguish two categories define the hierarchy at the top level with less significant differences distinguishing facts at lower levels.

The top levels of the hierarchy are trimmed because they are excessively abstract and so offer limited utility. The top visible layers of the hierarchy therefore identify

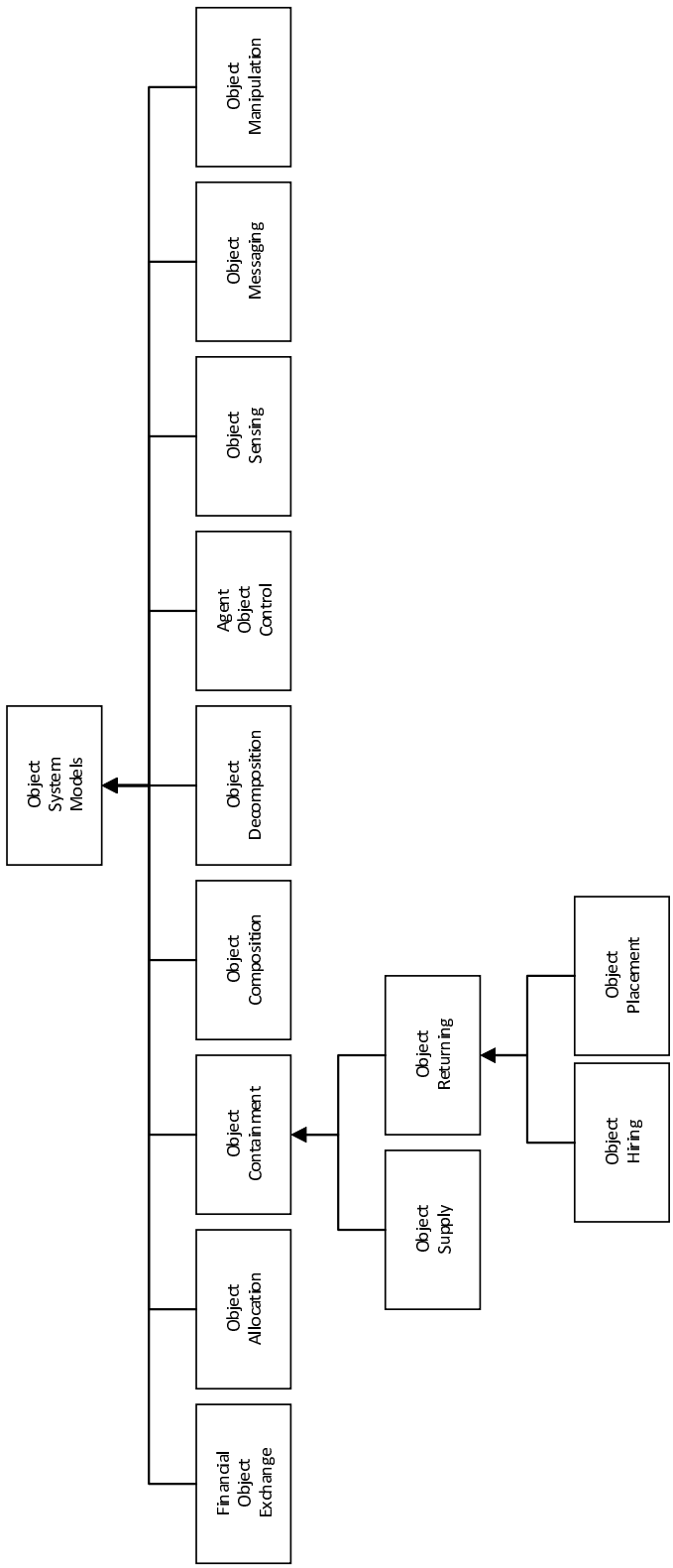

Figure 2.2: The 1998 version of the Domain Theory's library of generic domain models [SM98]

nine generic classes of application domain. These are further specialised by lower-level domain classes. The top-level generic OSMs are distinguished by their high-level structure as denoted by primary states and state transitions [SM98]. At lower levels domains are distinguished by less significant fact types. The second level is distinguished by goal states which are modelled as object structures. The third level is modelled by events and states over those events. Object properties distinguish generic domains at the fourth level.

In order to support the reuse of these generic domain models, the Domain Theory provides extensive tool support: the Advisor for Intelligent Reuse (AIR [Sut02], or the Intelligent Reuse Advisor (IRA) in earlier versions [Mai92]). The tool elaborates requirements artefacts expressed as concrete OSMs through cycles of fact acquisition and matching. The user is first asked to specify a set of fact types of a particular type in accordance with the prioritisation of facts in the OSM hierarchy. A round of matching is then conducted in order to match the concrete OSM to candidate abstract OSMs. AIR has a built in explanation component which presents candidate structures to the user and explains the function of each candidate structure. This allows users to critique the requirements model by determining which model best fits the concrete scenario [MS94]. Once the user has selected the most appropriate abstract OSM, it is integrated into the concrete OSM. A further round of fact acquisition is then performed in order to elaborate the model further [SM98, Sut02].

The domain theory supports two kinds of matching algorithm: analogical and rule-based matching [Mai92]. Analogical matching is used only in the initial round of matching to identify the top-level OSM family which best satisfies the current concrete OSM [Sut02]. Analogical matching is derived from Gentner's structure matching theory of analogy [Gen83] and is also informed by a cluster of other theories from cognitive science (see [Mai92]). As such, analogical matching is used to match concrete models to abstract models on the basis of state transitions with respect to primary states [SM98]. Analogical matching works by ignoring the specific properties of entities within a domain and emphasising instead the structural relationships which exist between entities [MS96]. Two domains may share the same structure but very different properties: borrowing a book from a library and taking a loan from a bank, for instance, are very different on the surface but share in common a basic structure. Analogical reasoning is useful, therefore, because it supports the transfer of knowledge across application domains and so increases the generality of reusable components.

Analogical reuse, however, is computationally expensive [Mai92]. It is for this

reason that the OSM library is organised into a tree structure. Analogical reasoning is used only to match concrete models to the top-level OSMs within this structure. In subsequent cycles of matching, AIR uses a set of rule-based matching procedures to refine the match between the OSM hierarchy and the concrete OSM at lower-levels of the tree structure [SM98]. The rule-based matching procedures are significantly more efficient than the analogical procedure. Overall, therefore, this mixed-model approach to matching avoids AIR utilising analogical reasoning to test all of the abstract domain models in the library. As such, this organisation offers a means to achieve efficiency.

The Domain Theory has a strong theoretical component [Sut02]. Gentner's theory of analogy has naturally informed the analogical reasoning component of the theory, as already discussed. Rosch's theory of natural categories has informed the design of the library hierarchy and of the overall matching procedure. Schank's script theory [Sch82] has strongly informed approach to behavioural modeling taken by the Domain Theory. Furthermore, Sutcliffe [Sut02] cites Yourdon and Constantine's principle of cohesion [YC] as an important influence on the granularity of reusable structures within the Domain Theory. Generic domain models within the Domain Theory are organised such that each generic model describes a single, cohesive, operationalisable goal [Sut02].

Sutcliffe has also outlined in detail a conceptual framework for modeling the reuse problem [Sut02]. The framework describes the relationships between granularity, abstraction, utility, generality and adaptation. (Sutcliffe refers to generality in terms of the scope of a component, but does not use the term explicitly). He observes that both high abstraction and fine granularity maximise the generality of a component, and that high abstraction and fine granularity often go hand in hand. He also observes that both tactics reduce the utility of an approach. Sutcliffe notes that a range of approaches to reuse are possible, but that providing support for adaptation thorough parameterisation balances utility against generality [Sut02].

The final component of the Domain Theory, Papamargaratis has investigated extensions to the domain theory to support application generation [Pap06]. The approach uses a product line approach which maps facts within an input OSM to reusable software-level components which are organised according to a high-level architecture. However, the approach is constrained to the Allocation family of OSMs, and is not readily applicable to other domains. As such, it undermines the basic approach of the Domain Theory, which overall exhibits a high degree of domain generality [SM98, PS04, SPZ06, MMJG05], because it lacks Domain Generality.

2.3.2 Problem Frames and POSE

Jackson has proposed the Problem Frames approach as an approach to requirements engineering [Jac01b]. At base, the Problem Frames approach is a theory about the structure and essence of software engineering problems. The theory is founded firmly on the tenet that engineering artefacts should describe only observable phenomena [HRJ05]. The approach identifies three major artefacts which collectively comprise a software system specification [Jac01b]: a description of the problem domains (K), a specification of the software solution (S), and a specification of the requirements for the software solution (R). According to the problem frames approach [Jac01b], K should describe the significant phenomena within the problem domains. R should describe the effects which it is desired that the machine should produce within the application domain. S should specify the behaviour which the machine will implement in order to produce the effects described in R. In order for software development to prove successful, a relationship should hold between these three artefacts such that the solution described by S would produce the effects described by R when integrated into the domains described by K; that is, the following relationship should be satisfied: $K, S \vdash R$ [HRJ05].

The problem frames approach introduces a novel type of diagram for modeling software problems: the Problem Diagram. The semantics of a Problem Diagram are described fully in [Jac01b, HRJ05]. The basic component of the a Problem Diagram is a set of domains and the relationships between those domains [HRJ05]. Domains may be causal, biddable, lexical or machines [Jac01b]. Causal domains comprise phenomena that form chains of causes and effects; phenomena that behave in a predictable, reactive manner. Biddable domains consist of agents which respond in a predictable manner to behaviour in accordance with a given set of instructions. Lexical domains are causal domains which also exhibit an informational value at an abstract level above the physical level.

Each problem diagram may consist of multiple domains of different types which interact with one another. Interactions are through shared phenomena [Jac01b]. Each domain consists of a set of "domain properties": entities, events, values, states, truths and roles [Jac01a]. In addition to the specification, K, of the domain descriptions, a problem diagram comprises the specifications, S, and requirements, R. The machine is, in essence, a fourth type of specialised causal domain, and so interacts with problem domains within the problem description through shared phenomena. Requirements are also associated with domains and describe properties which their associated domains

should exhibit.

Jackson identifies a set of five reusable and abstract problem diagrams: the five elementary problem frames [Jac01b]. *The Simple Behaviour* frame describes a scenario in which the key requirement is for the machine to control some causal domain. *The Simple Information Display* models contexts where the requirement stipulates that the machine must obtain information from the real world (a causal domain) and display this information on an information display (a second causal domain). *The Simple Information Answers* frame describes a scenario in which the machine must respond to questions from an enquirer (a biddable domain) by gathering information from the real world (a causal domain) and providing a response (a second causal domain). *The Simple Workpieces* frame comprises a requirement which stipulates that the machine should respond to commands from a user (biddable domain) in order to manipulate a computer-processible object (such as text or graphics, a lexical domain). *The Transformation* frame describes a scenario in which some inputs are transfomred in order to produce some output.

These five problem frames are highly abstract in part because, unlike the Domain Theory's generic domain models, they focus on the relationships between problem domains rather than the content of a domain. As a result, the frames can be composed to model a wide variety of more complex application domains [Jac01b]. According to Papamargaritis and Sutcliffe, however, problem frames are "close, but less comprehensive, relatives" of Sutcliffe and Maiden's generic domain models [PS04]. Sutcliffe et al have compared the Domain Theory and Problem Frames directly for the purposes of domain analysis [SPZ06] and drew a number of conclusions:

- That the two approaches are partly complementary: the Domain Theory provides a larger number of abstractions but less analysis advice than the Problem Frames approach;
- The Problem Frames approach helps to identify specification concerns more readily than the Domain Theory;
- The Problem Frames are less intuitive than the Domain Theory's generic OSMs and so their application may "require more thought".

Several other authors have extended or refined the Problem Frames approach in various ways. Lavazza and Del Bianco [LDB06] have integrated the Problem Frames approach with the UML in order to provide a more accessible and familiar notation

for the approach. Liu and Jin have investigated the integration of problem frames with the i* approach in order to support the integrated modeling of subjective intentions as well as hard facts about the application domain [LJ06]. Other authors have focussed on providing tool support for the approach. Jin and Liu have developed an ontology-driven tool to support automatic decomposition of problem models into problem frames [JL06]. Lavazza et al. advocate a manual approach but provide a meta-model to support problem decomposition [LCPCdB10]. Seater and Jackson, D. [SJ06a, SJ06b, SJG07] have described a detailed method for deriving specifications through problem descriptions and describe a mechanical procedure for validating the design rationale by which those specifications were produced.

Few authors, however, have focussed explicitly on the Problem Frames approach as an approach to requirements-level reuse reuse. The Problem Frames approach focuses specifically on modeling the application domain, its requirements and specifications to address those requirements. As such, it exhibits a degree of task generality but is primarily supportive of a horizontally narrow set of tasks. Authors who have focussed on the reuse aspects of the approach therefore extend the task generality of the approach. A common topic is the investigation of approaches to the integration of problem frames with solution space components [RHJN04, SLM$^+$07, WDC06]. Conversely, Wieringa et al investigate approaches to get to the initial phases of problem framing, an approach known as "value framing" [WGvE04]. Lin et al have adapted the notion of a problem frame into that of an "abuse frame" [LNI$^+$03] in order to support the analysis of non-functional security requirements. Similarly Wen et al propose an approach to capturing security problem patterns based on a hybrid of i* and problem frames [WZL11]. Tun et al [TJL$^+$09] have investigated, with positive results, whether or not problem frames can be useful in diagnosing system failures.

These authors confirm that the Problem Frames approach is limited in terms of task generality, but few of these authors have developed their research into systematic extensions to the Problem Frames approach. One approach which bucks this trend, however, is the POSE (Problem-Oriented Software Engineering approach). POSE is a systematic approach to the construction of problem specifications and the progression of those patterns into solutions [HRJ08]. POSE is directly rooted in the Problem Frames approach and shares the basic interpretation of the concept of a problem model: a set of descriptions, $K, S \vdash R$ [HRJ07]. However, POSE additionally introduces the notion of a problem transformation.

POSE transformations are represented as Gentzen Sequents comprising three basic

elements: a conclusion, a premise and a justification [HRJ08]. Like Seater and Jackson's approach [SJ06a, SJ06b, SJG07], Hall et al's approach is primarily intended to support the validation of specifications based on design rationale which directly connects the problem description to the solution specification [HRJ08]. As such, the conclusion describes the initial problem and the premise describes a set of sub-problems which can legitimately be generated from the parent sub-problems. The justification defines the circumstances under which the transformation can be applied. Hall and Rapanotti have developed a Prolog-based tool — POELog — to support the POSE approach [HR08]. POELog provides an engine for firing POELog scripts which are capable of generating POSE problem trees based on the POSE methodology. This, however, is a prototype and requires further development before it is a practical tool.

Although the POE (Problem-Oriented Engineering) Wiki identifies a set of general transformations [Hap13], the POSE approach is sufficiently general that additional transformations can be introduced into the approach. Furthermore, Hall notes that POSE does not commit software engineers to any specific notation for representing either the domain, requirements, or machine specification descriptions [HR08]. As a result, POSE generalises the Problem Frames approach with respect to both task and method generality. This conclusion is further supported by Mannering et al who have shown how POSE can be used to engineer software to support safety concerns [Man10]. The approach is supported by a formal implementation using the Alloy methodology, and is validated by an industrial case study [MHR08].

2.3.3 Patterns to Support Requirements Tasks

Authors in the literature have proposed patterns to support a wide array of tasks. These range across the full gamut of requirements engineering. Scheinholtz and Wilmont, for instance, have proposed patterns to support the elicitation of requirements [SW11]. Renault et al have similarly proposed PABRE, a pattern-based approach to supporting the elicitation of requirements [RMBFQ09b, RMBFQ09a]. Zhao has proposed an alternative approach to early-phase requirements engineering: a pattern-language for producing requirements models from scenario descriptions [Zha11].

At the other end of the spectrum, some authors have developed pattern-based approaches to support latter-phase requirements engineering. Bass et. al, for instance, propose patterns to support the construction of architectural specifications [BJJSS04]. Both Grunske [Gru08] and Konrad and Cheng [KC05a, KC05b] have produced patterns to support the construction of software system specifications. Furthermore, other

authors have investigated pattern-based approaches to the progression of requirements artefacts into design [WM08, XZAR06, KWK11] (see Section 2.4.4 for further discussion).

Most of the literature on requirements patterns focuses on patterns as a means to construct new artefacts; that is, the task of requirements and specification modeling. However, a handful of authors have focused on supporting other kinds of requirements task. A significant task which several authors have supported is the analysis of a problem domain and its requirements. Jackson's Problem Frames are a well-known example of such an approach [Jac01b], as are Fowler's analysis patterns [Fow97]. Other authors have similarly proposed pattern-based approaches to requirements analysis. Giorgini et al focus on the analysis of the organisational structure within a problem domain [Gio03] and Wen et al on security requirements analysis [WZL11].

Other tasks for which pattern-based solutions have also been proposed the process of interviewing stakeholders [SW11, CS06], the dissemination of requirements engineering knowledge [HL05], the decomposition of system requirements [PSP09, Pen11], and the prioritisation of requirements [Wei11].

2.3.4 Domain-Specific Pattern Approaches

Most of the requirements patterns which this author identified were not explicitly constrained to any particular application domain, but instead were scoped by other factors. Nonetheless, a handful of domain-specific approaches have been proposed. Mahfouz et al propose a collection of patterns to support requirements gathering in the service-oriented information exchange domain [MBLN06]. Penzenstadler has proposed a library of patterns to support the automotive domain [PSP09]. Finally Renault et al have proposed patterns to support the construction of Call-for-Tender system requirements [RMBFQ09b].

2.3.5 Domain Independent Pattern Approaches

Another popular approach to pattern library scoping is to gear the library towards requirements for a particular type of quality. Several authors have taken this approach. Serrano et al have produced patterns to support the modeling of "transparency-related" patterns [SL11]. Bass et al focus on usability requirements [BJJSS04]. Grunske's patterns address quality requirements at a higher-level, focusing in general on probabilistic quality properties [Gru08]. Houdek and Kempter take a similarly broad approach

[HK97]. Konrad and Cheng focus on real-time requirements [KC05b], while several other authors have focussed on security requirements [WM08, WZL11]. Xu et al by contrast focus on dependability requirements [XZAR06].

Alternatively, some authors have constrained their pattern approaches according to the views of requirements which they offer. Kolp et al [KGM03] and Giorgini et al [Gio03] provide patterns to support organisational modeling of the application domain. Zhao [Zha11] focuses on multi-perspective modeling of requirements. Finally, a small number of authors have made no explicit effort at all to scope their collections of pattern [HL04, HL05, CS06].

2.3.6 Limitations of Pattern-based Solutions

The pattern-based approach to requirements-level reuse has proven popular because it is extremely flexible. There is no restriction imposed on what knowledge a pattern expresses, or on the notation through which that knowledge is expressed. Because, as this review has shown, a wide array of reusable knowledge is expressed in pattern form, the approach is highly general and can be used to support the construction of requirements and software specifications across a wide range of domains. However, with this flexibility comes some important limitations. In particular, there is no consensus as to tool or process support for pattern-based requirements engineering. This factor, combined with the lack of conformity with respect to representation, abstraction and granularity means that reusers must work particularly hard in order to integrate different requirements pattern approaches. In short, the cost of very high degrees of generality which arises from the adoption of a pattern-based approach to reuse, is a lack of systematicity and a consequent loss of utility.

2.4 Alternatives to Patterns

2.4.1 Use Cases

Another approach to requirements-level reuse is to reuse use cases. Jacobson et al have proposed use cases as an approach to modeling software requirements from the perspective of user interactions with the software system [Jac92]. A small number of authors in the literature have addressed reuse in this context. Biddle et al, for instance, propose the reuse of use cases themselves in a form that is “abstract, lightweight and technology free” [BNT02]. Kamalrudin et al propose a similar approach [KHG11].

Saeki by contrast proposes a pattern-based approach to constructing use cases [Sae00]. Other authors have focused on other aspects of use-case reuse. El-Attar and Miller, for instance, have proposed the use of anti-patterns as a means to improve use case quality [EAM06]. Finally, Wang proposes that use case specifications can be transformed into feature models, providing a requirements engineering approach to product-line initialisation [WZZ+09]. Braganca and Machado also propose an approach to transforming use cases into feature models [BM07]. Chen et al has investigated the use of requirements clustering to construct feature models [CZZM05].

2.4.2 Feature Modeling

Feature modeling was originally proposed by Kang [Kan90] and was elaborated into a coherent methodology over the subsequent decade [KKL+98]. While not expressly a requirements engineering approach, it can be seen as an approach to requirements modeling and indeed some authors have used feature models to represent requirements models [WJ09] (although some authors argue that feature models are inadequate for representing requirements-level variability [BLP04]). Feature models are also not explicitly an approach to reuse. However, the advantage of feature models is that they can easily be reused to represent applications of a similar kind because they explicitly identify optional and alternative features, allowing variation within a family of products to be modelled and specific products to be described as configurations of a feature model. Feature modeling is a popular approach to supporting the specification and configuration of product lines and product line instances [KLD02]. Bittner has explicitly addressed the use of feature models to support requirements-level reuse [BBP+05].

As an approach to reuse, however, feature models are coarse-grained. They model entire families of products and do not readily support compositionality as individual features can typically not be reused [Hol06]. Holmes et al have attempted to address this problem but they cannot get away from the inherent domain-specificity, and consequent lack of generality of features or feature models as a whole. Waldmann and Jones, however, have found that a feature-based approach to requirements reuse can address specific industrial challenges, in particular the need to deliver rapid cycle times [WJ09].

2.4.3 Domain-Specific Languages

Another approach to requirements-level reuse is to develop domain-specific languages [MHS05]. While not an approach to the engineering of requirements, DSLs describe domain problems at a higher-level of abstraction than do solution-space languages. As such, they provide a means of describing the problem in terms that will be familiar to domain experts. Van Deursen et al provide a detailed systematic review of DSL literature [VDKV00]. By their very nature, DSLs lack generality, but it is in the domain-specific philosophy to eschew generality in favour of the development of libraries in narrow domains [NoCBDoIS80, Nei84]. One alternative to generality is to provide support for integrating different libraries.

However, this is difficult in the case of domain-specific languages because there is no consensus on the appropriate level of granularity for DSLs, or on the criteria by which a DSL should be scoped. Some authors define domains in terms of the subject matter of that domain. Van Den Bos and Van Der Storm, for instance, discuss the use of DSLs within the digital forensics domain [vdBvdS11]. Thibault discusses a domain specific language for defining video device drivers [TMC97]. Meanwhile, JAMOOS is a DSL for language processing [GT04]. Other authors, however, scope domains according to technical properties such as GUI styles [ABBC99] or non-functional properties [HM04]. As a result, DSLs not only lack generality but also cannot readily be integrated because the interfaces to DSLs are not readily compatible for integration. Furthermore, the design and implementation of DSLs is highly time-consuming [MHS05, FNP97] and so as an approach to reuse in domains for which a DSL is not already available it is not a viable solution.

2.4.4 Requirements Progression

A significant issue in the requirements engineering literature has been the progression of requirements from problem specifications into design. In 2001 and 2003 the Software Requirements to Architecture Workshops (STRAW) were held to discuss this question. A small number of authors have discussed reuse-based solutions to this question. The most popular approach is to use a mapping approach in which problem-space components are mapped directly to solution-space components [BBGM00, DK11, LM03]. This approach allows the reuse of solution-space components at various levels of granularity as well as the transformation process itself. This approach is akin to the procedural reuse in POSE, and has been advocated both by Gross and Yu [GY01] and

by Rajasree et al [RRJ03]. This allows reuse, not of the process, but of the solution space component itself. These approaches offer varying degrees of generality. Pattern-based approaches are dependent on the availability of patterns to support a particular context, whereas procedural-based approaches will tend to offer a greater degree of generality. However, with the exception of POSE, the approaches are designed to support only a specific task and all assume a particular representation of requirements and solutions. They therefore lack both task and method generality.

2.4.5 Other Approaches

A number of authors have proposed approaches to requirements-level reuse which do not neatly fit into any of those categories of solution which do not neatly fit into those categories of solution which have already been discussed in this chapter. Many of these approaches focus on the reuse of a variety of types of abstract model. Castano et al [CDA93, CDAFP98], and Ryan and Matthews [RM93] each propose approaches to requirements-level reuse based on the reuse of conceptual graphs. Castano et al, however, emphasise the use of requirements-level model abstractions as an aid to the composition of solution space components [CDA93], whereas Ryan and Matthews use conceptual graph matching as a means to compose requirements specifications [RM93].

Other approaches to the reuse of declarative artefact abstractions have also been proposed. Kaindl, Smiałek and Nowakowski argue that case-based reasoning is one possible approach to reducing the up-front effort involved in design-for-reuse [KSN10, ŚKK+10], because it avoids the need for packaging artefacts for reuse. This solution, however, increases the adaptation effort during design-by-reuse. Heumesser and Houdek [HH03], and Von Knethen et al [vKPKH02], have each proposed approaches to systematic requirements "recycling": an approach to reuse in which the domain-dependent and domain-independent aspects of requirements artefacts are distinguished and a specification is given which prescribes *how* a requirement artefact may be reused. These solutions make systematic the ad hoc reuse which is a natural part of software engineering. Cappiello et al introduce another alternative paradigm for declarative reuse — the mash-up [CDM+11] — which is designed to support engineering-by-reuse through the composition of existing artefacts into new ones. The approach gives significant guidance as to the composition and adaptation of reusable artefacts. Lindoso and Girardi's SRAMO technique [LG06] supports the reuse of domain models based on a meta-level ontology which expresses procedural knowledge to guide the

reuse process. Similarly, Lopez et al [LLG02] propose a reuse approach based on the notion of meta-modeling. Each of these approaches is interesting because they provide detailed systematic guidance to inform reuse, and so are distinguished from patterns-based approaches and domain-specific approaches by their emphasis on process rather than product.

Several authors have, however, proposed domain-specific approaches, and as is true of domain-specific languages, these approaches exhibit varying, but low, degrees of domain generality. Authors proposing approaches to support the modeling of systems which realise specific non-functional properties include Csyneiros et al [CWK05] focus on the reuse of knowledge, based on aspects, about the satisfaction of usability requirements. Sutcliffe and Carroll address HCI concerns more broadly, but propose claims as a solution for the reuse of HCI knowledge [SC99]. Toval et al propose an approach to requirements reuse, based on requirements templates and repositories, to support the realisation of security requirements [TNMG02]. Sindre et al also propose a repository-based approach to identifying and modeling security requirements [SFO03]. Finally, El-Maddah and Maibaum propose GOPCSD, a tool which is capable of generating automatically software specifications from requirements models which can be composed by reusing requirements models from different product families [EMM04].

2.5 Reuse Mechanisms

In any effective reuse approach it is necessary to have some mechanism by which reusable artefacts that are appropriate to the current reuse context can be retrieved. Component retrieval technology has evolved significantly over the past three decades, and today a large number of powerful retrieval algorithms are available. In [NZ11], Naish and Zhao argued that a reuse approach should not rely on just one retrieval mechanism as different mechanisms would likely be applicable in different scenarios. Analogical approaches, for instance, are not applicable to the earliest phases of requirements engineering because they depend on the availability of an initial set of facts on which to base a match [MVL97, Sut02]; however, browsing-based retrieval may be.

This section surveys a range of retrieval mechanisms from the requirements engineering literature, and from the reuse literature more broadly.

2.5.1 Recommender Systems

Perhaps the most recent approach to artefact retrieval are the recommender systems which have emerged over the past decade. A detailed survey is given in [AT05]. A wide range of algorithms and technologies have emerged to support component retrieval. Of these, the most significant developments are those approaches which support the retrieval of reusable artefacts from web-based open source repositories. Several authors have proposed such techniques. Code Conjurer, for instance, is an Eclipse plug-in which generates code-level tests from the Integrated Development Environment (IDE) in order to generate queries which are then issued to a test-driven code search engine [HA06, HJA08]. Linstead et al develop a sophisticated approach to the retrieval of code from open source repositories based on a crawling mechanism (Sourceror) and a code ranking algorithm which evaluated both code quality and code relevance [LBN+09]. Ye and Fischer have developed an approach to retrieval which monitors changes within development tools in order to construct queries which can be issued to a repository tool in order to retrieve context-relevant reusable artefacts [YF00]. The approach has been validated through empirical studies which show that the approach allows engineers to discover reusable components "they did not even know existed" [Ye02]. Other approaches include:

- Rascal [MCK05], which uses group usage histories to recommend components to specific developers;
- Citation-based component recommendation [CZW+11];
- Ant colony based rule generation for component retrieval [BDJ10].

These approaches are an indication of the role that web and open source technologies can play in significantly increasing the power of search technologies as well as the availability of reusable artefacts.

2.5.2 Match-based Retrieval

Recommender systems can be seen as an evolution of more traditional match-based approaches. Match-based retrieval uses information contained within an existing artefact to retrieve new or additional knowledge. This involves the calculation of a match score for a set of candidate reusable artefacts based on the number of facts that can be matched between the concrete and reusable artefacts. A number of general approaches

can be identified. Of these, analogical matching has received significant attention in the requirements engineering literature over a decade [Fin88, Mai92, SM98, MVL97, CJ97]. Analogical matching, which derives from Gentner's theory of analogy in cognitive science [Gen83], matches concrete artefacts to abstract reusable artefacts based on a shared deep structure between the two. An important reason for its popularity is that it has been shown to support the transfer of knowledge between application domains [MVL97, Sut02].

Related to analogical matching is similarity matching [CDA93, RM93, Jur95]. Similarity-based matching is also derived from the work of Gentner [GM97], who proposes similarity as a counterpoint to analogy. Whereas analogy is based on structural relations between entities, similarity is based on properties of entities. Similarity, however, has proven significantly less popular in the requirements engineering literature than analogical reasoning because its emphasis on entities rather than structure mandates a much greater conceptual closeness, and so works only between similar domains and not across domains.

A number of other approaches to matching have also been proposed in the literature but have not received widespread attention. Finkelstein proposes matching algorithms based on chains of semantic reasoning and generalisation [Fin88]. Reubenstein [RW91] and Gomaa have proposed matching approaches based on shared attributes defined over both the source and candidate artefacts. Particular approaches have emphasised goal matching [GY01, SFO03] and requirements-type matching [MBFQ08, RMBFQ09a]. Rolland [RPR98] and Maiden [Mai92] have also proposed rule-based matching which uses logical propositions defined over the candidate structure and tested against the source to determine a match. Additionally, some authors have addressed lexical matching to support the matching of natural language artefacts [GB97, KSN10].

2.5.3 Query-based Retrieval

Query-based retrieval provides an alternative to recommender and matching systems. Retrieval is distinguished by the fact that in a query-based mechanism the information on which retrieval is based comes from a user directly and not from a source artefact. Typically, queries are specified in terms of attributes of the artefacts within the library. There is little consensus in the literature as to the criteria or terms through which a query should be expressed. Prieto-Diaz advocates the use of low-level solution attributes such as "function" and "algorithm" to structure queries [PD91]. Poulin

also advocates the use of solution-space attributes but at a higher level of abstraction [Pou95]. Sindre uses goals to support the retrieval of non-functional requirements [SFO03] whereas SRAMO uses semantic relationships between patterns [LG06]. Finally, Gomaa uses domain object properties to structure the search query [Gom95].

2.5.4 Taxonomic Retrieval

The final approach to retrieval is based on taxonomic browsing. Taxonomies define conceptual hierarchies which users can browse to identify artefacts that are relevant to particular categories. Taxonomies are highly flexible, and allow users to explore a repository by moving from abstract concepts to lower-level concepts. They require, however, a large degree of user effort to retrieve components. Furthermore, their design is highly subjective and can often be contentious. Gorsheck has proposed the classification of requirements according to the level of technical detail they contain [GW06]. Waldmann's approach categories requirements artefacts by artefact type [WJ09]. Finally, PABRE [RMBFQ09a] and Mendez-Bonilla each advocate the classification of requirements based on their own respective typologies.

2.6 Summary

This chapter has reviewed literature within the field of requirements engineering, and related areas, which addresses reuse at the level of software requirements. The approach which has overwhelmingly dominated the discussion has been the use of requirements patterns. A considerable body of literature now exists which describes a range of approaches to pattern-driven requirements engineering from those authors who have simply chosen to describe individual patterns, or small libraries of patterns, through to a small number of systematic approaches to pattern driven requirements engineering. In particular, Jackson's Problem Frames and Sutcliffe and Maiden's Domain Theory are two comprehensive approaches which support the construction of complete functional requirements models predominantly by reuse, while Supakkul et al's NFR Patterns Approach supports the construction of non-functional requirements models. Patterns are a powerful approach to the construction of software requirements because they are sufficiently fine-grained and abstract to support the capture of requirements across several application domains, and to support the composition of new requirements models by reuse. However, there is a trade-off because this abstraction entails

a small utility cost as patterns must be adapted and this mildly increases the effort involved in applying patterns.

Alternative approaches, such as domain-specific languages and product-line approaches, overcome this limitation since they enable requirements engineers to describe requirements at precisely the right level of abstraction for a specific application domain. However, because they are constrained to a specific domain the abstractions which domain-specific approaches provide cannot readily be transferred to support new domains. When encountering new problems, therefore, requirements engineers who have adopted a domain-specific approach to reuse may need to familiarise themselves with a new modeling language or reuse library: a time consuming process.

This thesis concludes, therefore, that assuming the availability of a well-structured catalogue of patterns with effective retrieval support, pattern-based requirements engineering can offer a strong balance between generality and utility. However, the open problem in this area is the availability of such libraries to support a range of requirements engineering tasks and domains.

Chapter 3

The Conceptual Framework

3.1 Introduction

This chapter provides an overview of a Reuse-Oriented framework for Requirements Engineering (RORE) which aims to support the construction of a range of requirements artefacts by reuse. This section proposes RORE both as an approach to requirements engineering which supports a systematic approach to reuse, and also as a framework which amalgamates existing knowledge from the requirements engineering literature into a single, coherent approach. RORE addresses the limitations which Chapter 2 identified in two important ways.

- RORE aims to address the utility cost which patterns pay as a result of their abstraction and fine granularity, and as a result of the difficulty of developing a general and systematic approach to pattern application. RORE offers detailed, formal procedures for both retrieving and reifying reusable knowledge structures for a particular context and so offers a high degree of utility. A prototype tool further improves this utility by significantly reducing the amount of manual effort required to apply the RORE approach to generate new requirements models.

- RORE aims to address the lack of generality offered by domain-specific languages and pattern libraries. RORE utilises a a range of design heuristics (described in Section 3.3) to maximise the generality of the RORE approach without undermining utility.

RORE also has the following three important design features, which collectively are intended to ensure a high degree of generality and utility in the RORE approach:

1. Within RORE, requirements artefacts are produced by a sequence of fine-grained refinements, such that each refinement is achieved, where possible, by reuse. Novel refinements — those not reused from a knowledgebase — are utilised only as a last resort, and are achieved by giving the requirements engineer the opportunity to utilise their own creativity in satisfying a refinement goal.

2. RORE is non-prescriptive with respect to either the kind of artefacts that need to be constructed during requirements engineering, or the representation of those artefacts. RORE identifies the components that are invariant and variant across notations and methods. RORE has only knowledge of invariant components, and can be parametrised with knowledge structures that account for variant components. In this way, RORE is able to treat a range of different approaches to requirements engineering uniformly, allowing the approach to be reified for the needs of a particular context.

3. RORE serves as a framework by which existing knowledge within the requirements engineering literature can be organised and contrasted.

The remainder of this chapter is structured as follows. Section 3.3 introduces the rationale which underpins the design and development of the RORE framework for requirements reuse. Some basic design heuristics are presented and the influences on RORE are described. Section 3.5 presents a conceptual overview of the RORE framework and introduces the major layers, and actor perspectives, on this framework. RORE comprises a suite of generic procedures to support requirements engineering by reuse and Section 3.6.3 briefly introduces these procedures. Finally, Section 3.7 briefly summarises the major knowledge structures through which reusable requirements knowledge is expressed within RORE.

3.2 Assumptions

The version of RORE which is presented in this thesis, and the arguments which made about this version of RORE, are rooted in the following assumptions:

1. That RORE, as it is described in this thesis, will be applied for the engineering of requirements for software systems;

2. That RORE, as it is presented in this thesis, will be applied to generate functional, rather than non-functional, requirements models;

3. That the RORE approach, as it is presented in this thesis, shall only be applied under ideal conditions such, namely within domains for which a comprehensive knowledgebase of RORE-formalised reusable knowledge structures is available and stakeholders always respond appropriately and concisely to stimuli which are put to them by the requirements engineer;

4. That the reusable knowledge structures which are contained within a knowledgebase have been empirically validated by the knowledge engineers who design the knowledgebase in order to validate their generality and utility;

5. That the reusable knowledge structures which are specified within a knowledgebase, where they are applied, will always produce accurate inferences (as verified by the requirements engineer and stakeholders) over a target mode.

These assumptions would not, in fact, hold up in the real world and as such mechanisms would need to be built into a commercialisable or practical version of RORE in order to support the validation of inferences by stakeholders and the requirements engineer. However, these assumptions have been chosen in order to limit the scope of this thesis so that this research can focus on the task of demonstrating that the mechanisms which underpin RORE — such as a set of generic requirements engineering tasks and a set of reusable knowledge structure types — are sufficient to provide the balance between generality, systematicity, utility and practicality which was argued for in Chapters 1 and 2.

3.3 Design Rationale and Heuristics

3.3.1 Reuse as Efficient Information Retrieval

Requirements engineering is a process of information acquisition and transformation. A requirements engineer must elicit from some source information about an application domain, and reason over this information in order to determine requirements for a software system. This thesis therefore views requirements engineering, fundamentally, as a process of "Information Gain". Requirements-level reuse is one approach to achieving this information gain. Specifically, requirements-level reuse retrieves information about application domains and their software requirements from libraries of reusable artefacts. This is as opposed to the manual requirements engineering process

in which the same information must be elicited from stakeholders, existing documents and through observation of real-world processes.

Chapter 2 presented different reuse-driven mechanisms for gaining the information which is of relevance to a particular requirements engineering context. However, different mechanisms require more or less information about that context to be specified up-front against which a retrieval mechanism can match reusable artefacts. As discussed by Maiden [Mai92], the task of specifying this information can be time consuming and so it is desirable to minimise the amount of information which a user must specify upfront in order to get good quality information back from a reuse library.

Drawing on this line of reasoning, Naish and Zhao [NZ11] introduced the notion of "Efficient Information Gain" as one metric against which retrieval mechanisms can be evaluated. The metric can be quantified in the following way:

$$Information_{in} : Information_{out} \tag{3.1}$$

Where $Information_{in}$ is the quantity of information, quantified as Shannon entropy [SW62], which a requirements engineer must specify up-front in order to get a match, and $Information_{out}$ is the average quantity returned by the retrieval mechanism.

This is an important metric because it is closely tied to utility. Manually specifying information about the application domain requires significant effort on the part of a requirements engineer. Therefore, as $Information_{in}$ grows relative to $Information_{out}$, the requirements engineer will have to invest more effort in order to get the same amount of information back. The metric is naturally insufficient, by itself, to support an effective evaluation of retrieval mechanisms because the return of information is only useful if it is relevant to the requirements engineering scenario. However, the metric can be a useful heuristic for evaluating the utility which a particular reuse mechanism will offer.

3.3.2 Procedural versus Declarative Reuse

The literature survey presented in Chapter 2 introduced both the Domain Theory [Sut02] and Problem-Oriented Software Engineering [HRJ08]. These two approaches can be viewed as archetypal examples of two alternative paradigms for requirements-level reuse: declarative and procedural reuse respectively. This thesis characterises these two paradigms in the following way:

- **Declarative Reuse** is a paradigm in which the declarative facts which form the

contents of a requirements model is reused, either as-is or after some reification;

- **Procedural Reuse** is a paradigm in which structures that define transformations over requirements models are reused.

This thesis claims that procedural reuse is inherently more general, and without an inherent generality cost, than is declarative reuse. This thesis justifies this claim as follows.

Imagine two machines (shown in Figure 3.1) which produce a refined, concrete model chunk from an input model chunk.

Let the first machine (on the left of Figure 3.1) be called the **Look-up machine**. The Look-up machine determines a refined chunk by searching the left-hand column in a look-up table in order to find an abstract chunk which matches the concrete input chunk. If a match is found, then the chunk given in the right-hand column of the matched row of the look-up table is returned as the output chunk.

Let the second machine (on the right of Figure 3.1) be called the **Calculating machine**. The Calculating machine determines a refined chunk by following a logical procedure which derives the output chunk through a sequence of logical transformations from the input chunk.

These two (hypothetical) machines could be used to provide the basis for a reuse-driven approach to the transformation of, for instance, a requirements artefact as follows:

1. The requirements artefact is input into the machine as the (unrefined) input chunk;

2. The machine then reasons over that artefact (using either the Look-up or the Calculating procedure);

3. The machine returns an output model chunk which refines, in some sense, the requirements artefact which was input to the machine.

As a final step, the output model chunk would need to be integrated back into the requirements artefact to complete the process.

The Look-up machine offers an implementation of Declarative Reuse because it is based on reusing predefined, declarative chunks of a model. The Calculating machine reuses a logical procedure which derives the output model chunk from the input artefact and so implements Procedural Reuse.

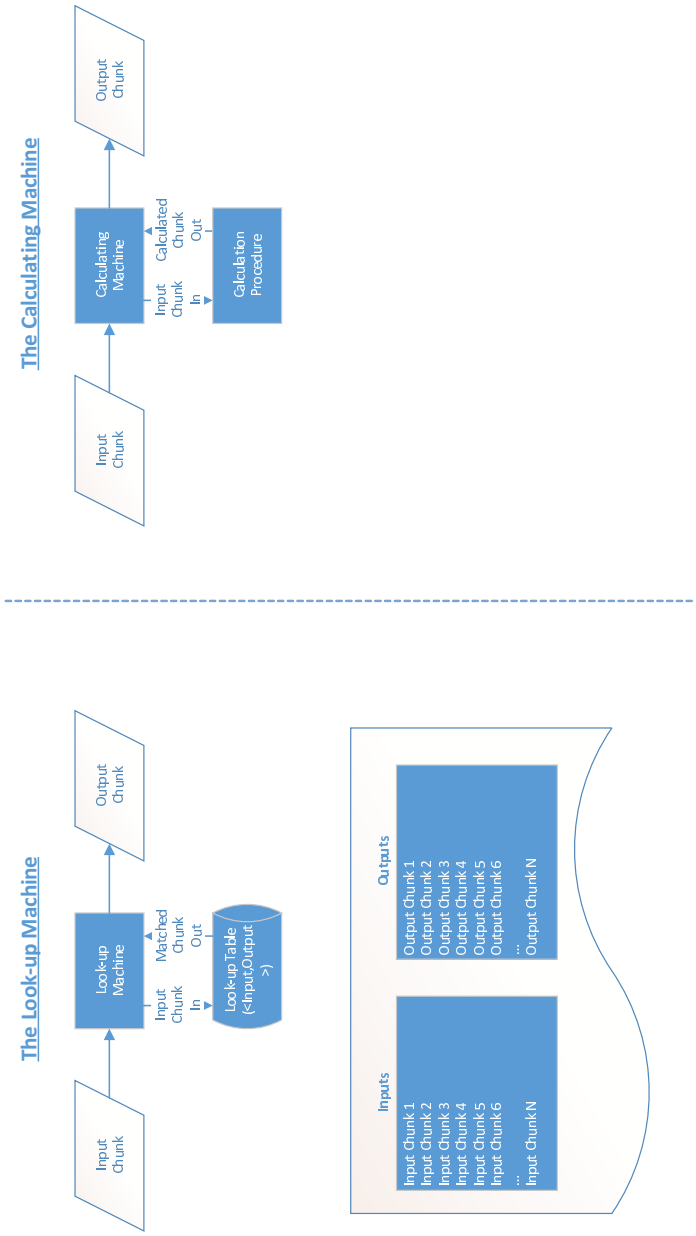

Figure 3.1: Comparison of the Look-up and Calculating machines, showing general structure of the look-up table for the Look-up procedure

If the term "generality" refers to the range of input chunks for which a machine can produce a valid output chunk, then it will be immediately apparent that the Calculating machine must be at least as general as the Look-up machine and will, in fact, be more general for realistic domains. This point can be confirmed in the following way. The Look-up and Calculating machines can be seen as two different implementations of an abstract reuse function, $o = f(i)$ where o is the refined output chunk and i is the unrefined input chunk. Conceptually, the function is a matching process which maps possible input chunks onto the corresponding output chunks (see Figure 3.2):

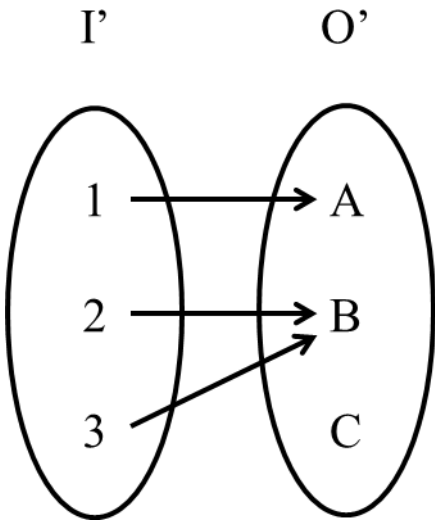

Figure 3.2: Conceptualisation of a reuse function which maps input model chunks to refined output chunks

However, the two separate machines differ in terms of their realisation of this conceptual view: the Look-up procedure (which implements Declarative Reuse) offers a literal implementation of this mapping through its internal look-up table, whereas the Calculating procedure (which implements Procedural Reuse) encodes the mapping in the calculation procedure. This distinction reveals the important and inherent limitation of the Look-up machine: that it can produce a refined output chunk only for a finite range of input chunks. This is necessarily the case because the look-up table must, itself, be finitely sized and so can only match a finite number of inputs to a finite number of outputs.

However, in practice the look-up tables on which the Look-up machine depends must be designed by a human designer. Therefore, the look-up table on which the Look-up procedure depends must in the first instance have been calculated in the sense of the outputs having been derived logically by reasoning about the input domains

over which the reuse function should operate. For any sound look-up table, therefore, there must exist an equivalent Calculating procedure which is capable of producing an output for each corresponding input in the look-up table. This Calculation procedure must be *at least* as general as the corresponding Look-up procedure because it must be capable of producing an output for all inputs within that look-up table.

Whether or not a Calculation procedure is *just as* general as (i.e. no more, and no less, general than) the Look-up procedures for which it can produce a look-up table is dependent on whether or not it is possible exhaustively and finitely to enumerate all of the possible concrete model chunks which might be input to either machine. In domains for which a finite and countable set of input models can be enumerated, then it will be possible to construct a look-up table which can handle all instances within that domain. As such, it would be possible to construct a Look-up machine for such domains which was no less general than a corresponding Calculating machine. In practice, however, it is rarely (if ever) the case that the condition is satisfied that the range of possible variants of a reference model for an application domain can be finitely enumerated. As such, Calculating machines — which do not require all possible input chunks to be enumerated a priori, but instead derive an output by transforming whatever input is provided — will be, for most domains, more general devices for producing a refined model chunk than the corresponding Look-up machine.

3.3.3 Influence of Heuristic Classification on RORE's Procedures

The procedures described in this chapter are also founded on Clancey's Heuristic Classification [Cla85]. Heuristic Classification was chosen because this research found that it provides an effective framework for modeling the process by which reuse is typically achieved. Heuristic Classification describes a class of knowledge-driven problem solving approaches which solve problems in three steps. Firstly, a concrete problem context is matched to an abstract problem in memory. Secondly, the abstract problem is reasoned about to determine an abstract solution. Finally, the abstract solution is reified in order to fit the specific details of the concrete problem context and is applied to resolve the problem. Each requirements task in RORE can be seen as an application of Heuristic Classification in that it consists of three basic phases: the retrieval of a knowledge structure, the firing of that knowledge structure by an appropriate reasoning mechanism and the reification of the resulting solution to fit the details of the current model. In this way, RORE achieves the construction of models through incremental steps which are driven predominantly by reuse.

3.3.4 Design Heuristics

In order to support the design of RORE, this thesis attempted to draw some of this rationale into a set of four of design heuristics to guide decisions about the trade-offs between generality and utility. These are briefly summarised below.

1. Declarative and Procedural Reuse Complement One Another. This thesis has introduced the paradigms of declarative reuse and procedural reuse. Section 3.3.2 argued that procedural reuse is inherently more general than declarative reuse. However, there is a trade-off here. Whereas declarative reuse supports the reuse of the end-point of a reasoning process, procedural reuse describes transformations themselves. When using procedural reuse, therefore, a requirements engineer needs to enact a transformation in order to acquire the information which it offers. "Computationally", therefore, declarative reuse is more efficient at reuse-time because less additional reasoning needs to be done in order to acquire the same unit of information. This thesis advocates, therefore, a mixed-model approach to reuse in which declarative and procedural reuse are treated as complementary, rather than competing, approaches. In particular, this thesis advocates the use of procedural reuse as a fall-back mechanism which can be employed to infer information in cases where no declarative reusable artefact is available. This thesis claims that this approach offers a more effective trade-off between computationally efficient reuse and generalised reuse than can be achieved by adopting either paradigm individually.

2. Gain Information Efficiently. Design-by-reuse can be viewed as a particular approach to information acquisition in which information is retrieved from specifically-designed reuse libraries. In order to retrieve a chunk of information from a knowledge base, a reuser needs to be specify upfront a certain amount of information against which reusable artefacts can be matched. This might be in the form of an explicit query, or, for instance, in the case of analogical reasoning [Mai92] this might be in the form of an (incomplete) pre-existing model chunk. A negative correlation is to be expected between, on the one hand, the amount of information which a retrieval mechanism requires in order to return a good quality match and, on the other hand, the utility of the approach. This is because retrieval mechanisms which require a large amount of information to be specified upfront in order to acquire a good match increase the effort which users must invest in order to acquire information, and so decrease the utility of the approach. Retrieval mechanisms should, therefore, support the efficient retrieval

of information in the sense that the mechanisms minimise the amount of information which needs to be specified upfront while maximising the information gain.

3. Maximise Information Gain. Given the definition of information efficiency above, in order to gain information efficiently, it is desirable to maximise the amount of information which can be gained in each reuse operation. It should be noted that this principle constrains the potential generality of individual reusable artefacts. Sutcliffe [Sut02] has motivated the principle that, in order to support generality, designers-for-reuse should minimise the granularity and maximise the abstraction of reusable artefacts. However, since doing so reduces the amount of information which those components contain, the principle that information gain should be maximised imposes a theoretical minimum limit on granularity and a maximum limit on abstraction.

4. Solution Structures Support Requirements Engineering. "Solution structures" can support requirements engineering in general, and reuse-driven requirements engineering in particular. This thesis uses the term "solution structure" both to refer to software structures which capture design and architectural solutions to software requirements, and — more generally — to refer to any structure which captures a refinement to, or extension of, some part of a requirements artefact. Presenting solution structures to requirements engineers can support reuse-driven requirements engineering in at least two ways.

Firstly, lower-level solution structures can play a role in clarifying higher-level requirements and specification structures. Guindon [Gui90] has conducted a rigorous verbal protocol analysis of software engineers engaged in requirements analysis, specification and design tasks. He found that, far from being a linear and top-down process, the subjects engaged in an iterative and opportunistic design process comprising two main features: that where software designers recognised familiar sub-problems at the requirements level, they were able to recall low-level design structures and so bypass the top-down design process for that sub-problem; and that working on low-level solution structures often gave rise to new requirements or prompted clarification of ambiguous requirements. The results of Guindon suggest that presenting requirements engineers with lower-level requirements specifications, or even software design structures, can facilitate the process of identifying and clarifying higher-level requirements.

Secondly, Maiden has examined the use of domain abstractions for the purpose of requirements critiquing [Mai92, MS94]. According to Maiden, abstract domain

abstractions describe abstract categories of scenario which requirements engineers can use in order to classify, and thereby reason analogically about, concrete requirements scenarios. Where a particular concrete requirements scenario can apparently be matched to more than one such domain abstraction, the requirements engineer is compelled to consider carefully which of the possible candidate domain abstractions most accurately characterises the concrete requirements scenario. The abstract domain abstractions, in this case, represent "solution structures" in the sense that they describe possible, and alternative, refinements of the concrete requirements artefact. The process of choosing which of these refinements should be applied to the concrete requirements artefact forces the requirements engineer to clarify precisely what the requirements should be for the current application domain.

Given the function which different kinds of solution structure can play in supporting the clarification and elaboration of requirements for software systems, therefore, the RORE approach to requirements engineering should provide a mechanism which enables requirements engineers to consider alternative reusable knowledge structures and thereby encourage them to consider alternative requirements specifications.

3.4 An Exemplar: The Package Router Problem

In order to illustrate the following discussion of the RORE approach, a running example is given. This thesis considers how Jackson's Problem Frames approach — including the notation for Jackson's Problem Models, as well as Jackson's elementary Problem Frames — can be formalised using RORE and then how this approach might be applied to a simple scenario. The meta-model for Jackson's Problem Model notation (as it is utilised in this thesis) is given in Figure 3.3.

The scenario which this thesis will use to illustrate requirements engineering aspects of the RORE framework is the Package Router Problem, taken from [HRJ08]. Hall et al summarise the package router problem as follows:

> A package router is used to sort packages according to barcoded destination labels affixed to the packages. Packages slide under gravity through a tree of pipes and binary switches into bins that correspond to regional areas. The problem with which we are concerned is the design of a software controller to ensure the following:
>
> - Packages are routed appropriately, with misroutes reported;

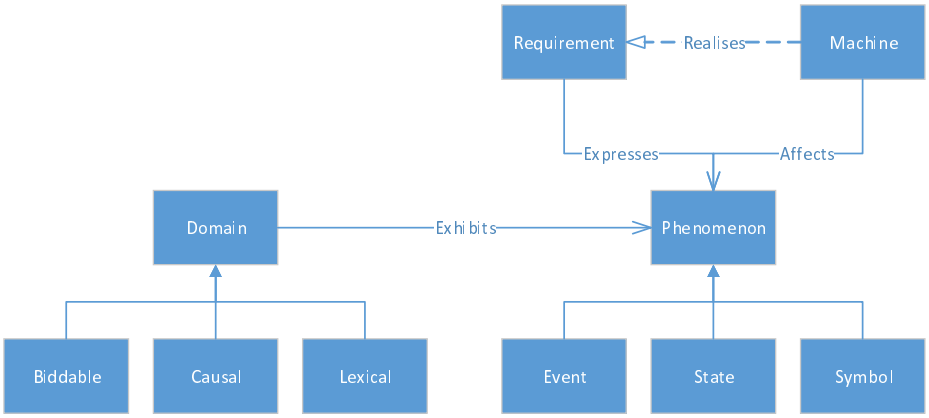

Figure 3.3: A simplified meta-model for Jackson's Problem Diagram notation

- The Operators commands to start and stop the conveyor are obeyed.

Hall et al's rich picture representation of the Package Router problem, and a novel Jackson Problem representation of the scenario, are given in Figure 3.4.

3.5 The Component View

At the highest level, RORE comprises four main components (see Figure 3.5). The first of these is a set of three invariant requirements tasks (and sub-tasks which specialise these tasks). These requirements tasks comprise a model-oriented process for the incremental acquisition of requirements knowledge, predominantly by reuse (see Section 3.6). The requirements tasks are organised into an Analysis-Action cycle in which the analysis task involves planning the action that is to be taken during the remainder of the cycle.

These requirements tasks depend on two knowledge bases (see Section 3.7): working memory and long-term memory are their cognitive science homologies. Working memory is a knowledge base which holds the temporary work pieces over which the RORE method operates during a specific RORE session. These work pieces include a source model and a target model, in addition to an information requirement defining the goal of the current Analysis-Action cycle, and structures which situate the current Analysis-Action cycle within a higher-level requirements engineering method. Long-term memory is a knowledge base holding persistent knowledge structures which define a method for requirements engineering (Process Specification) and its associated model types (Model Specification), as well as reusable knowledge structures (Reuse Library) which are used to parametrise each of the the requirements tasks.

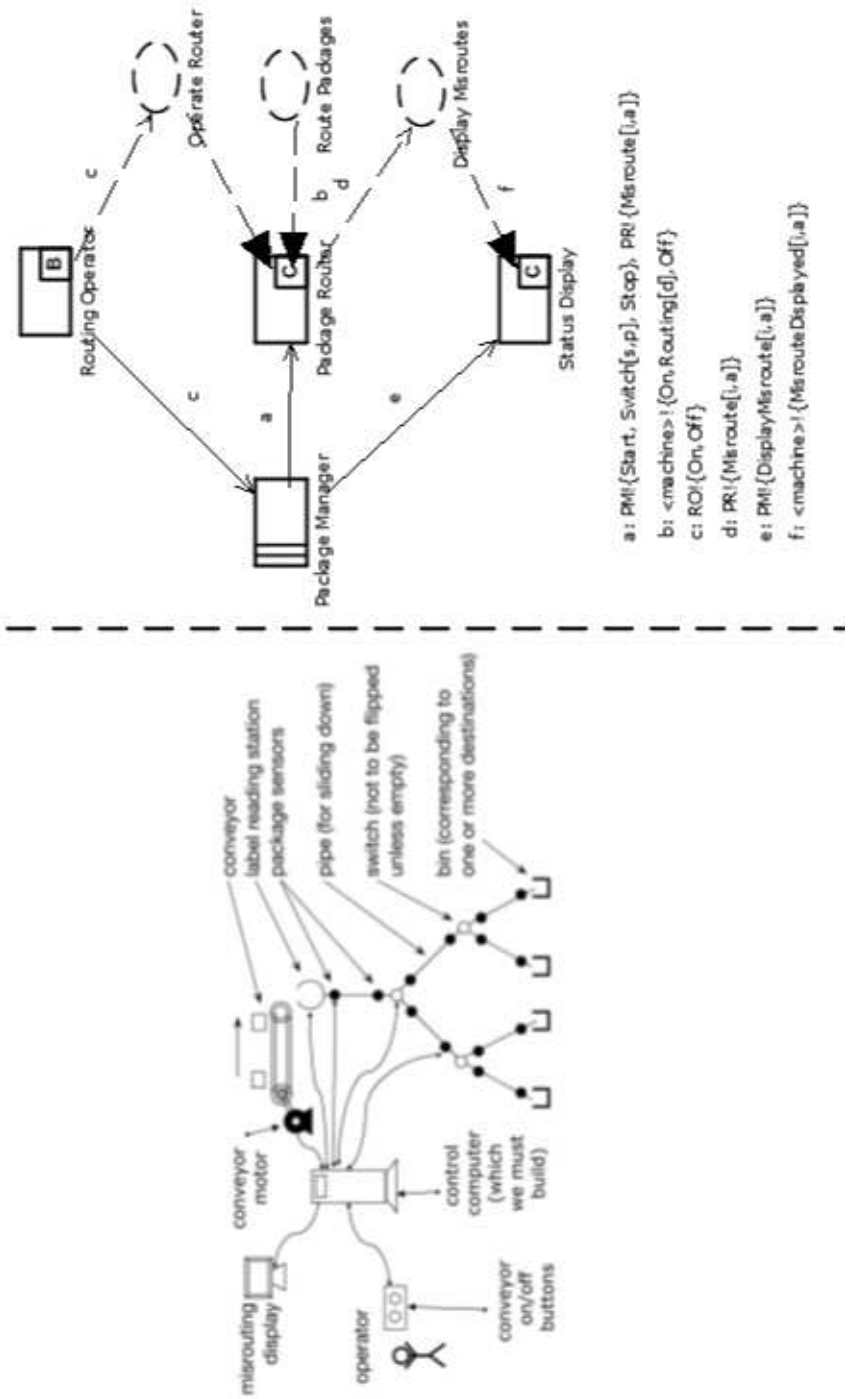

Figure 3.4: The package router problem shown as a rich picture (left, from Hall et al [HRJ08]) and a Jackson problem diagram

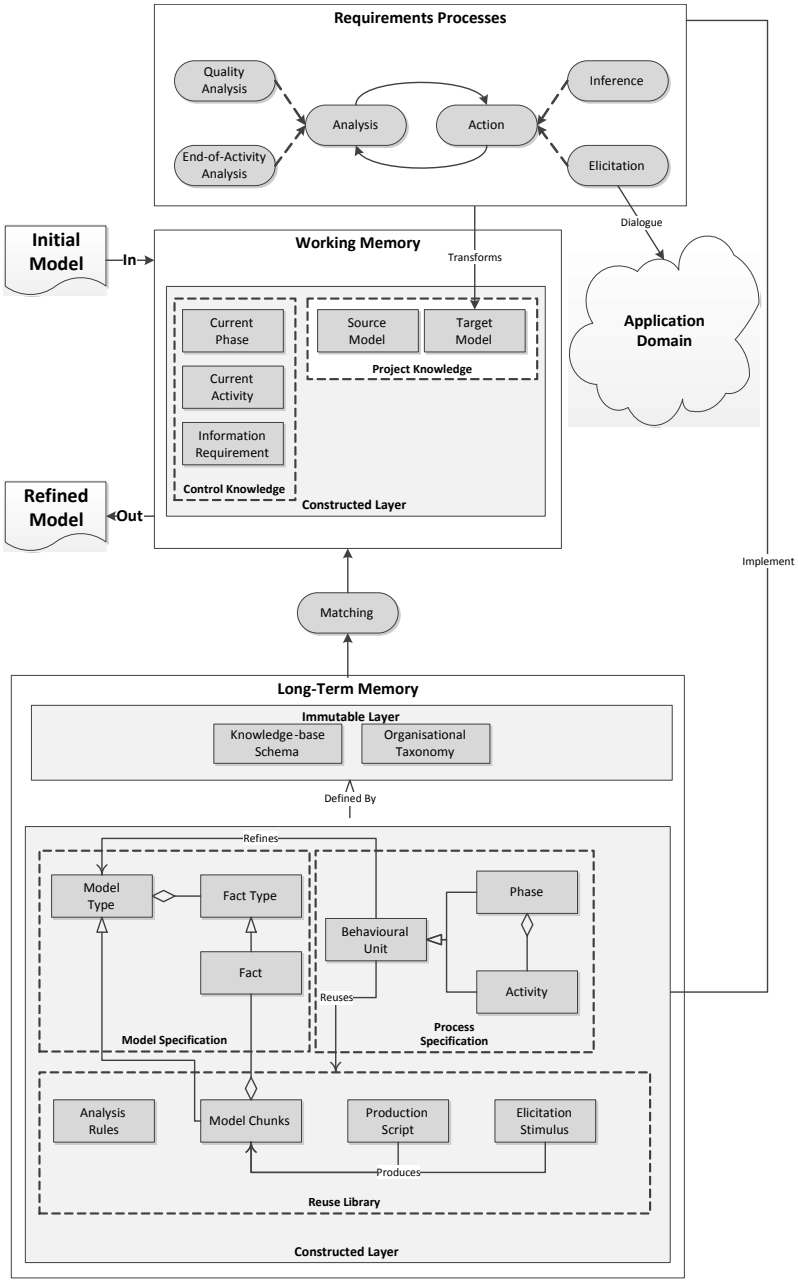

Figure 3.5: High-level architecture of the requirements engineering framework

3.5.1 Layers of the Framework

At the highest level, the RORE framework comprises three major layers as shown in Figure 3.6.

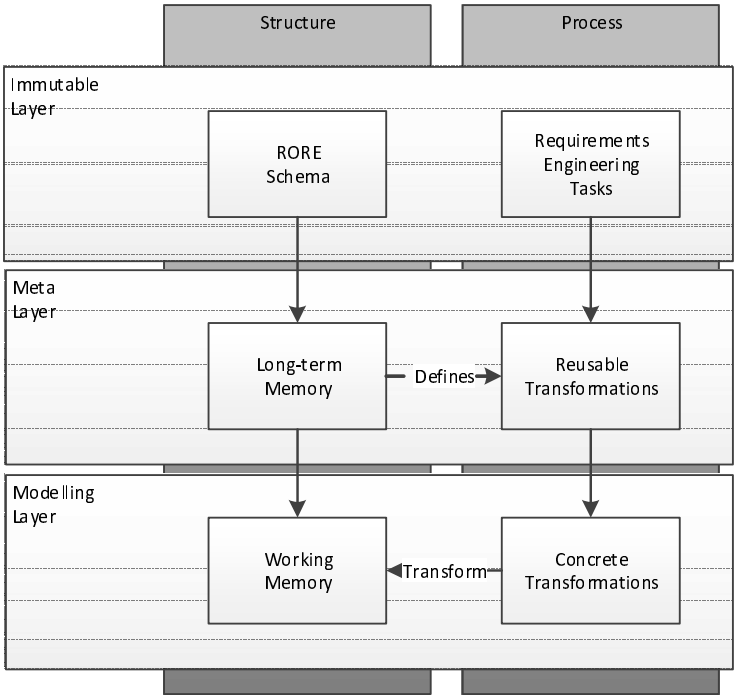

Figure 3.6: Layers of the RORE Framework

The three layers can be summarised as follows. Each layer has both structural and procedural aspects:

- **The Immutable Layer** comprises the fixed, built-in components of RORE: the Requirements Engineering Tasks (see Section 3.6.3) and the schema which defines the structure of RORE's Knowledge View (see Section 3.7). It is in the Immutable Layer that the major components of RORE are defined and formalised. The Immutable Layer is so named because the RORE framework provides no process or mechanism for mutating the components which are defined at this layer.

- **The Meta Layer** comprises *project-independent* components which are intended to be general across multiple RORE episodes, or requirements engineering projects. In particular, this includes the various type systems (Model Types, Model Chunk Types, and Fact Types), requirements engineering method definitions (Phases

and Activities) and the reusable knowledge structures (Analysis Rules, Model Chunks, Production Scripts and Elicitation Stimuli), which actors within RORE can specify as part of RORE's long-term memory. Process at this level is primarily defined through method definitions which stipulate process at a macroscopic level, and reusable knowledge structures which define fine-grained transformations over concrete Models. All components within the Meta Layer are potentially applicable across multiple RORE episodes (or requirements engineering projects).

- **The Modeling Layer** comprises *project-specific* components which are specific to a particular requirements engineering project. Components at this layer provide concrete instantiations of the various Types which are defined in the Meta Layer. The major components at this are the working memory Modeling Context definition, and Models themselves.

3.5.2 Perspectives on the Framework

Sutcliffe distinguishes two types of reuse activity [Sut02]:

- **Design-for-reuse** which is the process of abstracting reusable artefacts, building reuse libraries, and developing tools to support reuse;
- **Design-by-reuse** which is the process of applying those libraries tools in order to support the construction of new software artefacts from reusable components.

Conforming to this distinction, the RORE approach can be viewed from the differing perspectives of two major actors:

- **The Requirements Engineer** is responsible for producing new requirements artefacts. From this perspective RORE is viewed as a framework to support *design-by-reuse*. Within RORE, the requirements engineer reuses knowledge which is stored in a long-term memory knowledge base in order to produce new requirements artefacts;
- **The Knowledge Engineer** is responsible for building and maintaining RORE's long-term memory by specifying requirements modeling notations and libraries of reusable requirements knowledge structures. From this perspective RORE is viewed as a framework to support *design-for-reuse*. The knowledge engineer

treats RORE as a framework which supports the formalisation and organisation within knowledge bases of requirements methods and reusable requirements knowledge.

3.6 The Process View

The Process View describes the behavioural aspects of RORE in detail. RORE has two main perspectives, and corresponding processes, which are required for the approach to function in practice:

- **The Knowledge Engineering (KE) Perspective** involves identifying abstractions and reusable artefacts, and using these to populate a long-term memory knowledge base;
- **The Requirements Engineering (RE) Perspective** involves applying the knowledge structures, which are identified within the KE perspective, in order to build new, and to refine existing, requirements artefacts.

3.6.1 The Relationship Between Requirements and Knowledge Engineering

RORE identifies two distinct processes — a requirements engineering process, and a knowledge engineering process. The distinction between these two processes can be characterised in the following way:

- **Requirements Engineering** is a project-specific role which involves the analysis of a single, individual software application domain in order to understand that domain and produce the requirements for a software system to support some activity within the domain;
- **Knowledge Engineering** is a project-independent role which involves abstracting generalised knowledge structures from multiple instances of software application domains which share in common some salient characteristics so as to populate a knowledgebase with reusable abstractions.

As such, the former focuses on developing the requirements artefacts which describe and characterise an individual sofware application domain, whereas knowledge

engineering involves abstracting generalisations across the software requirements for . The RORE approach assumes that these two roles — Requirements and Knowledge Engineering — would, in practice, be enacted by separate and specialised teams of engineers. Such a separation would be logical because the two roles have different priorities, produce different products, and may well require different skillsets. Requirements engineers, for instance, require communication and interpersonal skills which enable them to interact creatively with stakeholders in order to abstract often-tacit knowledge about an application domain, whereas knowledge engineers need to be able easily to see patterns across multiple requirements artefacts so as to support generalisation from those artefacts.

However, it is important to note that while these two processes may be carried out separately — temporally and spatially — they cannot be enacted independently of one another, and should ideally be enacted in parallel. In RORE, there is a feedback loop between the two processes such that each depends on the product of the other:

- Requirements engineering depends on the availability of a long-term memory knowledgebase — containing the reusable abstractions which are the product of knowledge engineering — in order to drive the refinement;

- Knowledge engineering depends on the availability of existing requirements artefacts — which are the product of the requirements engineering process in RORE — over which domain analysis [PD87, PD90] can be performed in order to identify abstract knowledge structures with which to populate a long-term memory knowledgebase.

Thus the two teams — the requirements engineering team and the knowledge engineering team — will need to work in close collaboration in order for the approach to function as a whole. Knowledge engineers will need to interact with requirements engineers in order to understand the knowledge requirements for requirements engineering across different software application domains, and to validate the abstractions and long-term memory knowledgebases which the KE process produces. Similarly, requirements engineers will need to work closely with the knowledge engineers to in order to ensure that knowledge of individual application domains, for which requirements artefacts have been engineered, is explicit and properly understood by the KE team so as to ensure that it can be accurately captured in reusable abstractions.

This model of interplay between requirements engineering and knowledge engineering is in-line with the principles of domain analysis [PD87, PD90, Sut00] which

advocates the abstraction of reusable knowledge structures from concrete requirements artefacts, and also with the work of Krueger [Kru02a, Kru02b] who has advocated an "extractive" approach to design-for-reuse. Within Krueger's extractive approach, knowledge engineers work in parallel with project teams to extract reusable abstractions from the artefacts associated with individual software projects. Krueger argues [Kru02b] that this approach reduces risk versus a proactive approach (attempting to develop a "complete" knowledgebase up-front) because it scopes the task of building reuse libraries into realistic chunks, and that the approach increases payoff versus a reactive approach (producing reuse libraries, where they don't exist, only when they are needed by a specific project). Accordingly, the RORE approach recommends this tight feedback loop between requirements engineering and knowledge engineering as a way of managing risk while also ensuring the ready availability of a wide range of reusable knowledge structures with utility across a range of software application domains.

3.6.2 Overview of the Process: The KE Perspective

The Knowledge Engineering perspective of RORE focuses particularly on the specification of project-independent knowledge structures in the Meta Layer of RORE. RORE does not define a formal process for, or offer detailed guidance on, knowledge engineering within the framework. In principle, the different knowledge engineering activities within RORE could be undertaken in any order. In practice, however, the relationships between different types of meta-level knowledge structures suggests a general sequence in which these knowledge engineering activities might be undertaken (see Figure 3.7).

These steps can be performed incrementally as the application of RORE by the requirements engineer to a specific project is likely to raise new abstractions of modeling needs which a knowledge engineer should consider in order to refine long-term memory. The basic sequence of steps which is suggested can be summarised as follows:

1. **Meta-modeling** in RORE is the process of specifying a requirements modeling notation in long-term memory. This involves the specification of the Fact and Property Types from which that modeling notation is composed, as well as aggregating those Fact Types in order to define the Model Type itself. In practice, these tasks are likely to be undertaken incrementally or in parallel.

2. **Behavioural Modeling** in RORE involves specifying requirements methods which

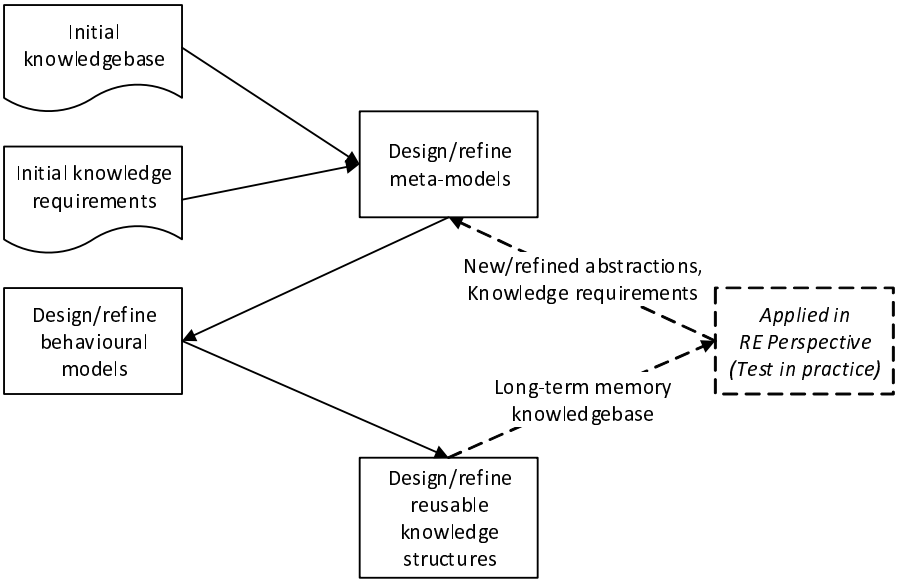

Figure 3.7: Overview of the knowledge engineering process in RORE

a requirements engineer follows which applying RORE to a project. Requirements engineering methods in RORE are specified at two levels of abstraction. Phases describe the macroscopic structure of the engineering process, while activities capture lower-level engineering behaviour. Each activity is defined by a goal which is to produce a model of a particular kind. Behavioural modeling is, then, the process of specifying the structure of Phases and Activities which define a method.

3. **Design-for-Reuse** in RORE is the process of specifying in long-term memory the reusable knowledge structures which define the specific logic through which requirements models are transformed during the requirements engineering process. Four types of reusable knowledge structure are specified in RORE: Analysis Rules, Model Chunks, Production Scripts and Elicitation Stimuli. Each of these knowledge structures is associated with, and supports, one of the requirements engineering tasks which is defined within RORE's behavioural model. These tasks are: Analysis, Chunk-based Inference, Rule-based Inference and Elicitation. Knowledge engineers should abstract reusable knowledge through a process of domain analysis [PD90, Sut02] which are then integrated into long-term memory to support the production, or refinement, of requirements models.

3.6.3 Overview of the Process: The RE Perspective

The architecture of RORE discussed above in Section 3.5 identifies a set of three requirements tasks (Analysis, Inference and Elicitation) and their sub-tasks. These tasks support the role of the *Requirements Engineering* in that they support the process of producing, by reuse, a new requirements artefact. The RORE Requirements Engineering process is designed primarily to support requirements engineering as a process of model transformation. According to this view, requirements engineering involves incremently refining the artefacts which capture the understanding that a requirements engineer currently has of the requirements for a software system. RORE provides a reuse-driven, model transformation-based approach to achieving these refinements. However, this research acknowledges that information relevant to an application domain cannot be derived from nothing, or in a vacuum. As such, this research designed into RORE a set of mechanisms which support the construction of artefacts, even in cases where no existing information about the application domain is available. RORE therefore supports both model transformation and from-scratch generation of models.

The requirements tasks which comprise the RORE approach can be briefly defined in the following ways:

- **Model Analysis (or "Analysis")** refers to a task which involves testing the current state of a target model against a set of rules in order to determine the extent to which the model is both "complete", and of sufficient "quality", where the definitions of both "completeness" and "quality" are encoded in the rules. These rules are retrieved from a knowledgebase as part of the Analysis task. Analysis produces an "Information Requirement" which stipulates if, and in what way, the target model must be further refined.

- **Inference** refers to a task which produces a new set of facts by reasoning exclusively over facts which are already contained in both the source and the target models.

- **Elicitation** refers to a task which involves a requirements engineer utilising their own skill and creativity in interacting with stakeholders so as to produce a set of facts which constitute an appropriate (in the judgement of the requirements engineer) response to some stimulus.

Analysis is a planning activity, whereas Inference and Elicitation are productive activities, meaning that they produce new or refined facts which are integrated into a

target model to support its refinement. A single instance of a productive activity in RORE produces a small *refinement* to a target model. Each such *refinement* is produced through the application of a single application of Clancey's heuristic classification (which is modified for each type of productive requirements task) as follows:

1. **Retrieval:** The current state of the source and target models are queried in order to retrieve a reusable knowledge structure which, when applied to transform the current state of the target model, will satisfy the current information requirement;

2. **Firing:** The reusable knowledge structure which was retrieved is reasoned over in order to produce some (possibly abstract) set of facts based on the source model;

3. **Reification:** If the set of facts that was produced during Firing is abstract, then these facts are reified to fit the needs of the current target model. Reification is achieved by substituting abstract labels within the facts by concrete labels which are selected from the source model in order to produce a concrete set of facts;

4. **Integration:** The concrete facts are appended to the target model and the appropriate relationships to existing facts are defined.

Thus each requirement task *refines* the target model by retrieving and firing a reusable knowledge structure, and then *reifying* and integrating the resulting set of facts.

The requirements tasks in RORE (Analysis, Inference and Elicitation) are organised cyclically, so that each instance of Analysis produces an engineering goal (known as an "Information Requirement") which leads to a further action. The information requirement describes the postcondition for the next round of action in terms of a delta over the current target model.

There are two major kinds of action: Inference and Elicitation. Both are procedures for producing a new set of facts as mandated by the current information requirement. New facts are produced based on the current state of the source models, and predominantly by the reuse of existing knowledge structures. Inference is attempted first, and Elicitation is attempted only if Inference is unable to resolve the *information requirement*.

This thesis provides detailed procedural guidance on the various requirements tasks and control processes that underpin the RORE approach in Chapter 5. However, this

section provides a high-level summary of the process by which requirements models are produced using the RORE method. Figure 3.8 presents an overview.

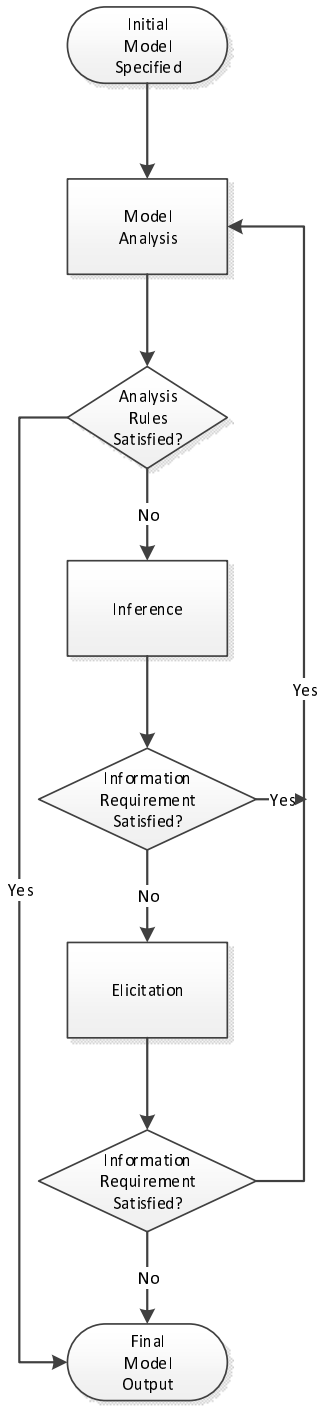

Figure 3.8: Procedural overview of the RORE approach to requirements engineering

The sequencing of tasks as shown in Figure 3.8 is strongly influenced by the second

design heuristic in Section 3.3.4. The basic sequence of events is such that, once Analysis has determined the need for the requirements engineer further to refine the target model, each of the productive requirements tasks is attempted in the sequence shown in Figure 3.8 and one at a time until the information requirement — which is the product of Aanlysis — has been completed. If a particular productive task fails to satisfy the information requirement (e.g., because no appropriate reusable knowledge structure is available to support that task in the current context) then the procedure continues on to attempt the next productive requirements task as a means to satisfy the information requirement. Once the information requirement has been satisfied, no further productive tasks are attempted for the current cycle, and the procedure iterates back to Analysis which re-evaluates the target model to determine the information requirement for the next round of refinement.

Although this sequence of events appears to be designed for the transformation of existing models, three mechanisms within RORE enable the generation of models from scratch:

- Model chunks — which are a type of reusable knowledge structure in RORE and which underpin one mode of Inference in RORE (see Figure 3.5) — contain sets of facts which can be reused and integrated into a target model. Model chunks can, therefore, be used to populate an empty model with an initial set of facts so that further refinement can then proceed in subsequent iterations of the Analysis-Action cycle. Model chunks will typically need to be reified in order to fit a particular concrete model and RORE provides a mechanism by which the requirements engineer can manually specify the fact labels which should be substituted into a model chunk in order to achieve this reification. In this way, model chunks can be used to populate an initial model with an initial set of facts.

- Elicitation stimuli — which describe prompts that request a requirements engineer to input into RORE information in a particular structure — provide a mechanism through which a requirements engineer can manually capture and formalise information from stakeholders. This mechanism enables the requirements engineer to capture an initial set of facts from stakeholders and then to populate the target model with that initial set of facts.

- Index descriptions — which are meta-data descriptions of reusable knowledge structures and which describe the conditions under which a reusable knowledge structure is to be utilised — can stipulate that a reusable knowledge structure is

useful in the event that no information is available within the target model. In this case, a reusable knowledge structure will be matched to an empty model, and so allow that reusable knowledge structure to be applied in order to generate an initial fact set.

Using a combination of these three techniques, a well-designed long-term memory knowledgebase can provide support for from-scratch generation of new models, as well as transformation of existing requirements models. The RORE framework is predicated on the premise that the ideal case for requirements engineering is one in which most of the refinements to a target model are achieved by Inference, rather than by Elicitation. This is because (in line with the second design heuristic described in Section 3.3.4) this thesis hypothesises that Inference is more efficient than Elicitation — that is, that it requires less effort to acquire refined facts by retrieving and reifying a reusable knowledge structure than by acquiring the same facts through a new interogation of stakeholders.

However, in the case of from-scratch model generation this thesis anticipates that in the earliest stages of generation, the acquisition of facts about the application domain will primarily be achieved by the intervention of the requirements engineer as part of the Model Chunk reification process and through Elicitation. This is because something must already be known about the application domain in order to achieve a good quality match between a model and a reusable knowledge structure which will be used to refine that model. A well-designed knowledgebase, therefore, will provide knowledge structures which can be used — through Elicitation and Inference — in order to clarify, in the minimum possible number of cycles, the key facts about the application domain, so that accurate matching to more efficient reusable knoweldge structures can occur. This thesis anticipates, therefore, a shift — once the basic structure for a requirements artefact has been generated — from a greater proportion of Elicitation-driven versus Inference-driven refinements, towards a greater proportion of Inference-driven refinements. Naish and Zhao [NZ11] discuss further this shift in the requirements engineering process, as more information is added to a model, from traditional elicitation methods towards reuse-driven requirements engineering approaches.

3.6.3.1 Summary of the RE Process

The starting point for the refinement of models within RORE is to specify the initial source and target models over which the requirements tasks will operate. One of three overall goals is possible:

1. To produce a new model from scratch;
2. To evolve an existing model;
3. To transform an existing model from one type into another type

The first step in the process, therefore, is to determine which of these initial scenarios must be realised, and to initialise working memory in the appropriate manner: by identifying the appropriate source and target models. The correct set-up for working memory is determined, for a scenario, according to the following rules:

- **For From-Scratch Model Production**, there should be a target model, but no source model;
- **For Model Refinement**, the same model should, initially, be both the source and the target model. Only the target model will be modified;
- **For Model Transformation**, two distinct models should be specified as the source and target models, usually of different model types.

Once RORE has been initialised according to these rules, the first Analysis-Action cycle commences.

Model Analysis is the first step in this cycle. As discussed by Cheng and Atlee [CA07], various kinds of analyses are discussed in the requirements engineering literature, but the unifying theme is the evaluation of a requirements artefact in order to determine what further action must be taken. Similarly, in RORE, the purpose of analysis is to determine whether and what further refinements should be undertaken. Analysis is a rule-driven process. Rules to support analysis are retrieved from long-term memory into working memory, and are then fired over the source and target models to draw conclusions about the current RORE context and the state of the target model.

Two kinds of analysis are distinguished. **Completeness Analysis** is performed, the purpose of which is to evaluate the overall completeness of the target model and so to determine whether or not any further refinement is needed. If not, then the target model is output in its current state as the final model. If so, then **Quality Analysis** is performed in order to evaluate the areas in which the target model requires refinement, as a product of which an information requirement is produced. The information requirement specifies the limitation that has been identified in the state of the current

model and thus defines the nature of the refinement that should be applied during the current Analysis-Action cycle. Once an information requirement has been constructed, the requirements engineer moves to the Action aspect of the cycle in order to resolve this limitation.

To illustrate the two kinds of analysis, consider that a Target Model which is a Problem Model specifying the domains and phenomena for the package router scenario (but not also stipulating the Machine or the Requirements for the scenario). The Target Model may be said to be incomplete because, in accordance with the metamodel given in Figure 3.3, the Problem Model must stipulate both the Machine and the Requirement but, in the current state, does not. In this case, a single rule may be applied to determine both that the Target Model is not currently complete and the direction in which further refinement is needed. Imagine that an Analysis Rule is defined which checks for the existence of a fact of type Machine within the Target Model. This rule is applied first by Completeness Analysis. Because no Machine is defined within the Target Model, the rule fails and so the Target Model is determined not to be complete. Quality Analysis is then conducted. The same rule can also be used for Quality Anaylsis, and again fails. An Information Requirement is therefore generated which comprises the antecedent which is defined for the Analysis Rule (and which the model has failed) and — because the Analysis Rule is stipulated to validate the Machine fact type — a pointer to the Machine fact type itself. This Information Requirement defines the goal of the next round of Action: to produce a non-empty set of facts instantiating the Machine fact type.

Fact Inference Inference produces new facts exclusively by reusing existing knowledge structures. RORE identifies two types of inference **Chunk-based** and **Rule-based inference**. These are distinguished by the type of knowledge structure that is used to inform the inference.

Chunk-based inference matches the source model to a model chunk in long-term memory containing the new set of facts to be integrated into the target model. *Chunk-based* inference is thus closely related to conventional design-by-reuse as in the case of pattern-based design. By contrast, *Rule-based* inference matches the source model to a production script (an ordered set of production rules) in long-term memory, which is used to infer - either semantically or syntactically - the new set of facts from a given set of source facts. Rule-based inference thus reuses procedural, rather than declarative, knowledge and so is akin to a *Calculator*-style reasoning approach, whereas

chunk-based inference is closer to a *Look-up* reasoning style. If either chunk-based or rule-based inference is successful in producing new facts to satisfy the information requirement, these facts are then reified and augmented into the target model. The information requirement now having been satisfied, the process returns to Analysis so that a new information requirement can be produced, and a new Analysis-Action cycle can commence. If, however, inference failed to produce any new facts by either method, the requirements engineer next attempts to satisfy the information requirement by Elicitation.

The two procedures for Fact Inference — Chunk-based and Rule-based Inference — offer different methods for attempting to satisfy the Information Requirement which was produced by Quality Analysis above (i.e. to produce the necessary Machine fact for integration into the Package Router Problem Model). Consider first how this might be achieved by means of Chunk-based Inference. Assume that each of Jackson's Problem Frames [Jac01b] has been specified in long-term memory as a Model Chunk. Chunk-based Inference can be applied to generate a Machine fact for the Package Router Problem Model in the following way. Firstly, the requirements engineer must retrieve from the knowledgebase a Problem Frame chunk which best fits the current state of the Target Model. Problem Frames are distinguished primarily by their structure, but also by the types of the given domains within the frame. The Package Router Problem Model comprises instances of the Required Behaviour, Controlled Behaviour and Information Display frames and so these will be retrieved by the RORE matching process. The requirements engineer may then select one of these retrieved chunks to apply in this cycle. All three Problem Frames comprise a single Machine fact, and so the requirements engineer may choose any of the three Frames to apply. Having made their choice, the requirements engineer must then reify the chosen Problem Frame for the Package Router Scenario. This is done by stipulating manually the name of the Machine and specifying the Phenomena with which the Machine interfaces. The Machine fact is then ready for integration into the Package Router Problem Model.

Now consider a scenario in which Chunk-based Inference were not an appropriate mechanism for establishing the Machine facts which the Information Requirement states are currently missing from the Target Model. This may be the case if, for example, the only chunks that are specified in long-term memory are Problem Frame chunks which are of greater granularity than the Information Requirement demands (including, as they do, Requirement facts as well as Machine facts). In such scenarios

— those to which Chunk-based Inference provides no satisfactory resolution to an Information Requirement — Rule-based Inference can provide a second-line alternative. In the Package Router Scenario, Rule-based Inference can provide an alternative approach satisfying the to generating the Machine facts which need to be integrated into the Target Model. However, a Production Script can only satisfy an Information Requirement if there is sufficient information in the Source Model, or if adequate axioms can be assumed (even if they are not in the Source Model), to infer the facts which are demanded by the Information Requirement. For example, in the case of the Package Router scenario, the source model contains only Given Domains and their Phenomena. No procedure can readily be identified which infers, directly from these facts, the details of the Machine that would be needed to support this context. However, a logical procedure for inferring the Machine fact in such a context might be that only one Machine is needed for each Problem Model and in this case a Production Script might be produced, based on this assumption, which simply creates a single Machine fact and uses information in the Source Model (a simple procedure might be to conjoin the names of the Domains within the Source Model) in order to infer a name for that Machine. Such a Production Script, when applied by Rule-based Inference to the Package Router scenario would generate a single Machine fact whose name might be "PackageRouterMachine".

Fact Elicitation Elicitation is commonly understood as an interactive process for information gathering between the requirements engineer and groups of stakeholder [CA07]. In RORE it is the third type of requirements task identified, and is a procedure for acquiring facts from an "external source" in order to satisfy an information requirement. An "external source" is understood as being any source which is not internal to the RORE framework itself (including the target and source model, or knowledge structures within long-term memory). Elicitation thus serves as a last-resort means of introducing new information into the RORE framework when no information currently within the RORE framework is sufficient to satisfy the information requirement.

The salient knowledge structure in the case of Elicitation is the **Elicitation Stimulus**. This is a knowledge structure which identifies a question and specifies the structure of an appropriate response to that question. Elicitation stimuli thus serve as stock questions which the requirements engineer can use as appropriate in order to acquire the information needed to answer that question. The elicitation stimulus also serves to direct the requirements engineer as regards the specific nature of the facts to be acquired.

As such, RORE provides a means for building a library of stock questions which have been validated within the KE perspective.

During elicitation, an elicitation stimulus is retrieved by matching the current state of working memory against candidate stimuli in long-term memory. Once an appropriate elicitation stimulus has been found to satisfy the current information requirement, the requirements engineer is free to use a range of requirements elicitation techniques (for example, those discussed in [CA07]) to acquire the necessary information. The ideal solution will be that the requirements engineer can answer the question of their own accord, but this will often not be the case, and so the requirements engineer may be required to refer to stakeholders or existing documents other than the source and target models to obtain the desired knowledge. In order to maximise the chance of producing a valid model, it is desirable that the requirements engineer consults with stakeholders where they are not sure of the answer to a question. The requirements engineer then formalises this knowledge into a set of model facts as specified by the elicitation stimulus, and these model facts are then augmented into the target model. If this procedure successfully satisfies the information requirement, then the requirements engineer returns to Analysis to commence a new Analysis-Action cycle. Alternatively, if the information requirement has still not been successfully satisfied, the target model is output from the RORE procedure and a note is made of the unsatisfiable information requirement.

Elicitation also offers a powerful solution to the problem of generating a Machine fact for the Package Router domain which might be called if neither the Chunk-based or Rule-based approaches to Inference is likely to generate a satisfactory solution to the Information Requirement that a Machine fact be generated. In fact, neither Chunk-based or Rule-based Inference (as stipulated in the examples given above) does offer a satisfactory solution to this Information Requirement (although this is an accidental, rather than an essential, limitation arising from the particular knowledge structures that are defined in the example long-term memory knowledgebase). Chunk-based Inference generates facts by reusing Model Chunks which, in the exemplar described thus far, captured Problem Frames and so generated more facts that the Information Requirement demands. Rule-based Inference generates facts by executing Production Scripts, and in the Package Router scenario the only exemplar Production Script which has been defined for generating a Machine relies on simplistic assumptions (such as the existence of only one Machine fact per problem). Elicitation, by contrast, can overcome the limitations which arise as a result of the lack of information from which to

infer the necessary Machine facts by enabling the requirements engineer to request this information from stakeholders. An Elicitation Stimulus to elicit Machine facts might request a requirements engineer to Elicit and define all Machine facts for the Problem Model — and their interactions with the existing Domains. This enables the requirements engineer to step outside of the RORE process and interact with stakeholders (such as business clients and software architects) in order to identify which Machines will be involved in satisfying the Package Router problem. The requirements engineer can then feed this information back into the RORE process as as set of Machine facts, thereby solving the Information Requirement that Machine facts be elicited. This has the advantage that human intellect and creativity can be applied to infer the Machine facts, where formal reasoning processes lacked the necessary information to infer the same facts without relying on simplistic assumptions.

3.6.3.2 Validation of Model Refinements

One important aspect of requirements engineering is the validation of requirements artefacts [NE00, CA07]. Validation of requirements artefacts is required for at least two reasons. Firstly, validation is needed to ensure that the understanding of the requirements for a software application domain which is held by the requirements engineer is consistent with the understanding of the requirements which are held by the stakeholders. Secondly, where stakeholders have differing — and possibly inconsistent — views on the software requirements, validation may be necessary to ensure that the view of the requirements which is documented in the requirements artefacts represents an adequate (as determined by the stakeholders themselves) compromise between the competing requirements views.

In the context of reuse-driven requirements engineering (e.g., as supported by the RORE framework), there is a third reason why validation may be necessary: to ensure that reusable requirements components are applied in their appropriate context, and to confirm that the knowledge which is generated by reuse is consistent with the stakeholders understanding. Validation is necessary in this context because reuse does not, by its nature, acquire information from stakeholders but rather from a knowledgebase. There is a need, therefore, to ensure that the inferences which are derived by reuse provide an accurate representation of the requirements as they are understood by the stakeholders. Unfortunately, any reusable knowledge structure is a generalisation over a domain which may not be consistent with all instances within that domain, and so there is no guarantee — even when a reusable knowledge structure is applied in the

correct domain — that it will accurately characterise a particular instance of that domain. As such, validation by stakeholders is necessary to validate inferences which are deduced by reuse.

RORE, in its current incarnation, does not provide formal or explicit support for validation of the requirements artefacts which it refines and transforms. This is because, in order to limit the scope of this research, this thesis assumes that RORE will be applied only in ideal circumstances, where:

- A good quality long-term memory knowledgebase has been specified which contains sufficient reusable knowledge structures to support requirements engineering across a wide range of application domains;
- Stakeholders are always able to provide, or the requirements engineer is always able to elicit, appropriate information in response to elicitation stimuli.

However, there are various points in the current incarnation of the RORE process at which the requirements engineer has the ability to modify manually the information contained within, or which is to be integrated into, a model. Firstly, the requirements engineer can modify information which is produced by inference as part of the reification process which adapts a model chunk in such a way that it is made concrete to fit the needs of a particular concrete target model. Secondly, the requirements engineer is responsible for formalising — as a RORE knowledge structure — the responses which are provided by stakeholders to elicitation stimuli. The requirements engineer can take this opportunity to validate information which is provided by stakeholders at this point. However, while both of these cases provide informal mechanisms through which the requirements engineer can informally validate the information contained within a model, RORE does not provide procedural guidance on doing so and consequently the requirements engineer must use their own skill and creativity when validating models in this way.

3.7 The Knowledge View

The RORE approach to requirements engineering is knowledge intensive. As discussed in Section 3.6, the facts and models - which RORE acquires and composes - are represented formally by a set of knowledge structures spread across RORE's two knowledge bases. The knowledge model in RORE is specified formally using the Web Ontology

Language (OWL) [BVHH+04]. OWL was chosen as a rich and expressive, yet decidable, description logic for which a wide range of tool support already exists (notably the Protege ontology editor [Uni12] and the Pellet ontology reasoning API for Java [SPG+07]). RORE also uses the Semantic Web Rule Language and Regular Expression substitution patterns to formalise certain aspects of RORE's knowledge model. Formalisation, and the available tool support, of these knowledge structures is an important step towards building tool support for RORE as part of future work, but also has utility in providing clear and unambiguous guidance to requirements engineers.

As shown in Figure 3.5, RORE utilises two distinct knowledge bases, termed **working memory** and **long-term memory** after their respective cognitive science homologies. This section now summarises informally the knowledge structures that are defined in each of these knowledge bases.

3.7.1 Long-Term Memory

Long-term memory holds persistent knowledge that is useful across multiple RORE projects, and is the primary focus of the Knowledge Engineering perspective. Long-term memory is organised into two layers: an *immutable layer* and a *constructed layer* (see Figure 3.5). The *immutable layer* is the layer in which all of the in-built knowledge of RORE is expressed. It is at this layer that each of RORE's knowledge structure types are defined (as illustrated in Figure 3.5). The *constructed layer* is the layer in which instances of these knowledge structures are specified in order to tailor RORE to support a particular requirements engineering method and its associated modeling notations. The primary difference between these two layers is that the knowledge structure types defined in the immutable layer are in-built and assumed by RORE, whereas the knowledge structure instances defined in the constructed layer are specified by the user of the RORE framework to configure RORE to their own needs. During evaluation of the RORE approach, this thesis concluded that applying the RORE knowledge structures in order to formalise a particular requirements engineering method (comprising both a process and a set of model types) enabled the method to be formalised within a practically reasonable period of time — within a week for a team of experienced knowledge engineers working on a significantly complex method.

RORE's long-term memory defines knowledge structures organised into three main packages.

Model Specifications are used to define model types and the fact types from which they are aggregated. Model types define representation languages for modeling some aspect of an application domain or its requirements. Fact types define specific kinds of assertion which can be made about the application domain within a model. Fact types may be *simple* or *complex*. *Simple fact types* are one-dimensional data types such as strings, numbers, boolean and enumerations. *Complex fact types* consist of tuples of properties such that each property is itself defined by a valid fact type defined within RORE. Individual facts instantiate fact types by assigning a value to each property of that fact type.

Model types are defined as unordered aggregations of valid fact types defined within RORE. Models instantiate model types by aggregating facts such that every fact within a RORE model must be an instance of a fact type which both has a valid definition in RORE and is aggregated by the model type associated with that model. Models may contain multiple instances of each fact type which is valid for that type of model. Model types thus define the kinds of facts which can legitimately be expressed in a particular kind of model.

Process Specifications. The requirements tasks and model-specific refinements which this chapter has discussed thus far represent low-level procedural knowledge. To give some context to these refinements, and to support the organisation of this knowledge in RORE's knowledge bases (see Section 5.8), RORE additionally supports the definition of higher-level procedural knowledge. This knowledge is represented in terms of *behavioural units* which are specialised into two types: *phases* and *activities*. *Phases* represent the highest-level form of procedural knowledge within RORE. *Activities* represent a level of procedural abstraction between phases and specific requirements-tasks. *Phases* are composed of activities, activities are composed of requirements tasks. *Each activity* is associated with a particular source and target model, and it is the job of the activity to build a model of the given target type from one of the given source type. Behavioural units are associated with reusable libraries which define model-specific refinements that are relevant to the goals which those behavioural units represent.

As an example, consider Jackson's Problem Frames approach which this thesis has used thus far as a method for addressing the Package Router problem. Jackson's Problem Frames approach, in the first instance, provides a method for producing a Jackson Problem Model. Jackson describes this process in two broad steps. Firstly, the

requirements engineer produces a "Context Diagram" which describes the problem domain in terms of a set of sub-domains and the relationships between these in terms of shared phenomena. Secondly, the requirements engineer elaborates this Context Diagram into a Problem Diagram, which additionally describes the software Requirements that define the problem, and the Machines which will satisfy those requirements. The "problem", according to Jackson, is then to build the specified Machine [Jac01b]. This method can be modeled in RORE using one Phase ("Machine Specification") and two Activities ("Context Modeling" and "Problem Specification") as shown in Figure 3.9.

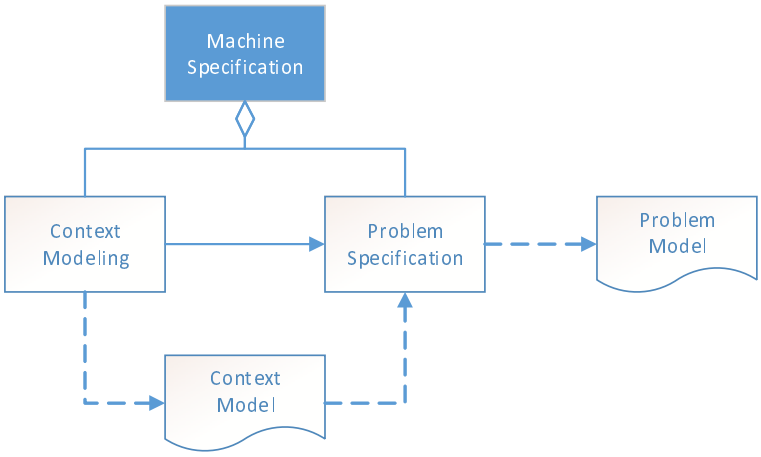

Figure 3.9: Representing part of the Problem Frame Approach in RORE

In RORE, the Machine Specification phase would be defined as having the two activities Context Modeling and Problem Specification. These two activities in turn are defined in terms of the Model Types which they transform — one Source Model Type and one Target Model Type per activity. Thus the Context Modeling activity, which produces a Context Model, would be defined as having no Source Model Type (because it produces a Context Model from scratch) and with the Context Model as the target type.

Reuse Libraries aggregate knowledge structures which can be used to parametrise one of RORE's in-built requirements tasks. This includes those knowledge structures which are used to define model-specific refinements. Reuse libraries comprise four kinds of knowledge structure: analysis rules, model chunks, production scripts and elicitation stimuli. Examples of each kind of reusable knowledge structure were given as part of the commentary on the procedural view of RORE (see Section 3.6.3.1).

Analysis rules are used in RORE to formalise notions of completeness and quality

so as to support planning of requirements tasks. They are applied during analysis to implement checks over model states which determine which kinds of refinements need to be carried out during the current Analysis-Action cycle. Each analysis rule specifies a condition which should hold true if a particular model has reached a desirable level of "completeness" and so form the basis for evaluating models.

Model chunks support chunk-based inference. Model chunks consist of aggregations of facts forming a non-directed acyclic graph. Formally, there is little difference between a model chunk and a model itself. Informally, model chunks are finer-grained and more abstract than models themselves so as to support the composition of models from aggregations of chunks. Model chunks are also stored and indexed in long-term memory to support reuse, whereas models themselves are not packaged for reuse.

Production scripts support rule-based inference. Production scripts are ordered sets of production rules. These production rules consist of an antecedent and a consequent. The antecedent is a SPARQL query [SP07], which selects from the set of input facts the specific facts that are to be transformed by the particular production rules. The consequent specifies a regular expression substitution which is used to transform the facts specified by the consequent to produce new facts. The production rules are performed by the requirements engineer one at a time and in order to incrementally transform the input facts.

Elicitation stimuli provide the knowledge required to support the Elicitation requirements task. Elicitation stimuli consist of two main elements: a question (the stimulus itself) and a response structure. The question is an accessible, natural language, description of the information requirement which the requirements engineer must satisfy. The second component of the elicitation stimulus is a description of the expected structure of the requirements engineer's response to the elicitation stimulus. This is defined by an aggregation of fact types, along with a cardinality indicating how many facts of each type should be specified by the user.

3.7.2 Working Memory

Working memory holds temporary knowledge that is specific to the current RORE project and so is the predominant concern of the Requirements Engineering perspective. It is the knowledge structures in working memory that are manipulated by the requirements tasks of the RORE framework. Two kinds of knowledge are represented in working memory: project knowledge and control knowledge.

Project Knowledge. It is in working memory that both the source and the target models are represented. These model structures represent, in some sense, the application domain about which requirements are to be gathered. The target model should, therefore, evolve in line with the requirements engineer's understanding of this application domain. The source model is that model from which facts are input to a model refinement. The target model is that model to which the refinement is to be applied. It is entirely valid for these two models to be one and the same. Models aggregate facts which instantiate the fact types from which the associated model type is composed.

Control Knowledge. Control knowledge is used to co-ordinate and manage the procedural aspects of RORE. It represents knowledge about the requirements engineering process itself, in particular the goals of the requirements engineering process at various levels of abstraction. The control knowledge package within working memory holds pointers to the phase and activity with which the current refinement is associated. This provides big-picture context for the refinement.

Control knowledge also comprises the information requirement for the current Analysis-Action cycle. Information requirements consist of two main properties. The first of these identifies the type of facts which are to be produced during the the current Analysis-Action cycle cycle. These fact types must be valid for the type of the target model. The second property of an information requirement is a post-condition. Information requirements are generated during analysis when an evaluative test fails. The post-condition records the failed test, and so indicates how the model must change during the current current cycle. Both properties of an information requirement are used to match model-specific refinements to the current context in order to realise the post-condition of the information requirement.

3.8 Summary

This chapter has introduced the Reuse-Oriented Requirements Engineering (RORE) framework. The framework supports the production, and refinement, of software requirements models by systematic reuse. There are three major layers within RORE. In the top (Immutable) layer, RORE comprises a knowledge representation schema, through which knowledge in lower layers can be formalised, and a set of generic

reusable requirements tasks. The two lower layers support the construction of project-independent knowledge bases comprising reusable knowledge and definitions of requirements modeling notations (the Meta Layer), and project-specific requirements models (the Modeling Layer). Cutting across these layers, RORE can be viewed from the perspectives of two actors. From the knowledge engineering perspective, RORE is a framework to support the formalisation of requirements engineering methods and notations, and the definition of reuse libraries to support systematic reuse within those methods. From the requirements engineering viewpoint, RORE is a framework for applying methods and reusable knowledge which have both been defined by a knowledge engineer in order to construct and refine concrete requirements models.

RORE identifies four generic Requirements Engineering Tasks - Analysis, Chunk-based Inference, Rule-based Inference, and Elicitation - through which the refinement of a requirement model is achieved. Firstly the model is analysed by applying a set of Analysis Rules in order to determine its completeness and thus the need for any further transformation. If the model is determined by Analysis to require further refinement, then an Information Requirement is generated and an action cycle is initiated. During this action cycle RORE attempts to resolve the Information Requirement by applying Chunk-based Inference, Rule-based Inference and Elicitation in turn. Each of these requirements engineering tasks is supported by its own type of reusable knowledge which is used by the task in order to capture domain knowledge. The knowledge type which is associated with a particular generic requirements engineering task therefore parameterises the generic task so as to reify it for a specific domain.

Chapter 4

Designing the RORE Approach

4.1 Introduction

This chapter presents the detailed design of RORE, and describes the design of a prototype tool to support both the Knowledge Engineering and the Requirements Engineering perspective. Section 4.2 presents an overview of the technical architecture of the RORE framework and its prototype implement. Each of the layers of the prototype, and the two perspectives of the framework, are elaborated and the key macroscopic components of the prototype are summarised. Section 4.3 drills down into each of these layers describing the individual components of RORE: Section 4.3.1 describes the design of RORE's Long-Term Memory Manager which supports the Knowledge Engineering perspective on RORE, while Section 4.3.2 describes the Requirements Task Assistant which supports the Requirements Engineering perspective.

The Long-Term Memory Manager and Requirements Task Assistant components of the RORE architecture share in common a set of reusable user interface components, and these are described in Section 4.3.3. These two components also utilise the Ontology Management Layer which is a thin persistence wrapper around RORE's knowledge bases. The Ontology Management Layer is described by Section 4.3.4.

4.2 Overview of the RORE Architecture

Figure 4.1 provides an architectural overview of the RORE prototype which this research has implemented and which this chapter summarises.

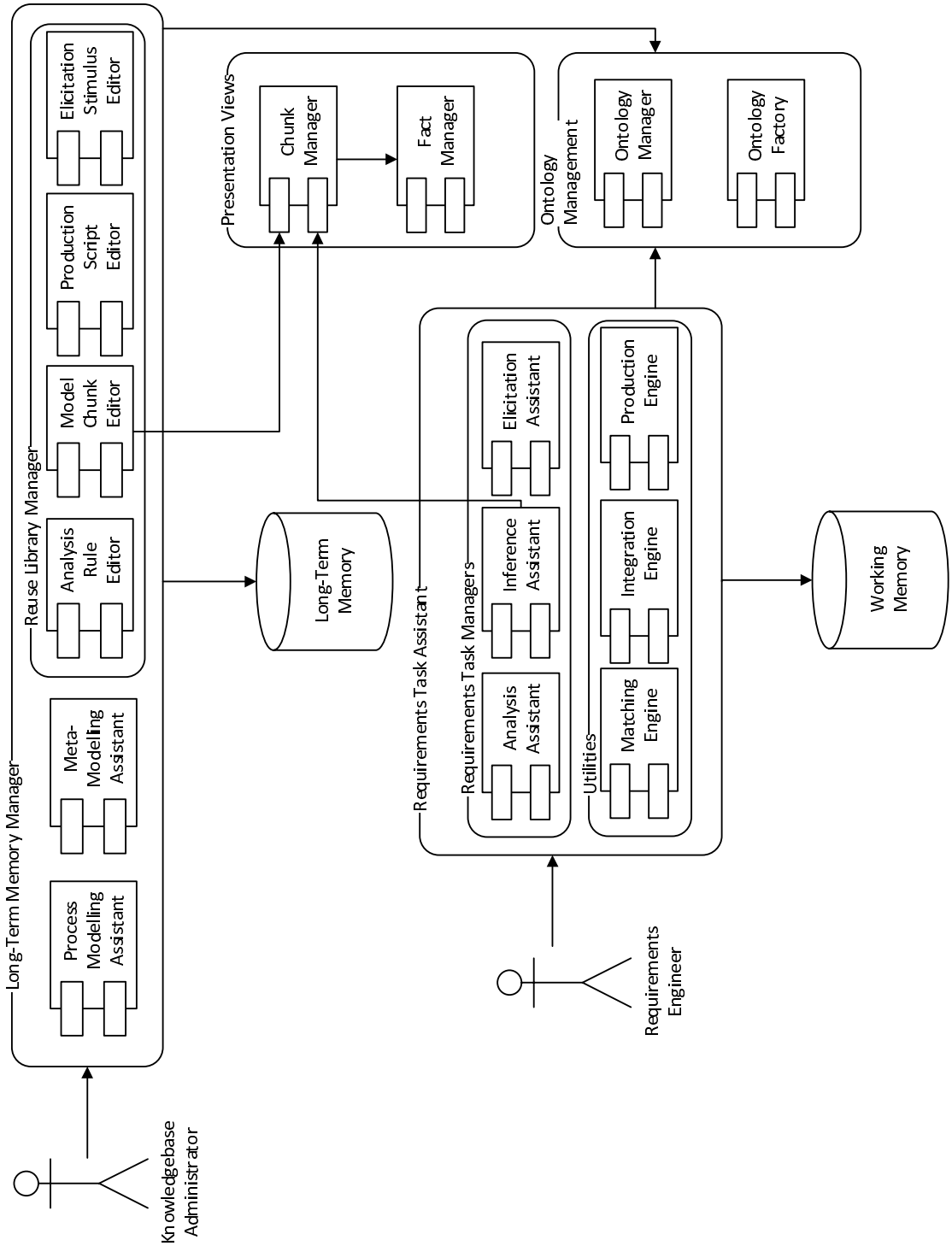

Figure 4.1: Architectural overview of the RORE prototype

4.2.1 Layers of the RORE Architecture

The architecture shown in Figure 4.1 defines four major layers:

- **The Long-Term Memory Manager** is a prototype tool which provides support for the Knowledge Engineering perspective by implementing functionality which facilitates the specification of long-term memory knowledge bases;
- **The Requirements Task Assistant** is a separate prototype tool which provides support for the Requirements Engineering perspective through prototype implementations of the requirements tasks described in Section 3.6.3;
- **The Presentation Layer** is a component library which defines a suite of reusable graphical interfaces that facilitate the specification and manipulation of different kinds of RORE knowledge structure in a RORE knowledge base;
- **The Ontology Management Layer** is a component library which supports the enactment of CRUD (Create, Read, Update and Delete) operations [MI83] over RORE knowledge bases.

The Long-Term Memory Manager is designed to provide support for the Knowledge Engineering perspective of RORE (see Section 3.5.2) by providing a suite of interfaces which enable knowledge engineers rapidly to build long-term memory knowledge bases. The Long-Term Memory Manager (LTMM) provides functionality which facilitates the formalisation of requirements processes, requirements modeling notations, and reusable requirements knowledge. The LTMM provides functionality for adding to, or removing from, long-term memory those knowledge structures which are associated with the long-term memory layer (see Section 3.7.1).

To achieve this, the LTMM depends on both the Presentation Library and the Ontology Management Library. While the LTMM defines user interfaces which are specialised to support long-term memory management (such as interfaces which support procedural modeling), several functions (such as fact editing interfaces) are useful beyond the scope of the LTMM. These shared interfaces are defined within the Presentation Views Library, and the LTMM reuses these interfaces from the Presentation Library.

Similarly, in order to manipulate a long-term memory knowledge base, the LTMM requires access to persistence mechanisms which enable the CRUD operations over a knowledge base file. This functionality is provided by the Ontology Management

Layer, which the LTMM therefore depends on in order to read from and write to knowledge bases.

The Requirements Task Assistant is the second of the tools within the prototype implementation and supports the Requirements Engineering perspective of RORE (see Section 3.5.2). The Requirements Task Assistant provides a prototype implementation of the requirements tasks described in Section 3.6.3: Analysis, Inference and Elicitation. Within the Requirements Task Assistant, Requirements Engineers can define new models, or choose to refine existing models. The Requirements Task Assistant then guides the Requirements Engineer one step a time through Analysis-Action cycles in order to incrementally refine the requirements model towards a more "complete" version. Since each of the requirements tasks is a reuse-driven process, the Requirements Task Assistant provides functionality which supports the automatic matching of modeling contexts to task-relevant reusable knowledge structures. Furthermore, the Requirements Task Assistant also provides a semi-automated procedure for integrating the facts which are produced by an Analysis-Action cycle into the model.

In order to provide this functionality, the Requirements Task Assistant depends on two other layers: the Presentation Layer and the Ontology Management Layer. The Presentation Layer provides fact and model chunk manager components which are shared beyond the scope of the Requirements Task Assistant, but on which the Requirements Task Assistant is dependent. These components are utilised within the Requirements Task Assistant by the requirements engineer when, for instance, responding to elicitation stimuli. The Requirements Task Assistant thus reuses these components from the Presentation Layer.

The Requirements Task Assistant also depends on the persistence functionality which is provided by the Ontology Management Layer. The Requirements Task Assistant accesses two types of knowledge base: a long-term and a working memory knowledge base. Access to the long-term working memory knowledge base is to retrieve reusable knowledge structures, and procedural and metamodel specifications of the requirements engineering method being utilised. Access to the working memory knowledge base is to access the current state of working memory and to write refinements to the target model. The CRUD functionality, as well as advanced querying functionality, on which these activities depend is provided by the Ontology Management Layer.

The Presentation Layer is a layer which defines a suite of user interface views which support the production and manipulation of RORE knowledge structures. There are two major interfaces which are shared across both the LTMM and the Requirements Task Assistant: the Chunk Manager and the Fact Manager. The Presentation Layer decouples these shared interfaces from the particular applications by which they are used, and so makes them reusable beyond the scope of a single application. The only interactions that the Presentation Layer has with other components are its interactions with the LTMM and the Requirements Task Assistant, both of which depend on the Presentation Layer for access to these shared interfaces.

The Ontology Management Layer implements the persistence mechanisms through other layers are able to access RORE knowledge bases. In the RORE prototype, knowledge bases are formalised using the Web Ontology Language (OWL) [BVHH+04] and the Ontology Management Layer defines the functionality which is needed in order to perform CRUD operations over RORE models which are formalised through this notation. The Ontology Management Layer also provides the services that are needed to perform advanced semantic querying [SP07] over RORE knowledge bases. The only interactions in which the Ontology Management Layer is involved, therefore, are its interactions with the LTMM and the Requirements Task Assistant. Both utilise the Ontology Management Layer in order to access Long-Term and Working Memory knowledge bases.

4.2.2 Perspectives on the RORE Architecture

Section 3.5.2 introduced two different perspectives on the RORE framework: the Knowledge Engineer's perspective and the Requirements Engineer's perspective. Figure 4.1 illustrates the interactions of the two main actors in the RORE approach with the major components of RORE.

The Knowledge Engineering Perspective focuses on the task of ensuring that long-term memory is well populated with a rich library of knowledge structures which is designed to support the requirements engineer in performing their own role. The Knowledge Engineer, therefore, interacts with the Long-Term Memory Manager (outlined in Figure 4.1). Different requirements modeling notations may be useful in different scenarios, while different requirements engineers may prefer one modeling approach over another. The knowledge engineer should, therefore, ensure that a collection of

different modeling notations is properly defined in long-term memory. The knowledge engineer is also responsible for performing domain analysis [PD90, Sut02] in order to abstract reusable requirements knowledge structures, and for integrating these into long-term memory. Both tasks are essential to ensuring the adequacy of the library contained within long-term memory. The conclusion which this thesis draws from the evaluation of the prototype RORE tool is that the tool significantly reduces the time which it takes a knowledge engineer to formalise requirements engineering methods over the manual application of the RORE approach. This thesis also concludes that this can be done within a feasible time period by experienced RORE knowledge engineers — within one week for realistic methods.

The Requirements Engineer Perspective emphasises the task of producing new requirements artefacts by reusing knowledge contained within a long-term memory knowledge base. To this end, the Requirements Engineer utilises the Requirements Task Assistant which implements the RORE requirements tasks (see Chapter 5). Each of these tasks is a reuse-driven activity in which reusable structures are retrieved from long-term memory in order to produce new information which can be integrated in order to refine a new or existing artefact. As such, the requirements engineer is engaged in design-by-reuse, in which the reusable artefacts are reusable requirements knowledge structures, and the designed artefact is a new requirements model.

4.3 The Design of the RORE Components

4.3.1 The Long-Term Memory Manager

The RORE Long-Term Memory Manager is a tool which is designed to support the metamodeling and domain analysis tasks of the knowledge engineer (see Section 3.5.2). The tool allows knowledge engineers to build new, and refine existing, long-term memory knowledge bases by specifying three broad kinds of knowledge:

- **Procedural Requirements Engineering Knowledge** which describes the sequence of activities within a requirements engineering method and the type of model which each activity produces from a given type of input model;
- **Metamodels of Requirements Modeling Notations** which define the kinds of facts which a particular modeling notation comprises;

- **Reusable Domain and Procedural Knowledge** which describe chunks of knowledge from which new information can be synthesised in order to refine the state of a concrete requirements model.

Each of these broad kinds of long-term knowledge comprises several finer-grained knowledge types (see Section 3.7.1). The LTMM provides interfaces which allows a user to manipulate knowledge of each of these fine-grained knowledge types within a long-term memory knowledge base.

These interfaces, in effect, are thin wrappers around the Ontology Management Layer. The Ontology Management Layer is described in detail in Section 4.3.4. However, in general the Ontology Management Layer provides one software object for each knowledge structure which is defined in the RORE knowledge view (see Chapter 3 for an overview). Each of these objects in turn provides functionality to CRUD any knowledge structure in long-term memory. The objects also provide services to create, retrieve and remove links between knowledge structures in accordance with the valid relationships between knowledge structure types as defined in the RORE knowledge view.

In general, the LTMM interfaces use these services to retrieve the contents of a particular long-term memory knowledge base, and then the LTMM interface displays the retrieved knowledge structures to the user in an accessible visual form. The user can then manipulate these knowledge structures within the LTMM itself. The LTMM on request from the user then passes the updated knowledge structures back to the Ontology Management Layer which in turn updates the knowledge base file. The LTMM, then, acts simply as a workpiece editor [Jac01b] for creating and editing long-term memory knowledge bases.

The Process Modeling Assistant enables the knowledge engineer to add requirements engineering procedures to a long-term memory knowledge base. Process modeling in RORE provides a structure within which a particular stage of model refinement can be situated so as to support efficient retrieval of knowledge structures during design-by-reuse. The purpose of process modeling in the Knowledge Engineering viewpoint, therefore, is to establish the procedural structure within which specific reusable knowledge structures can be situated.

Figure 4.2 shows the "Process Manager" tab within the LTMM.

In RORE, process models consist of sequences of Phases and Activities which are interlinked by sequence and decomposition links. Sequence links connect behavioural

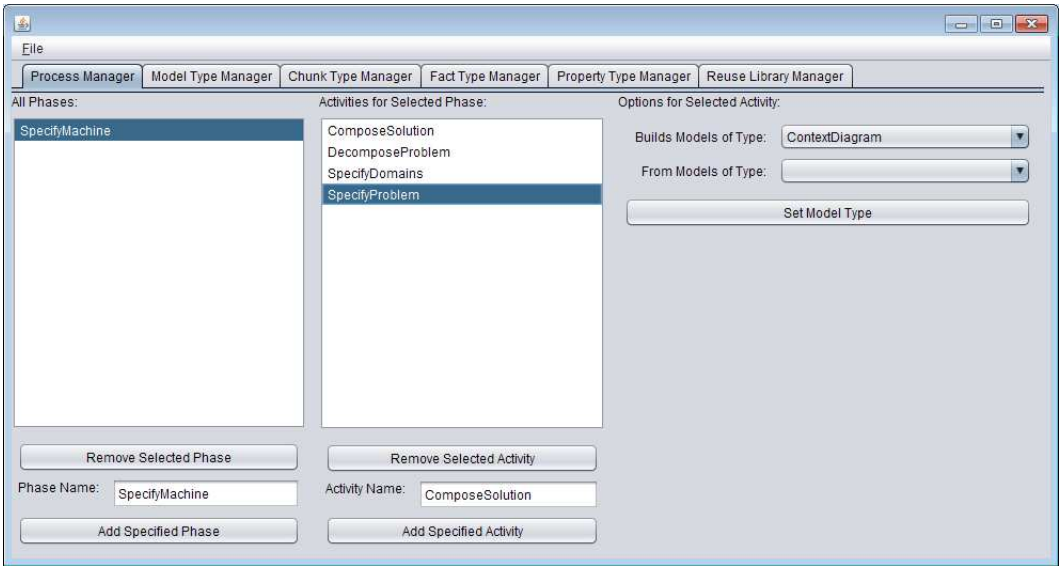

Figure 4.2: The prototype of the Process Modeling Assistant

units of the same type (e.g. Phase-to-Phase) and imply a temporal ordering of process, whereas decomposition links show how a single Phase decomposes into multiple Activities. Activities also stipulate the type of model which they require as input, and the type of model which they produce as output.

The Process Modeling Assistant allows the knowledge engineer to create a new Phase by specifying its name. Similarly, the knowledge engineer can create a new Activity by specifying its name. However, the knowledge engineer must also indicate the Phase to which the Activity will belong, since all Activities must be part of one Phase. The Process Modeling Assistant passes the details of the new Phase or Activity object to the Ontology Management Layer, invoking the "Create Phase" or "Create Activity" service respectively. The Ontology Management Layer will then update long-term memory by creating the new objects as specified.

There is a second step when creating a new Activity: the knowledge engineer must also stipulate the input and output model types over which the new Activity should operate. The knowledge engineer does this by selecting from the model types which are already defined in long-term memory. The Process Modeling Assistant then invokes the "Add Input Model Type" and "Add Output Model Type" messages on the Ontology Management Layer as appropriate.

Finally, to remove a Phase or an Activity the knowledge engineer selects from a list of defined Phases or Activities the object which is to be removed. The Process Modeling Assistant invokes either "Remove Phase" or "Remove Activity" message

on the Ontology Management Layer as appropriate, passing the name of the Phase or Activity to be removed, and the Ontology Management Layer then updates long-term memory accordingly.

Figure 4.2 illustrates the creation of the Machine Specification phase, described in Section 3.7.1. In Figure 4.2 the phase has been named "SpecifyMachine" and has been elaborated further than in the previous chapter: in this chapter, the SpecifyMachine phase comprises four activities describing the process of creating a Jackson Problem Model. The decomposition of the SpecifyMachine phase is given, here, at a finer level of granularity than was given in the exemplar in Section 3.7.1. In Figure 4.2, the SpecifyMachine phase is decomposed into four activities (arranged in alphabetical order): SpecifyDomains, which is the process of producing an initial Context Model by stipulating the given sub-domains within the problem domain; SpecifyProblem, which is the process of elaborating the initial Context Model by stipulating the Machine and Requirement facts for the problem domain; DecomposeProblem, which is the process of producing a Problem Model by decomposing the Problem Model into a set of elementary problem frames (see [Jac01b]); and ComposeSolution, which is the process of defining a solution for each elementary sub-problem in the Problem Model and then recomposing its solution. Each of these activities is defined in terms of the Model Types it transforms and produces: SpecifyDomains produces a Context Diagram from scratch; SpecifyProblem elaborates one context diagram into another; DecomposeProblem produces a Problem Model from a Context Diagram; ComposeSolution produces an unspecified model type (because the process is not yet well-understood in Jackson's Problem Frames approach) from a Problem Diagram. The SpecifyMachine phase is hilighted, and so each of these activities is displayed in the central column. The SpecifyProblem activity is also hilighted and the Source and Target Model Types are displayed for the SpecifyProblem activity (although in the Figure, these have not been specified): the right hand column shows that no Source Model Type has yet been given for the activity, but that the Target Model Type has been selected as the ContextDiagram.

The Metamodeling Assistant in RORE allows knowledge engineers to adapt the RORE approach to support the production of a range of different requirements modeling notations. This is one way in which RORE aims to improve on the generality of previous reuse frameworks (e.g. the Domain Theory and POSE). This thesis concludes that this approach enabled RORE to achieve a high level of generality which is greater

than that of the Domain Theory, and close to (if not quite in line with) that of POSE. The Metamodeling Assistant provides four sub-components which support the process of defining new model types:

- **The Model Type Manager** supports the aggregation of new model types from collections of fact types;
- **The Chunk Type Manager** similarly supports the aggregation of new model chunk types from collections of fact types;
- **The Fact Type Manager**, which is further divided into the simple and complex fact type managers, supports the definition of primitive fact types and the aggregation of those primitives into arbitrarily complex fact types;
- **The Property Manager** allows the knowledge engineer to specify the properties through which complex fact types aggregate primitive fact types.

An overview of these knowledge structures (Model and Chunk Types, Fact Types and Property Types) is given in Section 3.7.1.

Figure 4.3 illustrates the model-type manager.

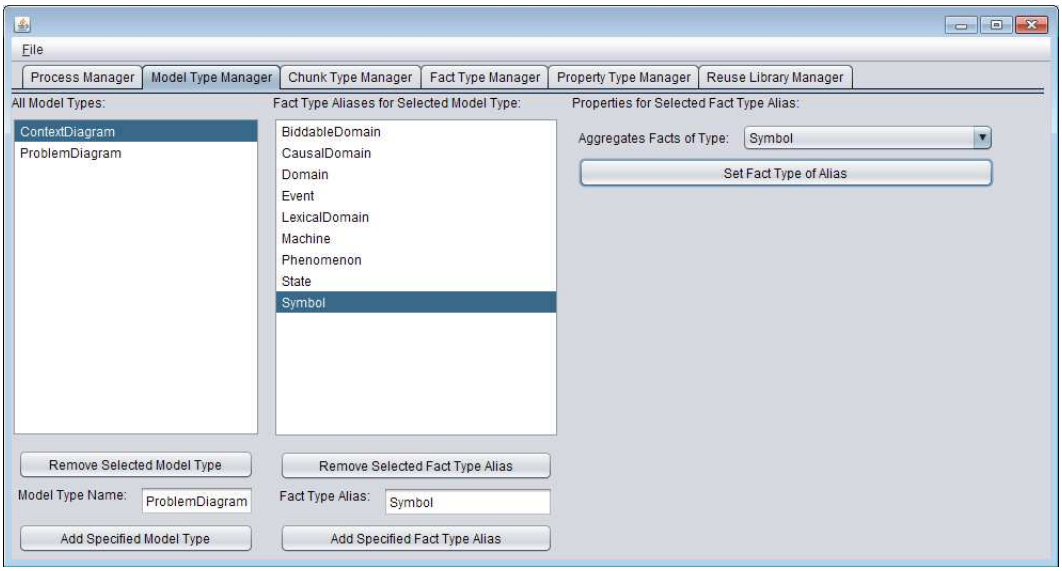

Figure 4.3: The Model Type Manager tab allows users to specify Model Types and the Fact Types which they aggregate

The Metamodeling Assistant allows the knowledge engineer to define new Model Types by aggregating fact types. Fact Types are aggregated through knowledge structures known as “Fact Type Aggregations” which provide a layer of indirection between

Model and Fact Types. To create a new Model Type the knowledge engineer specifies the name of that Model Type. The "Create Model Type" message is then invoked on the Ontology Management Layer which updates long-term memory with the new Model Type.

Fact Type Aggregations can then be added to the new Model Type by specifying an alias for the aggregation. Fact Type Aggregations have an alias which corresponds to the name that is used for a particular type of fact within the context of a particular modeling language (e.g. object-like structures may be called "Objects" in one language and "Entities" in another). The "Create Fact Type Aggregation" message is invoked on the Ontology Management Layer to add the aggregation to long-term memory, and to link it to the selected Model Type. Finally, the knowledge engineer should select the Fact Type which the Fact Type Aggregation will link to the Model Type. This is done by choosing an existing Fact Type from a list. The Modeling Assistant then passes the name of the chosen Fact Type Aggregation and Fact Type to the Ontology Management Layer which links the two knowledge structures in long-term memory. This completes the link from the Model Type to the Fact Type.

Model Chunk Types are created through a very similar process. The only significant difference is that Model Chunk Types specify a cardinality on each aggregated Fact Type. This provides a mechanism for constraining the granularity of Model Chunks.

Figure 4.3 illustrates the creation of the ContextDiagram and ProblemDiagram Model Types in RORE using the Long-Term Memory Manager. Both Model Types are composed of sets of Fact Types through Fact Type Aggregations. The Fact Type Aggregations allow equivalent Fact Types to be given different names in different Model Types (although the name of a Fact Type Aggregation may be the same as the name of the Fact Type which it aggregates). In the illustration above, the ContextDiagram is hilighted and so the Figure also shows the Fact Types which it aggregates: all of the Fact Types which are contained within a Problem Diagram as stipulated in Section 3.4, save for the Requirement Fact Type which is a part of the problem and not of the context. A specific Context Diagram may, therefore, comprise facts of any of these types and these facts may be related through the properties which are defined for each fact type.

In order to create new Model and Chunk Types, therefore, a set of Fact Types must already have been defined in long-term memory. In the RORE prototype this is done through the "Fact Type Manager". This tab is decomposed into two sub-tabs

to represent the different structures of Simple and Complex Fact Types. Figure 4.4 illustrates the Complex Fact Type Manager tab.

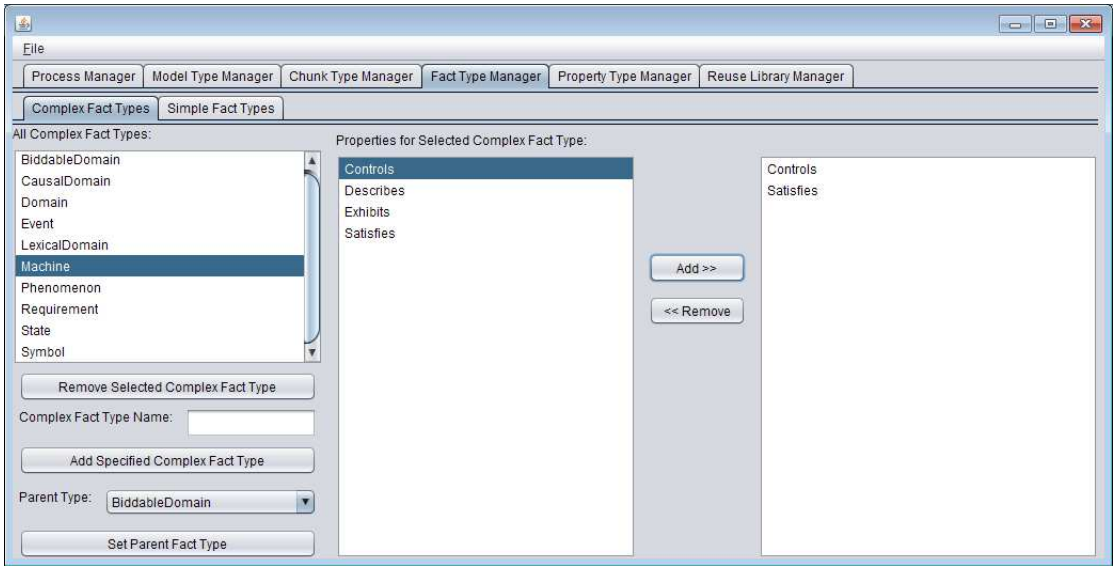

Figure 4.4: The prototype Complex Fact Type Manager tab

The process for specifying both Complex and Simple Fact Types is somewhat similar. Fact Types in either case are created by specifying the name of the new Fact Type. This is then passed, in the usual way, to the Ontology Management Layer which creates the Fact Type in long-term memory. The different lies in how the two classes of fact type are defined. Once a Simple Fact Type has been created in long-term memory, the knowledge engineer stipulates the primitive data type which the Fact Type will have as its value type. Again, the Ontology Management Layer is invoked to create this link in long-term memory.

By contrast, Complex Fact Types are defined in terms of aggregations of Property Types. In the Complex Fact Type Management tab the knowledge engineer can associate Property Types with Complex Fact Types by choosing one of each. A call is then made to the Ontology Management Layer which updates long-term memory to create this link. Similarly, invoking the "Remove Property Type from Complex Fact Type" service on the Ontology Management layer, and passing a linked Complex Fact Type and Property Type will update long-term memory to unlink the two knowledge structures.

Property types can be added using the property type manager. Once a property type has been added to long-term memory, it can be associated with or disassociated from a complex fact type.

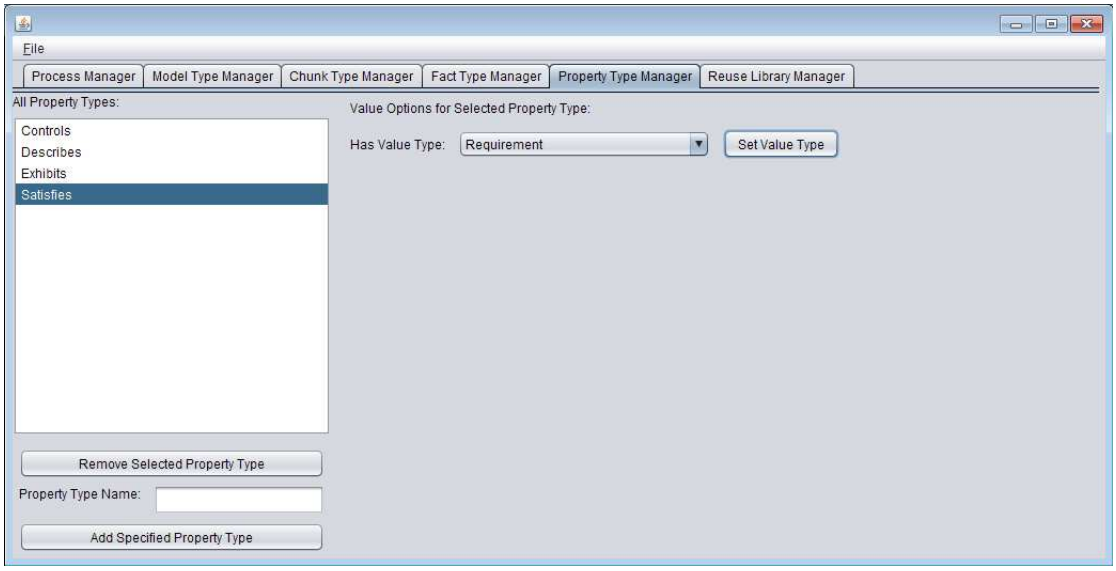

Figure 4.5: The Property Type Manager tab allows users to specify property types, indicating the type of value they reference

Figure 4.5 shows the property type manager. Property types are specified as follows. New property types are created by specifying the name of the property type which is passed to the Ontology Management Layer for addition into long-term memory. Property Types have a value type, which is the Fact Type which the Property Type has as its range. The knowledge engineer selects, for a given Property Type, the value type, and the Metamodeling Assistant passes the details of this link to the Ontology Management Layer which writes it to the knowledge base.

Figures 4.4 and 4.5 show an example of the use of the Long-Term Memory Manager to specify Complex Facts and the Properties through which those Complex Facts are related to one another. Four Property Types are defined in the Property Type Manager shown in Figure 4.5. These are: “Controls”, the value of which is of type Phenomenon; “Describes”, which has the value type Domain; “Exhibits”, which has the value type Phenomenon; and “Satisfies”, which is selected in the Figure and has the value type Requirement. Figure 4.4 shows the Complex Fact Type Manager with the Machine Fact Type highlighted. The central column of the Complex Fact Type Manager shows the four Property Types which have been defined in long-term memory (Controls, Describes, Exhibits and Satisfies). The right-hand column shows those Property Types which have been assigned to the Machine Fact Type: Controls, indicating that a Machine is linked to a set of Phenomenon facts through the Controls

relationship; and Satisfies, which links a Machine to a Requirement. Using the Complex Fact Type Manager and the Property Type Manager, then, the Machine Fact Type (as it is specified in Figure) 3.3 has been specified.

Model and Chunk Types, Fact Types and Property Types can each be removed from long-term memory by passing the name of the knowledge structure which is to be removed to the Ontology Management Layer and invoking the correct "Remove" service for that knowledge structure type.

4.3.1.1 The Reuse Library Manager

The Reuse Library Manager in RORE is designed to support knowledge engineers in formalising and specifying reusable knowledge structures within long-term memory. As is the case for the other aspects of the LTMM, specialised user interfaces are provided to support the definition of each of four types of reusable knowledge structure:

- **The Analysis Rule Editor** supports the definition of Analysis Rules which are knowledge structures that define the conditions against which the quality and completeness of requirements Models are assessed;
- **The Model Chunk Editor** supports the definition of Model Chunks which are declarative knowledge structures comprising sets of Facts which can be integrated into a requirements Model either as-is, or after some reification;
- **The Production Script Editor** aids the definition of Production Scripts which comprise sequences of transformations over requirements Models and can be executed automatically during requirements engineering;
- **The Elicitation Stimulus Editor** facilitates the definition of Elicitation Stimuli through which requirements engineers can be prompted to elicit information manually in order to refine requirements models.

The Reuse Library Manager, like the LTMM in general, provides a thin wrapper around the Ontology Management Layer in order to support the construction of a reuse library as part of RORE's long-term memory.

The Analysis Rule Editor supports the knowledge engineer in the task of defining Analysis Rules (see Section 3.7.1) to support the Analysis task (see Section 3.6.3). Analysis Rules are used to reason over requirements models in order to determine the

quality and completeness of those models. As such, Analysis Rules are central to determining the goals that should be acquired in each cycle of the Requirements Engineering process. The task from the Knowledge Engineering perspective, therefore, is to utilise the LTMM to formalise Analysis Rules into LTMM.

The Analysis Rule tab is shown in Figure 4.6.

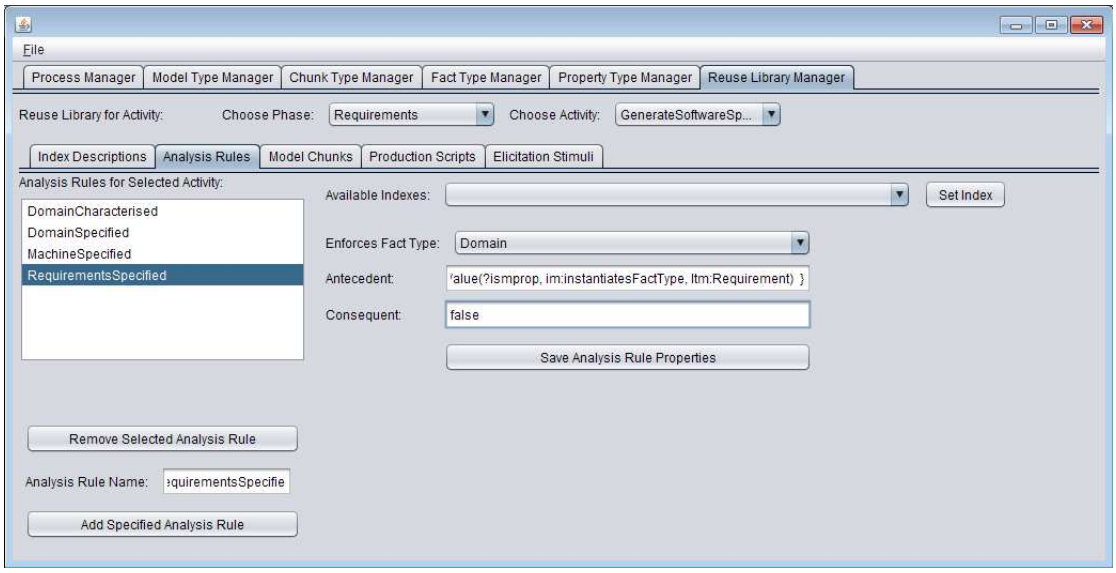

Figure 4.6: The Analysis Rule Editor tab

Analysis Rules are added to long-term memory in the usual way: the knowledge engineer specifies the name of the rule, and the Analysis Rule Editor passes this onto the Ontology Management Layer which in turn writes it to long-term memory. Analysis Rules also have three other major properties: an Enforced Fact Type, an Antecedent and a Consequent. The Enforced Fact Type indicates the kind of information which the Analysis Rule checks for, and its value is chosen from a list of fact types which are defined in long-term memory. The Antecedent of the Analysis Rule stipulates the main condition of the rule, and the Consequent specifies the Boolean value which should be returned in the event that the rule holds true over a requirements model. Both the Antecedent and Consequent are input manually by the knowledge engineer. Once the knowledge engineer has specified values for each of these attributes of the new Analysis Rule, they are passed down to the Ontology Management Layer which updates the Analysis Rule record in long-term memory.

Figure 4.6 shows the Analysis Rule Editor populated with a number of sample Analysis Rules for assessing the quality of Jackson Problem Models. The rules determine whether or not facts have been specified for each of the four main fact types

which is defined by the Jackson Problem Model: Domain, Phenomenon, Machine and Requirement. The selected rule (RequirementSpecified) verifies that at least one Requirement Fact has been specified and determines the completeness, or otherwise, of a Problem Model accordingly. In Figure 4.6, the list on the left-hand side of the interface displays all of the defined Analysis Rules with the RequirementSpecified rule hilighted. The right-hand column of the interface provides components for defining the parameters of an Analysis Rule. The RequirementSpecified rule, having been selected, is shown here. It is not currently classified by an Index Description (which defines meta-data, used during retrieval), but has both an Antecedent and a Consequent defined. The Antecedent in this case (which is partially shown) simply checks that no Requirement Fact has been specified. The Consequent has been set to false, indicating that if the Antecedent condition is true, the rule will be considered to have failed. This illustrates the manner by which a user can manipulate Analysis Rules.

The Model Chunk Editor enables the knowledge engineer rapidly to formalise Model Chunks in long-term memory. Model Chunks represent units of declarative knowledge which can be either integrated as-is into a target model, or else reified and then integrated. As such, Model Chunks offer a means of rapidly refining a requirements model by reuse.

The Model Chunk Editor is shown in Figure 4.7.

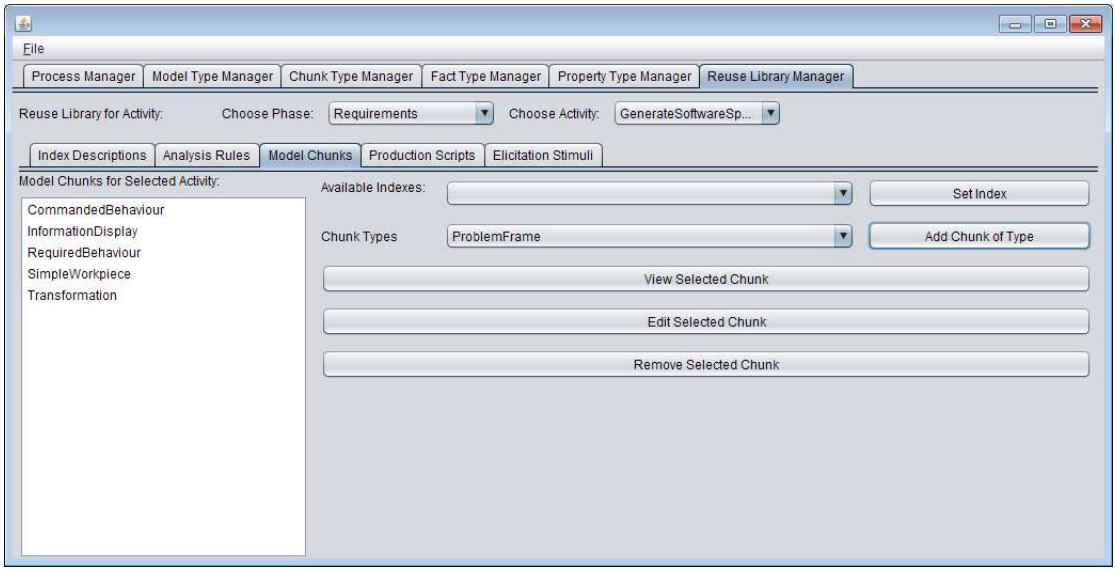

Figure 4.7: The Model Chunk tab

The Model Chunk Editor relies predominantly on the Chunk Manager component

in order to manipulate chunks. Accordingly, when the knowledge engineer indicates that they wish to create a new Model Chunk in long-term memory, the Chunk Manager is displayed. The output of the Chunk Manager will be a new (or reified) Model Chunk, which the Model Chunk Editor passes to Ontology Management Layer. The Ontology Management Layer accordingly updates long-term memory with the new (or reified) Model Chunk.

Figure 4.7 illustrates the Model Chunk Editor populated with five Model Chunks in the left-hand list which correspond to Jackson's five elementary problem frames: Required Behaviour; Commanded Behaviour; Information Display; Simple Workpiece; and Transformation. Each of these model chunks captures the domain structure and the requirement for the element problem frame, and so, accordingly, they are defined to instantiate the Problem Frame chunk type. From the Model Chunk Editor, the user can opt to view the Facts for the currently selected Model Chunk, Edit the Chunk or Remove the Chunk.

The Production Script Editor facilitates the process of defining new Production Scripts (see Section 3.7.1) in long-term memory. Production Scripts provide a procedural alternative to the declarative reuse offered by Model Chunks. Whereas Model Chunks represent the product of some prior reasoning process, Production Scripts represent the reasoning process itself: they describe fine-grained procedures which encapsulate transformations over requirements models. The Knowledge Engineering perspective is concerned with creating and specifying Production Scripts.

The Production Script Editor is displayed in Figure 4.8.

Production Scripts are complex knowledge structures which comprise several significant features. Production Scripts are defined initially by specifying a name for the Production Script, and an input query. Once these have been specified, the Production Script Editor invokes the Ontology Management Library in order to write the new Production Script to long-term memory.

This done, the knowledge engineer can now define the body of the Production Script. Each Production Script consists of multiple Production Rules, sequenced in a particular order. Each Production Rule comprises a name, an antecedent and a consequent. Within the Production Script Editor, the knowledge engineer can add a new Production Rule to the current Production Script by specifying each of these attributes. The Production Script Editor passes to the Ontology Management Layer the name, antecedent and consequent, as well as the name of the Production Script to which the rule

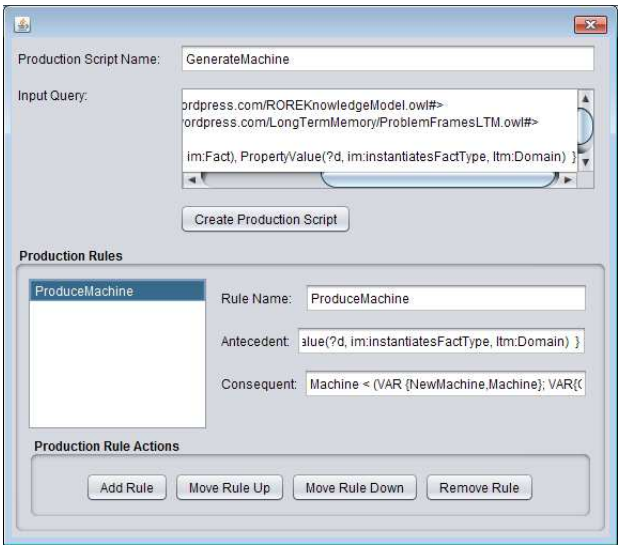

Figure 4.8: The Production Script Editor

should be linked. The Ontology Management adds the Production Rule to long-term memory and creates the link between the Production Rule and the Production Script of which it is a part.

The knowledge engineer can also remove both Production Rules and Production Scripts from long-term memory using the LTMM. The knowledge engineer does this by choosing the Production Rule or Production Script to remove, and the LTMM then passes the name of the to-be-removed knowledge structure to the Ontology Management Layer. The Ontology Management Layer updates long-term memory accordingly.

Figure 4.8 illustrates the Production Script Editor as populated by a single Production Script: the GenerateMachine script, which was described in Section 3.6.3, as one possible way for producing a Machine Fact to satisfy the Information Requirement that all Problem Models have such a Fact. The name of the script is stipulated in the textbox at the top of the editor. Immediately below this is a textbox to specify an Input Query, which is the set of facts — selected from the Source Model — which the Production Script will transform in order to produce the new facts to satisfy an Information Requirement. In the exemplar, shown in the illustration, an Input Query is specified to retrieve from the Source Model the Domains which the Machine will operate over. A single Production Rule ("ProduceMachine") is defined for the GenerateMachine script and is listed in the list on the left of the Production Script Editor. The name of the rule is stipulated in a textbox to the right of this list. The Antecedent of the rule is

given immediately below the Rule Name textbox. The Antecedent of a Production Rule retrieves, from the results of the Input Query, the specific set of facts which the individual rule will transform. Because there is only one rule in the GenerateMachine script, all facts which were retrieved by the Input Query are to be transformed by that one rule, and so the Antecedent of the ProduceMachine rule is the same as the Input Query of the script as a whole. This is a condition which is particular to the exemplar script and is not necessarily true of Production Scripts, or Rules, in general. Finally, the Consequent of the ProduceMachine rule is defined in the textbox below the Antecedent. In this case, it stipulates that the output of the ProductionRule is to be a single fact, of type Machine, which is produced by simply generating a new Machine Fact and associating it — via the "Controls" property — with the Phenomena for all Domains which were retrieved by the Antecedent of the rule.

The Elicitation Stimulus Editor allows the knowledge engineer to define new Elicitation Stimuli. Elicitation Stimuli are reusable knowledge structures representing questions or prompts (stimuli) which can be put to a requirements engineer to prompt them manually to provide new information to be integrated into a model. The Elicitation Stimulus thus complements the declarative and procedural reuse which is offered by Model Chunks and Production Scripts respectively. The Knowledge Engineering perspective is concerned with defining new Elicitation Stimuli in long-term memory.

Figure 4.9 shows the Elicitation Stimulus editor.

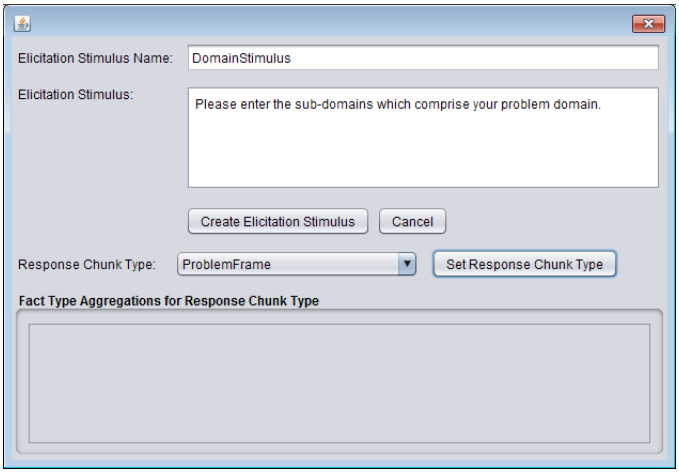

Figure 4.9: The Elicitation Stimulus Editor

This allows the knowledge engineer to create a new Elicitation Stimulus by specifying the name of the stimulus, and the stimulus question itself. The knowledge engineer

must also set the response structure for the Elicitation Stimulus. To do this, the knowledge engineer chooses from a list the Model Chunk Type which best characterises the structure of the response which should be supplied by requirements engineers in relation to the stimulus. For convenience, the Elicitation Stimulus Editor displays the structure of each Model Chunk Type to the knowledge engineer as they select a response type. Once the knowledge engineer has made their choice the name of the stimulus, the stimulus question and the response type are passed to the Ontology Management Layer which writes the new Elicitation Stimulus to long-term memory.

Knowledge engineers can also remove Elicitation Stimuli from long-term memory by selecting the name of the Elicitation Stimulus within the LTMM. The LTMM then invokes the "Remove Elicitation Stimulus" service of the Ontology Management layer, passing the name of the selected Elicitation Stimulus, and the Ontology Management Layer updates long-term memory accordingly.

Figure 4.9 shows the Elicitation Stimulus Editor populated with a single Elicitation Stimulus ("DomainStimulus"). This stimulus is described, in Section 3.6.3 as one possible way to generate the Given Domains which comprise the initial Context Model in the Package Router example. The DomainStimulus is an example of a Chunk-Based Stimulus: a type of Elicitation Stimulus which elicits a Model Chunk as a response. The name of the stimulus is given in the textbox at the top of the Editor. Immediately below this is a second textbox for stipulating the Elicitation Stimulus itself. This stimulus is a human-readable question which a requirements engineer can either present directly to stakeholders, if the requirements engineer believes the stakeholder will comprehend the question directly, or which the requirements engineer can use as the basis for a more creative approach to eliciting from stakeholders the information which the stimulus requests. Finally, the Editor offers the knowledge engineer the opportunity to determine which type of Model Chunk the Elicitation Stimulus should request as a response, and this determines the kinds of Facts which the requirements engineer will need to elicit from the stakeholder in order to satisfy the stimulus.

4.3.2 The Requirements Task Assistant

The Requirements Task Assistant is designed specifically to provide direct tool support for each of the Requirements Subtasks describes in Chapter 5, and the various lower-level procedures on which the Requirements Tasks depend. As such, the Requirements Task Assistant directly supports the Requirements Engineering perspective on RORE. The Assistant operates over a working memory knowledge base which comprises, in

addition to contextual information (such as details of the current Requirements Engineering Task), a source and a target model. The source model is the input model, and the target model is the model which is to be refined.

Each RORE Requirements Engineering Task is supported by a tab within the Requirements Task Assistant. To use the Requirements Task Assistant, the requirements engineer should work their way through each of the tabs in the order shown. Tabs are laid out in the order in which they should be applied in accordance with the RORE requirements engineering process (see Section 3.6.3). In general, these tabs provide support for the RORE reuse tasks, and each tab is customised to the particular requirements task which that tab supports. The prototype thus guides the requirements engineer through the process in an intuitive fashion.

The Requirements Task Assistant offers three major features: support for each of the Requirements Engineering tasks defined by RORE (see Section 3.6.3).

- **The Analysis Assistant** supports the utilisation of Analysis Rules to enact both Completeness and Quality Analysis;
- **The Inference Assistant** supports the utilisation of Model Chunks to enact Chunk-based Inference, and Production Scripts to enact Rule-based Inference;
- **The Elicitation Assistant** supports the utilisation of Elicitation Stimuli to enact Elicitation.

Section 3.3.3 described the influence of Clancey's heuristic classification [Cla85] on the design of each of the RORE Requirements Engineering procedures. Each of these procedures consist of two major steps (Matching and Firing), while Inference and Elicitation also comprise a third step (Integration). The Requirements Task Assistant therefore provides three additional components to support these lower-level operations:

- **The Matching Engine** supports the retrieval of context-relevant knowledge structures from long-term memory to enable enactment of the current Requirements Engineering Task;
- **The Production Engine** is responsible for the automatic execution of Production Scripts in order to produce a set of output Facts;
- **The Integration Engine** supports the integration into the target model of the Facts which are produced on each Analysis-Action cycle.

In general each of the major Requirements Engineering functions which are provided by the Requirements Task Assistant is implemented by composing these lower-level functions, in addition to the services provided by the Presentation and Ontology Management Layers, into the three (or two, in the case of Analysis) step process:

1. **Matching** of an appropriate reusable knowledge structure to support the enactment of the current Requirements Engineering Task (e.g. Analysis Rules to support Analysis) in the current modeling context. This is performed by the Matching Engine;

2. **Firing** of the retrieved reusable knowledge structure using an appropriate lower-level service (e.g. by using the Production Engine to support the firing of Production Scripts);

3. **Integration** (for Inference and Elicitation, the "productive" tasks) of the resultant Facts into the target model through the use of the services provided by the Integration Engine.

In order to illustrate the application of the Requirements Task Assistant to a concrete example, imagine an initial version of the Problem Model — presented in Section 3.4 — in which the Machine and Requirements Facts have not yet been specified. The following sections describe the different features of the Requirements Task Assistant and illustrate how they might be applied to refining this Problem Model by producing and appending appropriate Machine, and Requirements, facts.

The Analysis Assistant supports the requirements engineer in enacting both Completeness and Quality Analysis (see Section 3.6.3): the first steps in the Analysis-Action cycle. Completeness Analysis performs an initial check to determine whether or not the target model requires further refinement, while Quality Analysis is intended to check further in order to determine in what way the target model needs to be refined.

One tab is provided each for Completeness Analysis and Quality Analysis, although the structure of both tabs is broadly similar. Figure 4.10 shows the Completeness Analysis tab.

To perform Analysis of either kind the requirements engineer first opts to retrieve from long-term memory an appropriate set of Analysis Rules. To do this, the Analysis Assistant passes the modeling context (the source model, target model and the current Analysis Task Type) to the Matching Engine, which will return a set of Analysis Rules

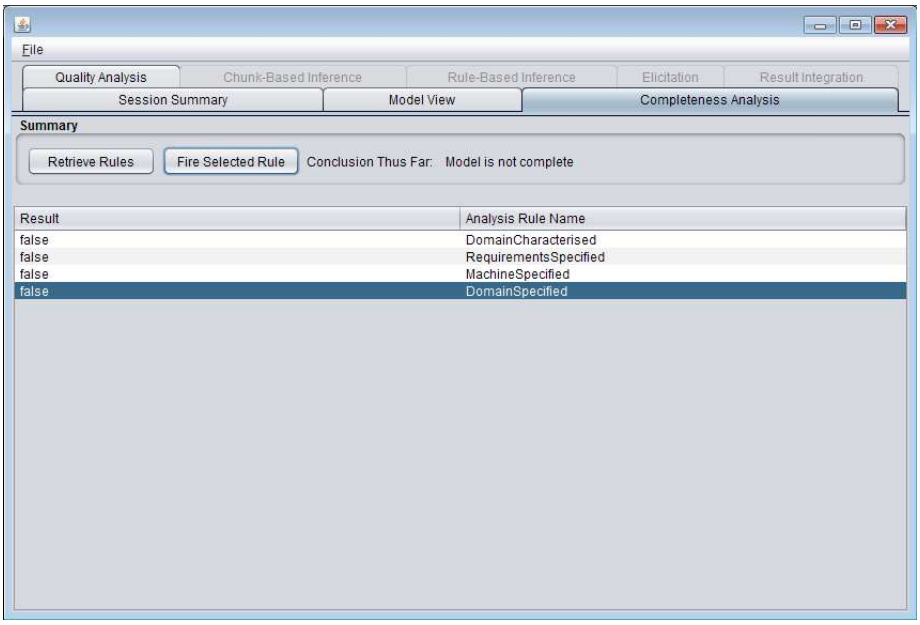

Figure 4.10: The Requirements Task Assistant's Completeness Analysis tab

which are appropriate to that context. These will be displayed to the requirements engineer in the appropriate tab.

The requirements engineer should then use the Analysis Assistant to fire each Analysis Rule in turn. It is at this point that the two types of Analysis are distinguished. In the case of Completeness Analysis, the requirements engineer must fire all of the retrieved Analysis Rules. If an Analysis Rule was retrieved which the target model does not satisfy then the target model is determined to be incomplete and the requirements engineer should progress to Quality Analysis. The target model must, therefore, satisfy all of the retrieved Analysis Rules in order to be considered complete. The Completeness Analysis tab indicates whether or not the target model has passed all of the Analysis Rules that were fired thus far (the Quality Analysis tab does not).

During Quality Analysis the requirements engineer again performs Matching. The requirements engineer then fires each retrieved Analysis Rule in turn until an Analysis Rule fails. When this occurs, the Analysis Rule tab generates an "Information Requirement" which indicates the condition which the target model fails to satisfy and, therefore, the goal of the next Action cycle.

The Analysis Assistant automatically evaluates Analysis Rules at the request of the requirements engineer. To do this, the Analysis Assistant uses the services which are provided by the Ontology Management Layer. Analysis Rule antecedents are

expressed as Boolean conditions over the target model, and the Ontology Management Layer provides a service which is capable of directly evaluating these conditions against a particular model. The Analysis Assistant therefore fires an Analysis Rule by passing both that rule and the target model to the Ontology Management Layer.

This process can be illustrated by reference to the incomplete Package Router Problem Model which was introduced above. Applying the Completeness Analysis Assistant to the Package Router model, the requirements engineer first chooses to "Retrieve [Analysis] Rules". This, as illustrated in Figure 4.10, would retrieve four Analysis Rules which have previously been defined by a knowledge engineer: DomainSpecified; DomainCharacterised; MachineSpecified; and RequirementsSpecified. In order for the Problem Model to be complete in the terms of Jackson's Problem Frames approach, or at least in the terms of the particular formalisation of the approach in the current exemplar, the Problem Model must satisfy all of these Analysis Rules. The requirements engineer must, therefore, fire each rule in turn until either all rules have been satisfied or a rule fails when tested against the Problem Model. Suppose, then, that the Problem Model in its initial state — as stated above — contains Domain and Phenomena facts, and that these are related, but that the Model does not comprise — at this stage — either Requirements or Machine facts. In this case, the requirements engineer fires first the DomainSpecified rule, and then the DomainCharacterised rule, by selecting each in turn from the list in the Completeness Analysis Assistant and then pressing the "Fire Selected Rule" button. Both of these rules fire and, because the current Problem Model contains both Domain and Phenomena facts, both of these rules are found to have been satisfied. However, the Problem Model cannot at this point be determined to be complete, since not all rules have yet been fired, and so the requirements engineer additionally fires the MachineSpecified rule. This rule is not satisfied, because the Problem Model in its current state does not contain any Machine facts, and so the Problem Model is determined to be incomplete since it does not satisfy all of the rules which were retrieved. At this point, the Quality Analysis Assistant is activated and the requirements engineer progresses to this stage of the process.

Within the Quality Analysis Assistant — which is similar in structure to the Completeness Analysis Assistant, save that it does not offer a conclusion regarding the completeness of the Target Model (in this case, the Problem Model) — the requirements engineer again chooses to "Retrieve Rules". In the exemplar long-term memory knowledgebase, Quality and Completeness Analysis rules are not distinguished, and so the

same four Analysis Rules are retrieved: DomainSpecified; DomainCharacterised; MachineSpecified; and RequirementsSpecified. The requirements engineer recalls from Completeness Analysis that the MachineSpecified rule was not successfully satisfied by the Package Router Problem Model, and so chooses to refire this rule within Quality Analysis. Again, the rule fails but this time an Information Requirement is generated as per the process for enacting Quality Analysis. The Information Requirement stipulates that the requirement engineer should now progress to the Action stage of the Analysis-Action cycle in order to produce one or more Machine facts which are to be integrated into the Problem Model in order to represent the Machine that will control the Package Router system. The Quality Analysis Assistant is now disabled, and the Action tabs (Chunk-based Inference, Rule-based Inference and Elicitation) are activated.

The Inference Assistant enables the requirements engineer to attempt Inference in order to satisfy the Information Requirement that was produced by Quality Analysis. Inference has two sub-tasks. Chunk-based Inference produces new Facts to satisfy an Information Requirement by reusing and reifying Facts from a Model Chunk. Rule-based Inference produces Facts by firing Production Scripts over the current modeling context.

The Inference Assistant provides one tab to support each of these sub-tasks.

Figure 4.11 shows the Chunk-Based Inference tab.

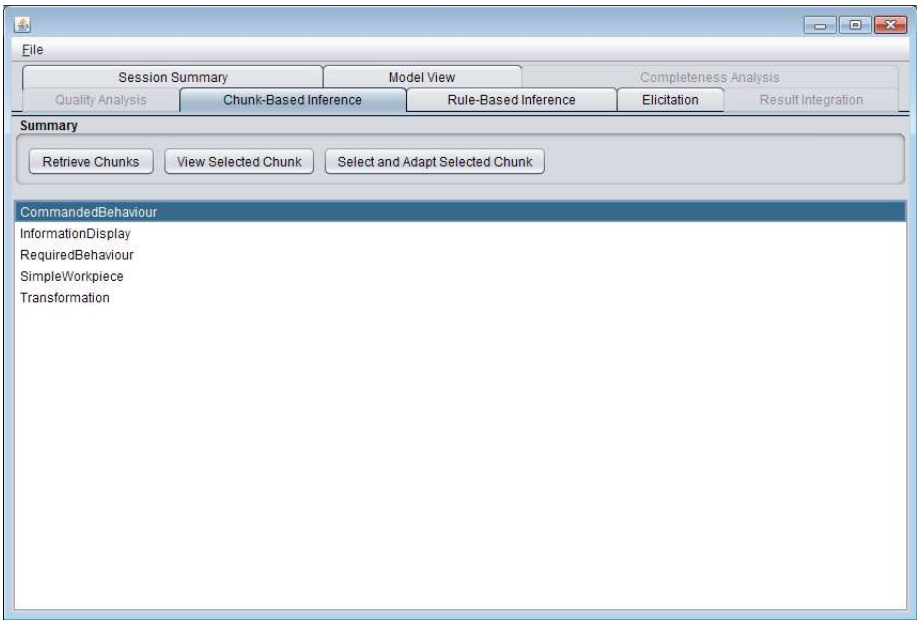

Figure 4.11: The Requirements Task Assistant's Chunk-based Inference tab

Figure 4.12 shows the Rule-Based Inference tab.

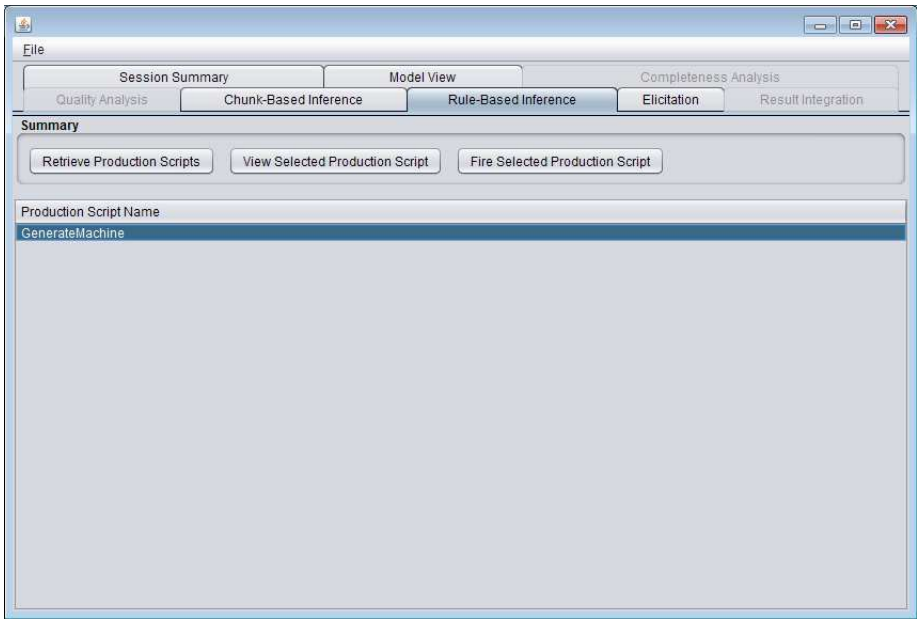

Figure 4.12: The Requirements Task Assistant's Rule-based Inference tab

Both Chunk- and Rule-based Inference start with Matching. At the request of the requirements engineer, the Inference Assistant passes the current modeling context and the current Requirements Engineering Task Type (Chunk- or Rule-based Inference) to the Matching Engine. The Matching Engine then returns a set of context-relevant knowledge structures which are appropriate to the current Requirements Engineering Task: Model Chunks for Chunk-based Inference or Production Scripts for rule-based inferences.

The retrieved knowledge structures are displayed to the requirements engineer through the Inference Assistant. The major visual difference between the two tabs is that, whereas the Rule-Based Inference tab contains a table in which key attributes of retrieved Production Scripts are summarised, the Chunk-Based Inference tab shows only the name of each retrieved Model Chunk. The Chunk-Based Inference tab also provides an option which utilises the Chunk Manager from the Presentation Layer to allow the requirements engineer to examine a retrieved Model Chunk before applying it.

The requirements engineer next selects one of the retrieved knowledge structures to apply. It is at this point that the two Inference sub-tasks diverge. In the case of Chunk-based Inference, Model Chunks are applied in the following way. The Chunk-based Inference tab passes the Model Chunk that was selected by the requirements engineer

to the Chunk Manager in the Presentation Layer and sets the Chunk Manager to edit mode. This allows the requirements engineer manually to reify the Model Chunk to suit the particular context of the target model. This may involve adding additional Facts, removing extraneous ones, or reifying the names of Facts with domain-specific names (e.g. changing the name of a "Vehicle" Fact to "Car"). By contrast, Production Scripts are executed automatically: the Inference Assistant passes both the selected Production Script and the current Modeling Context to the Production Engine which executes the Production Script.

Once the selected knowledge structure has been "fired" through the appropriate mechanism, the product will be a set of newly generated Facts. The final step of Inference is to Integrate these Facts into the target model. To achieve this, the Inference Assistant passes the generated facts to the Requirements Task Assistant's Integration Engine.

In order to illustrate the use of the Inference tabs in order to satisfy an information requirement, consider the exemplar information requirement that was described in relation to the Analysis Assistant and which mandates the creation of "one or more Machine facts". The RORE procedure stipulates that the requirements engineer should first attempt to satisfy this information requirement by applying Chunk-based Inference, and so, using the Requirements Task Assistant, the requirements engineer progresses to the Chunk-based Inference tab. Here, the requirements engineer presses the "Retrieve Chunks" button and the five Model Chunks, which were defined using the Model Chunk Editor (see Figure 4.7) and which correspond to the five elementary Problem Frames, are retrieved. Each elementary Problem Frame does indeed describe a Machine which will satisfy the requirement that the Problem Frame represents. However, each elementary Problem Frame also comprises a Requirement fact. The current Information Requirement stipulates only that a Machine fact should be generated in the current round of action, and neither requires or prevents the creation of a Requirement fact. The requirements engineer must, therefore, use their discretion in order to determine whether applying a Problem Frame chunk in order to satisfy the Information Requirement is appropriate at this point. Consider the two possible scenarios.

Firstly, consider the scenario in which the requirements engineer does choose to apply a Problem Frame to satisfy the Information Requirement for a Machine fact. The Package Router Problem comprises three elementary problem frames: a Commanded Behaviour frame, to capture the ability of the operator to start and stop the router; a Controlled Behaviour frame to capture the ability of the Machine to route

packages according to their destination; and an Information Display frame to capture the ability of the Machine to display misroute information. However, at the centre of the Package Router problem, just one Machine is required to perform all three tasks. The requirements engineer may, therefore, apply any one of the three elementary problem frames to generate the Machine in the current round of action: Commanded Behaviour, Controlled Behaviour or Information Display. In the event, the requirements engineer chooses to deal with the highest priority requirement first (the routing of packages) and thus applies the Controlled Behaviour problem frame in the current round of iteration. The requirements engineer thus selects the Commanded Behaviour problem frame, and presses the "Select and Adapt Selected Chunk" button. At this point, the Commanded Behaviour frame is displayed in a Model Chunk Manager instance which allows them to adpat the Commanded Behaviour Problem Frame by reifying the generic labels which are attached to the Machine and Requirement facts of the elementary Commanded Behaviour for the Package Router scenario, as shown in Figure 3.4. Having completed this task, all of the Domain, Phenomena, Machine and Requirements facts from the adapted Chunk are made available by the Requirements Task Assistant to the Integration tab so that the Information Requirement can be satisfied. In this case, a Machine Fact will indeed have been added to the Package Router model by the end of the Action round, but a Requirement fact will also have been added and so this course of action will have advanced the Model further than the Information Requirement intended. RORE leaves it to the discretion of a requirements engineer to determine whether or not this is appropriate.

Secondly, consider the alternative scenario in which the requirements engineer determines that the Information Requirement should be satisfied strictly — that only a Machine fact, and no other Facts, should be generated in this round of Action. In this case, having retrieved Model Chunks from Long-Term Memory, the requirements engineer finds that no elementary problem frame can strictly satisfy the Information Requirement in this sense. The requirements engineer thus progresses to the Rule-based Inference tab where they press the "Retrieve Production Scripts" button. A single Production Script is retrieved: the GenerateMachine script, which was described in Section 4.3.1.1. This fact generates a single Machine fact and assigns it a name which is derived from Domain Facts that are already defined in the Source Model. The requirements engineer chooses to fire this rule, which will generate the Machine fact to satisfy the information requirement, and a single Machine Fact is created with

the name “PackageRouter_RoutingOperator_StatusDisplay_Machine”. The requirements engineer has the option, once the fact has been generated, of adapting this name further if appropriate. The requirements engineer chooses to rename the Machine fact to “PackageRouterMachine” and then accepts it for integration into the target model.

The Elicitation Assistant supports the requirements engineer in enacting the “Elicitation” Requirements Engineering Task. Elicitation is a single task with no sub-tasks. It is performed only in the event that no Model Chunk or Production Script could be retrieved in order to satisfy the current Information Requirement. Figure 4.13 shows the Elicitation Assistant.

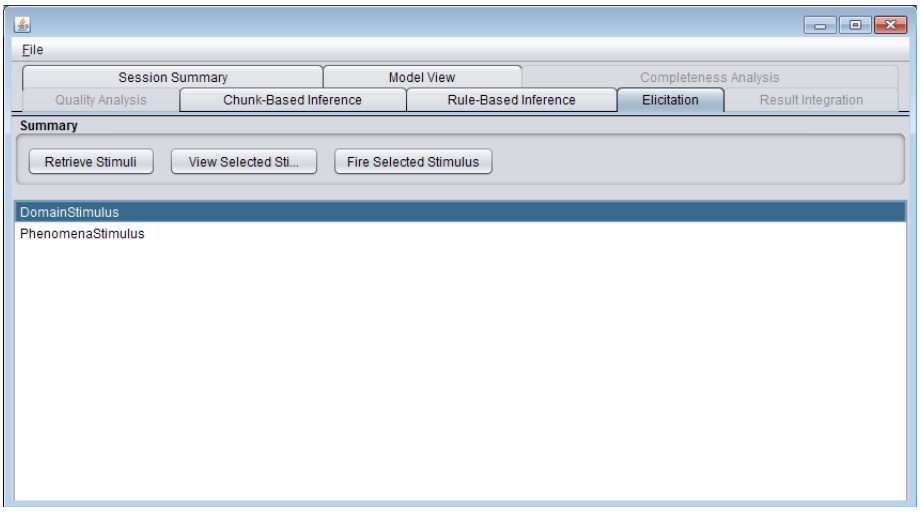

Figure 4.13: The Requirements Task Assistant’s Elicitation Assistant

Elicitation, as with all Requirements Engineering actions in RORE, begins with Matching. The Elicitation Assistant passes the current Modeling Context to the Matching Engine, and indicates that Elicitation Stimuli are required. The Matching Engine returns a set of Elicitation Stimuli to the Elicitation Assistant. These are displayed to the requirements engineer as a list. The requirements engineer can view the Elicitation Stimulus in order to evaluate whether or not it is likely to satisfy the current Information Requirement.

Having selected an Elicitation Stimulus to which to respond, the requirements engineer is asked to provide their response to that stimulus. The requirements engineer responds to the Elicitation Stimulus in the form of a Model Chunk. To support this, the Elicitation Assistant invokes the Presentation Layer’s Chunk Editor which enables

the requirements engineer to specify the Facts which the Response Chunk will comprise. The difference between responding to an Elicitation Stimulus in this way and applying a Model Chunk during Chunk-based Inference is that in the latter case the requirements engineer reifies an existing Model Chunk, whereas in the former case the requirements Engineer specifies the Model Chunk from scratch. Model Chunks which are defined as responses to specific Elicitation Stimuli are not in the current version of RORE stored in long-term memory.

Once the requirements engineer has specified and confirmed the Response Chunk, the Elicitation Assistant passes the Facts which are contained within that chunk to the Integration Engine so that the new Facts can be integrated into the target model.

If no Elicitation Stimulus could be retrieved to satisfy the current Information Requirement then the RORE process ends at this point, and the requirements engineer will need to refine the target model manually before loading it back into the Requirements Task Assistant for further refinement.

In order to illustrate the function of the Elicitation Stimulus in the context of the Package Router exemplar, consider a scenario which might have arisen were the RORE approach applied to the Package Router Problem Model at the very start of its development. In this scenario, no facts have yet been added to the Package Router Model (it is, in essence, a blank sheet of paper) and an Information Requirement was generated by Quality Analysis as a result of the failure of the Model to satisfy the DomainSpecified rule which stipulates that at least one Domain fact must be specified. Consider also that the requirements engineer has chosen to adopt a strict interpretation of this information requirement and so has not chosen to utilise elementary Problem Frame Model Chunks to satisfy the requirement, and has been unable to satisfy the Information Requirement by Rule-based Inference because the GenerateMachine production script (the only one to have been specified by the knowledge engineer) produces Facts of type Machine, not of type Domain and so cannot satisfy the current Information Requirement.

In this scenario, the requirements engineer will find themselves unable to satisfy the Information Requirement for at least one Domain Fact by either Chunk- or Rule-based Inference. Thus, the RORE process stipulates that they should attempt to satisfy the Information Requirement by Elicitation. The requirement engineer therefore selects the Elicitation tab in the Requirements Task assistant, and presses the “Retrieve Stimuli” button. Two Elicitation Stimuli are retrieved (as shown in Figure 4.13) DomainStimulus which can be used by the requirements engineer to elicit Domain Facts;

and PhenomenaStimulus, which can be used to elicit Phenomena Facts. The Information Requirement requests the production of a Domain Fact, and so the requirements engineer selects the DomainStimulus and presses "Fire Selected Stimulus". The Elicitation Stimulus is then displayed to the requirements engineer, requesting them to specify all of the Given Domains within the application domain. The requirements engineer, at this point, must use their own discretion and creativity in order to acquire — in consultation with the stakeholders — the facts which have been requested by the Elicitation Stimulus. However, in the context of the Package Router scenario they identify three Given Domains) (as shown in Figure 3.4): RoutingOperator; PackageRouter; and StatusDisplay. Using the Chunk Manager which the Requirements Task Assistant provides for responding to Chunk-based Stimuli, the requirements engineer enters these three Facts before confirming their response in order to pass the facts to Integration.

The Matching Engine is responsible for retrieving task- and context-relevant reusable knowledge structures to the Analysis, Inference and Elicitation Assistants. Each of these Assistant components utilise reusable knowledge structures to perform the functions for which they were designed. These reusable knowledge structures must be retrieved from long-term memory. The enactment of a particular Requirements Engineering Task using the Requirements Task Assistant is a semi-supervised process in which the requirements engineer makes the final decision as to which particular knowledge structure should be applied at any given moment in time. To minimise the number of reusable knowledge structures that are presented to the requirements engineer at any one moment in time, RORE attempts to retrieve from long-term memory only those knowledge structures that are most likely to satisfy the task goal in a given cycle. Identifying those knowledge structures is the task of the Matching Engine.

The Matching Engine achieves this goal in four broad steps, as illustrated in Figure 4.14. The detailed process is described in the following chapter (see Chapter 5).

The first three steps of Matching are "Filtering" procedures which are lightweight processes designed to remove from the pool of candidate matches the least likely candidates using the minimum possible computational resource. The final step - Conditional Matching - aims to seek positive matches between the remaining candidate structures and the current Modeling Context. Those candidate structures which cannot be matched are thrown out of the candidate pool. These five steps can be summarised as follows:

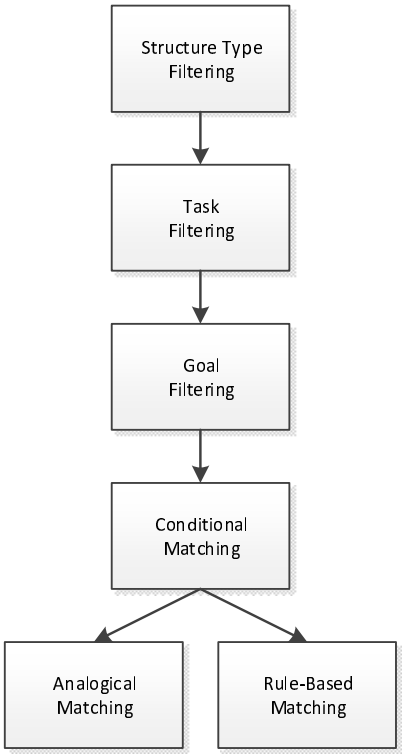

Figure 4.14: Summary of the Matching Process

1. **Structure Type Filtering** is the first step in the Matching process. During this step, the type of the RORE knowledge structure is evaluated and compared against the type of Requirements Engineering Task which is being performed. All knowledge structures which are not appropriate to the task type are removed from the candidate pool. Knowledge structures are considered appropriate as follows: Analysis Rules are only appropriate for Analysis; Model Chunks are only appropriate for Chunk-based Inference; Production Scripts are only appropriate for Rule-based Inference; Elicitation Stimuli are only appropriate for Elicitation.

2. **Task Filtering** removes candidates for which the Phase and Activity which are stipulated by their Index Description does not match the Phase and Activity which are currently being performed by the requirements engineer.

3. **Goal Filtering** further narrows the candidate pool by comparing the Index Description of remaining candidates against the source model and the current Information Requirement to ensure both that the source model contains Facts of the type required as input to the candidate structure, and that the candidate structure produces the type of Fact which is mandated by the current Information

Requirement. Candidate structures not satisficing this condition are thrown out of the pool.

4. **Conditional Matching** checks sets of detailed conditions - which are defined in the Index Description of the remaining candidate structures - against the Modeling Context. Two types of matching are supported at this stage. Analogical Matching [Mai92, Gen83] uses Chunk-based Conditions to determine whether a candidate structure is applicable to the current context. Chunk-based Conditions are are specified as part of Index Descriptions in the form of Model Chunks. Analogical Matching looks for shared structure between the source or target model on the one hand and a Model Chunk, which is attached to the Chunk-based Condition, on the other hand. The second form of matching is Rule-based Matching, which uses Analysis Rules to determine a match between a reusable knowledge structure and the Modeling Context. During Conditional Matching, reusable knowledge structures are removed from the candidate pool if their Index Descriptions define just one condition which is not satisfied by the current Modeling Context.

The result of Matching is, then, a set of reusable knowledge structures which has been narrowed to contain only those structures which (assuming the knowledge engineer has designed long-term memory effectively) are most likely to resolve the goal of the current Requirements Engineering Task.

The Production Engine has responsibility for interpreting and executing Production Scripts. Production Scripts are one of the four types of reusable knowledge structure described by RORE. As discussed above, Production Scripts describe reusable transformations over source and target models. Because following a Production Script manually would be a laborious and error-prone process, the Requirements Task Assistant executes Production Scripts automatically. This is the function of the Production Engine.

The Production Engine comprises a hierarchy of objects, each of which corresponds to a construct in the language through which Production Rules are specified (see Chapter 6). Each object is specialised to support both the parsing of a source statement into an internal representation of that statement and the execution of that statement. The Production Engine also provides a number of additional objects which support memory management during the execution of a production script. Figure 4.15 shows the design of the object structure into which Production Scripts are parsed.

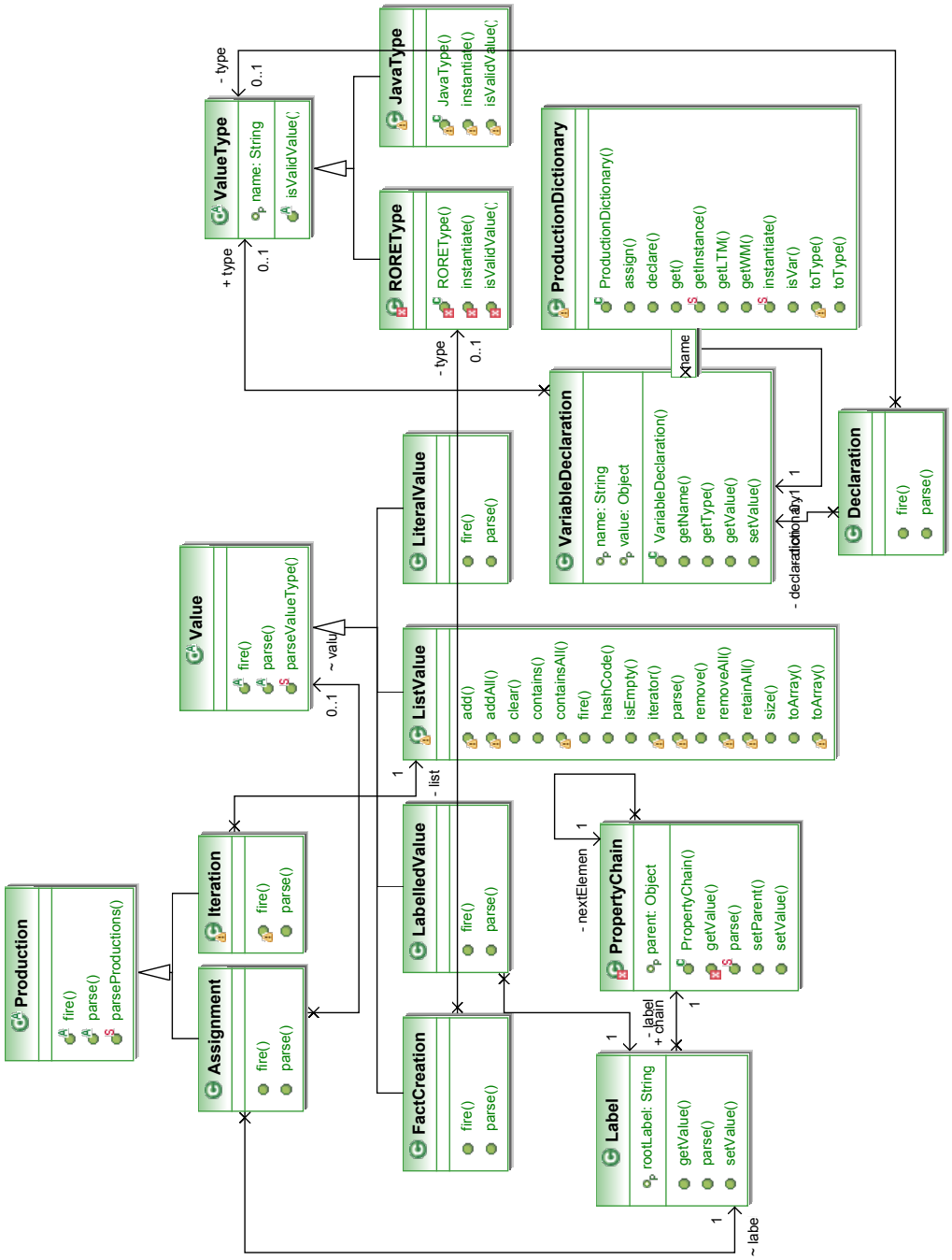

Figure 4.15: The Internal Design of the Production Engine

The Production Engine achieve the execution of a Production Script in three major steps:

1. The Production Engine accepts a Production Script as input. It evaluates each Production Rule within that Production Script to determine the type of statement it represents. It then passes the statement to an object of the appropriate type which parses the Production Rule into its own internal representation. The result is a hierarchy of objects which mirrors the structure and semantics of the Production Script itself.

2. The Production Engine then traverses this hierarchy, firing each object in turn. Each object provides a "fire" operation which, when called, enacts the type of production operation it represents;

3. The Production Script specifies a value which should be returned from the Facts which it produces, and once the object hierarchy has been completely traversed, the Production Engine returns this value to the Inference Assistant.

The Integration Engine allows the requirements engineer to integrate Facts which are produced or reified through the two productive Requirements Engineering Tasks (Inference and Elicitation) back into the Target Model. As this chapter has shown, each of these productive Requirements Engineering Tasks produces new Facts which aim to satisfy an Information Requirement. In RORE, the refinement of a Model is achieved either by adding additional Facts to a Model or by replacing existing Facts with new Facts. These are the two Integration strategies which this thesis will refer to as Additive and Substitutive Integration respectively.

Figure 4.16 shows a screenshot of the Integration tab.

Integration is achieved by pairing new Facts to existing Facts in the target model. Facts should be paired if a new Fact refers to the same individual as a Fact in the target model. The Integration Engine requires that this be done manually by the requirements engineer. The Integration tab displays to the requirements engineer the Facts that were produced during the current Analysis-Action cycle, as well as the Facts which the target model current comprises, and allows the requirements engineer to pair new Facts with existing Facts.

The rest of the process is fully automatic, and is initiated by the requirements engineer confirming that they have appropriately specified all desired Fact pairs. The

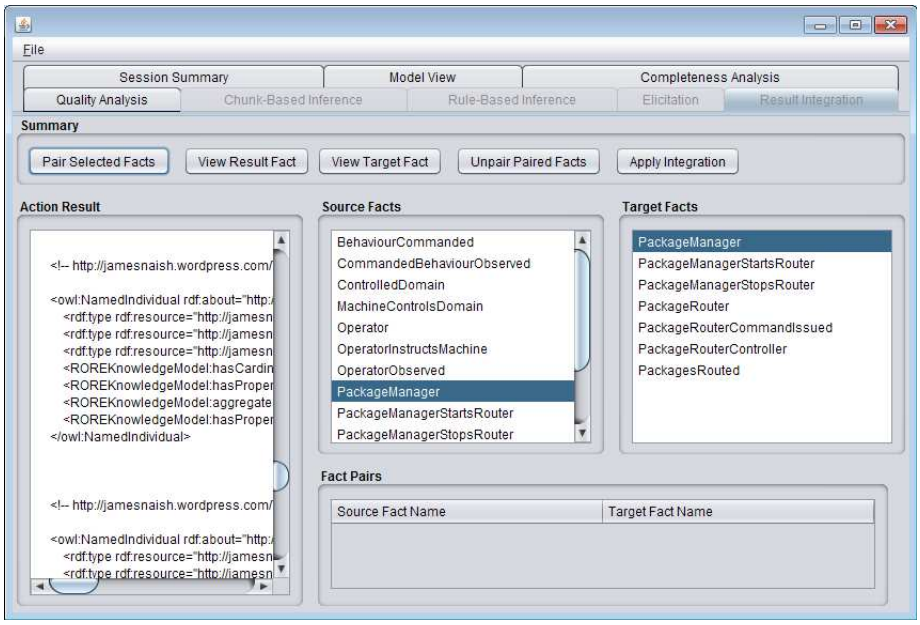

Figure 4.16: The Requirements Task Assistant's Integration tab

Integration Engine iterates over these Fact pairs and performs one of two actions depending on the preferred Integration strategy:

- In the case of Additive Integration, the Integration Engine appends the properties of each new Fact to the properties of its paired target Fact. Any new Facts which were not paired are simply added as "floating Facts" to the target model: they will not be related to any existing Fact within the target model;

- In the case of Substitutive Integration, the target Fact is entirely removed from the target model and the new Fact is inserted into the target model in its place. The associations in which the original target Fact was involved will be maintained. Again, unpaired new Facts will simply be appended to the target model.

Figure 4.16 shows the Integration Management Tab populated with Facts which were produced by the application of a Problem Frame Model Chunk to satisfy an Information Requirement to produce a Requirement Fact by means of Chunk-based Inference. The central "Source Facts" column displays all of the Facts which were produced by Chunk-based Inference in response to the current Information Requirement, and many of these Facts have been reified for the Package Router Problem Model. For instance, the phenomena involved in the Commanded Behaviour Problem Frame have been named to "PackageManagerStartsRouter" and "PackageManagerStopsRouter".

The "PackageManager" Fact represents the operator who commands the PackageRouter Machine itself. It should be noted that some of the Facts in the Source Facts list are duplicates of Facts which are already in the Target Model. Integration, therefore, does not simply append new Facts to the Target Model, but does so in a logical way which avoids duplicates. In order to achieve a proper integration, therefore, the requirements engineer must pair facts from the Source Fact List and Target Model where facts in the former duplicate facts in the latter. In the given example, the requirements engineer would pair, for instance, the "PackageManager", "PackageManagerStartsRouter" and "PackageManagerStopsRouter" facts in the Source Fact List with the same facts in the Target List. Once the requirements engineer has specified all appropriate Fact Pairs, they then press the "Apply Integration" button. The Requirements Task Assistant will then execute the integration algorithm, which appends the child facts of all Source Facts to the list of child facts of the Target Facts to which those Source Facts are paired. Unpaired Source Facts are appended directly to the Target Model. Once this process has been completed, the round of Action is also complete, and the requirements engineer returns to the Completeness Analysis tab in order to re-evaluate the Target Model.

4.3.3 The Presentation Layer

The Presentation Layer is a small library which defines two reusable user interfaces which enable users to create new instances of RORE knowledge structures. Both interfaces are shared across both the LTMM and the Requirements Task Assistant. As such, both interfaces cut across the two perspectives of RORE:

- **The Chunk Manager** provides functionality which enables users easily to create new Model Chunks;
- **The Fact Manager** provides the functionality which enables users easily to define new Facts of a given type. Because Model Chunks essentially aggregate sets of Facts, the Fact Manager is utilised primarily through the Chunk Manager.

The Chunk Manager allows the user to define new Model Chunks which instantiate a specified Model Chunk Type. Model Chunk Types aggregate sets of Fact Types. Model Chunks instantiate Model Chunk Types in the sense that a Model Chunk aggregates a set of Facts such that each Fact instantiates a Fact Type which is aggregate by

the Chunk Type that the Model Chunk itself instantiates. The Chunk Manager allows the user to do this easily, and also enforces this constraint.

Figure 4.17 illustrates the Chunk Editor.

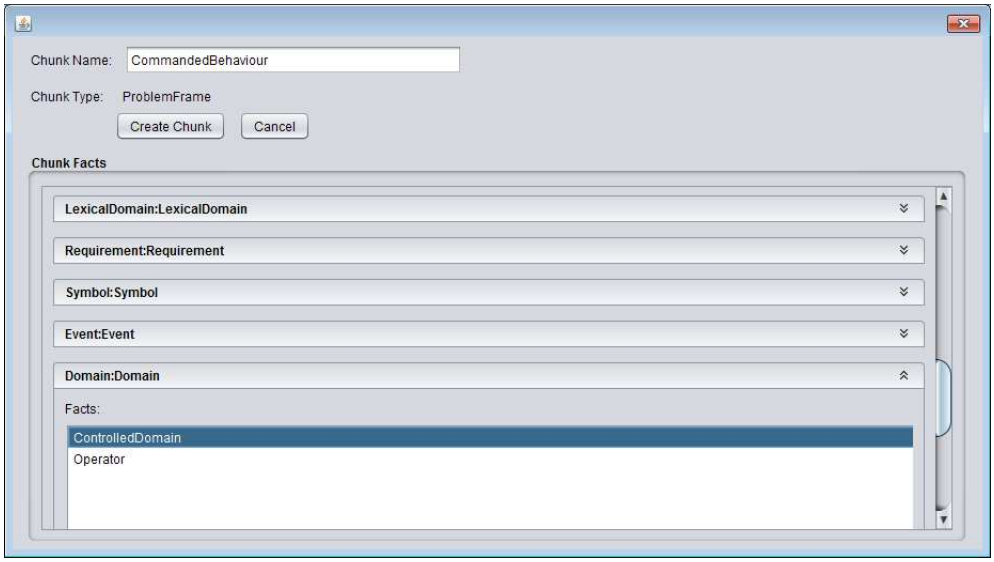

Figure 4.17: The Chunk Manager

The Chunk Manager can be invoked in one of two modes. In Edit Mode the user can both view the Facts which the Model Chunk currently aggregates, and can manipulate those facts (Create, Update and Delete). In View Mode, all write operations (Create, Update and Delete) are disabled.

The Chunk Manager can only manipulate a single Model Chunk at a time (the "Current Chunk"). The Chunk Manager comprises a set of panels each of which is associated with one Fact Type which the Current Chunk Type aggregates. The Chunk Manager allows users to add, view, remove and edit Facts for each of these Fact Types. When the user chooses to view or manipulate a Fact of a given type, the Chunk Manager displays that Fact to the user through an instance of the Fact Manager interface. The Fact Manager will always be loaded in the same mode as the instantiating Chunk Manager.

Figure 4.17 shows the Chunk Manager populated with the abstract Commanded Behaviour Model Chunk. The Chunk Type has been set to "ProblemFrame", and this determines the Fact Types which the Chunk will comprise. Collapsible panels in the "Chunk Facts" segment of the Manager allow the user (requirements engineer, or knowledge engineer depending on the context within which the Chunk Manager component is invoked) to add Facts of each type which the Chunk Type comprises. In

the case of the Problem Frame Chunk Type, the comprised Fact Types are: Domain (Generic, Biddable, Causal or Lexical); Phenomenon (Symbol; Event or State); Requirement and Machine. The Domain panel is expanded and the two given Domains which comprise the Chunk Manager have been specified: the Operator Domain, which represents the operator who issues the commands; and the ControlledDomain, which the commands affect. Each of these Facts can be edited by hilighting it and pressing an "Edit Fact" button, which displays the selected Fact in the appropriate Fact Manager (the Complex Fact Manager for Domain Facts).

The Fact Manager allows the user to define new Facts which can be integrated either into long-term memory as part of a Model Chunk, or into a target model. The structure of the Fact Manager mirrors closely the structure of the Fact Type Editor in the LTMM's Metamodeling Assistant, providing users with an interface which guides them through the process of creating a Fact which instantiates a given Fact Type.

The Fact Manager consists of two sub-components for creating and managing Complex and Simple Facts respectively. Figure 4.18 shows the Complex Fact Editor.

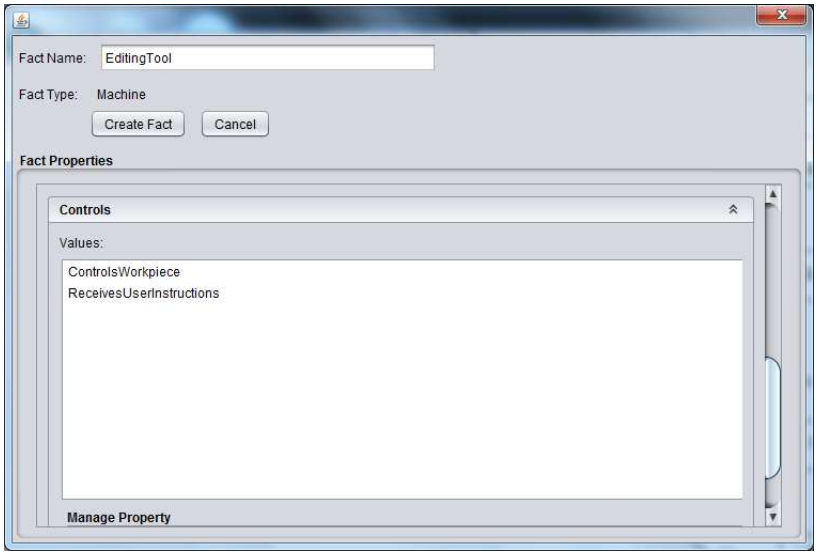

Figure 4.18: The Complex Fact Editor

Like the Chunk Manager, the Fact Manager can be invoked in one of two modes. In Edit Mode the user can both view Facts and their associated values, and can manipulate those Facts (Create and Update). In View Mode, all write operations (Create, Update and Delete) are disabled.

The Complex Fact Editor allows users to create Facts which instantiate a Complex Fact Type. To do this, the Complex Fact Editor must be passed a Complex Fact Type which the Complex Fact will instantiate. The Complex Fact Editor displays the name of the Fact Type which the Fact that is currently loaded into the editor (the "Current Fact") must instantiate. A new Complex Fact can be created by specifying the name of the new Fact. The Complex Fact Editor maintains a record of the details of the new Fact to be returned to its parent component when the editor is closed.

The Complex Fact Editor also displays a list of panels: one for each Property Type which is associated with the Complex Fact Type which the Current Fact instantiates. Each of these panels displays a list of all of the Facts which are associated with the Current Fact via a Property of that Property Type. Users can remove these property values, or add an additional instance of that Property Type to the Current Fact. Both operations update the internal record of the attributes of the Current Fact which the Fact Editor maintains. Because Complex Facts are recursive in nature (they aggregate less Complex - and ultimately Simple - Facts), choosing to add a new value to the Current Fact for a particular Property causes a second instance of the Fact Type Manager to be displayed. Whether or not the Complex or Simple Fact Editor is used to specify the value of the new Property instance is dependent on the value type of that Property: if the Property Type has a Simple Fact Type as its value, then the Simple Fact Editor will be used; conversely, if the Property Type has a Complex Fact Type as its value, then the Complex Fact Editor will be used.

Figure 4.18 presents the Complex Fact Editor with the Machine Fact ("EditingTool") from the Simple Workpiece Problem Frame Model Chunk. The type of the fact is set to Machine, and this determines the panels that are displayed in the Fact Properties section of the interface. Each collapsible panel within this section corresponds to a single Property of the Complex Fact Type of the Fact being edited. The Machine Fact Type has the "Controls" and "Satisfies" Properties which link Machine Facts to Phenomenon and Requirements Facts respectively, and so collapsible panels are displayed for each of these Properties. The EditingTool Fact has two values defined for the Controls property: "ControlsWorkpiece", which corresponds to the phenomenon which the workpiece editor manipulates within the workpiece itself; and "ReceivesUserInstructions", which corresponds to the phenomenon by which the workpiece editor receives instructions from a user. Facts can be added, removed and edited for each of the Properties of a Complex Fact.

The Simple Fact Editor mirrors the structure of the Simple Fact Type Editor. The

Simple Fact Type for the Current Fact should be passed to the Simple Fact Editor by its parent component when it is opened for editing. To create a new Simple Fact, a user need only specify the name and value of a simple fact. When the Simple Fact Editor is closed, these details are then passed back to the parent component.

4.3.4 The Ontology Management Layer

At their core, both the LTMM and the Requirements Task Assistant are prototype tools for manipulating two kinds of knowledge base:

- **Long-Term Memory Knowledge-bases** which comprises project-independent and reusable knowledge;
- **Working Memory Knowledge-bases** which comprises project- and domain-specific knowledge.

Both kinds of knowledge base are formalised, in the RORE prototype, through the Web Ontology Language (OWL) [BVHH+04]. OWL was chosen for this task because it offers rich and flexible support for conceptual modeling, as well as powerful and decidable reasoning support [SPG+07]. Furthermore, the OWL API provides an existing open-source library (the OWL API [HB11]) for working with OWL ontologies.

The Ontology Management Layer wraps around the OWL API in order to provide the services which the LTMM and Requirements Task Assistant each require to manipulate and query these knowledge bases. Furthermore, the Ontology Management Layer provides an interface to these services at a level of abstraction which corresponds to that of the RORE Knowledge View (see Section 3.7), rather than at the level of generic OWL constructs. The Ontology Management Layer thus enables the LTMM and Requirements Task Assistant to work with the RORE knowledge bases at precisely the level of abstraction that they require, which allows both tools to focus on the specific functionality which they are designed to provide.

The Ontology Management Layer comprises two major components:

1. **The Ontology Manager** which provides a single interface through which all of the services of the Ontology Management Layer are exposed;
2. **The Ontology Factory** provides wrapper objects which support the CRUD operations for each kind of knowledge structure that is defined in the RORE Knowledge View (see Section 3.7).

The Ontology Manager provides a single interface through which higher-level layers can access the services which are provided by the Ontology Management Layer. The Ontology Manager exposes two major kinds of functionality:

- Basic CRUD [MI83] operations for each type of knowledge structure which is defined in the RORE Knowledge View;
- Powerful query features to support complex reasoning over RORE knowledge bases (specifically over working memory).

The RORE prototype does not, itself, implement this functionality but instead provides a thin wrapper around two open-source libraries which do offer this functionality:

- Basic CRUD operations are implemented using Redmond's Protg code generation library [Red12]. This library provides support for the implementation of factory objects which facilitate the manipulation of OWL ontologies. The RORE Ontology Factory object is treated as a separate component of the Ontology Management Layer, but it is accessed from outside the Ontology Management Layer through the Ontology Manager;
- Querying functionality in the Ontology Management Layer Derivo's SPARQL-DL API [Sys12] which provides support for reasoning over and querying of OWL ontologies through the SPARQL-DL query language [SP07].

The purpose of the Ontology Manager, therefore, is not directly to implement the functionality required to access and query the RORE knowledge bases since the hard work is done by the Redmond and Derivo libraries. Instead the Ontology Manager wraps around these two libraries - which have radically different interfaces - in order to provide a common, intuitive interface through which the services provided by the Ontology Management Layer as a whole can be accessed. This allows components that are external to the Ontology Management Layer to work with objects whose structure mirrors those of the RORE knowledge structures, while encapsulating the specific mechanisms through which those objects are written to, read from and queried within RORE's knowledge bases.

The major services which are exposed by the Ontology Manager accept these wrapper objects as input, and return them as output. The Ontology Manager has responsibility for interpreting those messages and routing them to the appropriate library for the message to be processed.

The Ontology Factory provides a library of wrapper objects for each of the knowledge structures which is defined by the Knowledge View of RORE. These wrapper objects support basic CRUD operations for each knowledge structure type in a RORE knowledge base. The wrapper objects also support the linking of knowledge structures that are specified within the knowledge base in accordance with the relationships which are defined between knowledge structure types in the Knowledge View. Figure 4.19 illustrates the wrapper objects which are defined for the reusable knowledge structures in long-term memory.

Each wrapper object was generated automatically through Redmond's Protg code generation plug-in [Red12]. The plug-in produces for each TBox concept in a given OWL ontology a wrapper object similar to that shown. The resultant wrapper object is then used to instantiate the TBox concept as an entity in an OWL ABox. The wrapper objects utilise the OWL API [HB11], which is the de facto standard library for working with OWL ontologies, to manipulate RORE knowledge bases. Each wrapper object provides a set of operations which specialise the interface provided by the OWL API to support manipulation of a given type of TBox concept: in the case of the RORE Ontology Factory, a RORE knowledge structure. The interface of each wrapper object is defined exclusively in terms of other wrapper objects, and each wrapper object is responsible for translating this request into a message which is compatible with the OWL API.

4.4 Summary

This chapter has introduced the design of RORE and a prototype of that design. The RORE architecture comprises four macroscopic layers. The Long-Term Memory Management layer is designed to support the Knowledge Engineering perspective of RORE. It provides a set of user interfaces which support the knowledge engineer in describing requirements modeling notations, requirements engineering methods and in building libraries of reusable requirements knowledge structures. The Long-Term Memory Management layer is complemented by the Requirements Task layer which supports the Requirements Engineering perspective. The Requirements Task Assistant provides a set of user interfaces which support the five Requirements Engineering Tasks which RORE defines. Each of these interfaces support the requirements engineer in retrieving and then applying a task-appropriate reusable knowledge structure. For the Inference

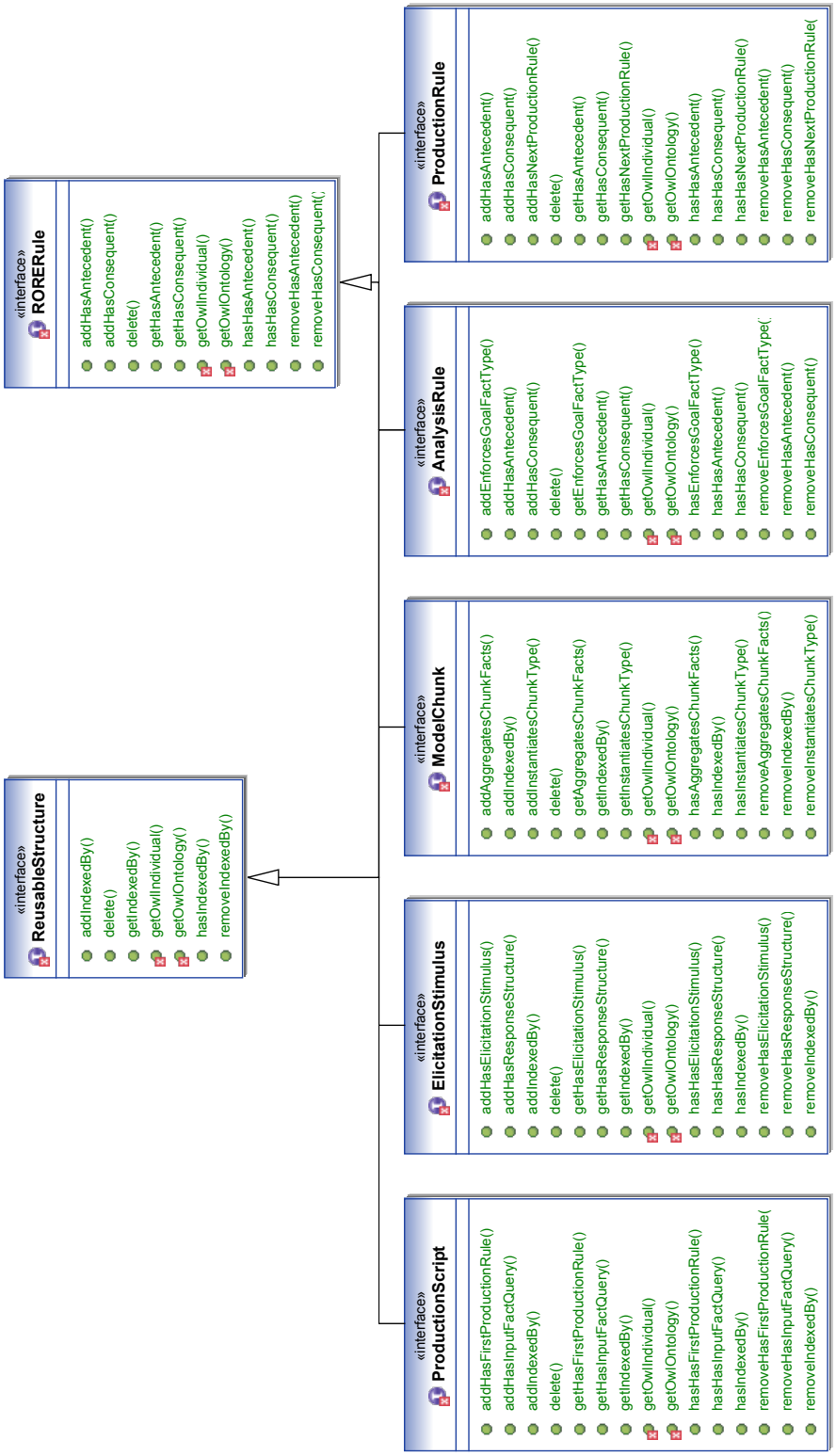

Figure 4.19: Wrapper Objects for Knowledge Structures in RORE Knowledge-bases

and Elicitation tasks, each of which produce new Facts to satisfy an Information Requirement, the Requirements Task Assistant also provides an interface to support the Integration of these facts back into the target model.

Both the Long-term Memory Management and Requirements Task layers depend on two other layers. The first of these - the Presentation Layer - defines two reusable user interfaces which are shared in common by the Long-Term Memory Management and Requirements Task Layers. These user interfaces are designed to support the creation and display of Model Chunks, and both Complex and Simple Facts. The Long-Term Memory Management and Requirements Task layers also both depend on the Ontology Management layer which is a thin wrapper around the RORE knowledge bases in order to provide the ability of manipulate and query those knowledge bases. It is through the Ontology Management layer that other layers interact with those knowledgebases.

Chapter 5

Reusable Tasks and Procedures for RORE

5.1 Introduction

This chapter fully specifies the RORE approach to requirements engineering. The approach utilises four classes of in-built operation such that each class is responsible for managing a different aspect of the overall process.

These four classes are:

- **Control Procedures** which are responsible for co-ordinating the overall sequence of events and interactions between activities. Control procedures are also responsible for managing resources, within a RORE session;

- **Requirements Tasks** represent those activities which are responsible for building and refining requirements models;

- **Reuse Procedures** have responsibility for the retrieval and reification of knowledge structures to support requirements tasks;

- **Reasoning Strategies** are procedures which have been designed to make use of different kinds of knowledge structure to produce new facts which can be integrated into a model.

This chapter presents the detailed procedures for each aspect of this approach.

5.2 Requirements Engineering Tasks

Requirements tasks are all of those activities which this research abstracted from existing requirements literature and which are directly relevant to the process of gathering requirements about an application domain and building models of both the domain and the requirements. There are three top-level requirements tasks:

1. **Analysis** (see Section 5.3) which involves the evaluation of a model in order to determine the quality of that model and thus the necessity for further refinement;

2. **Inference** (see Section 5.4), an action which exclusively utilises either procedural or declarative reuse to produce new information which can be integrated into a model to refine that model;

3. **Elicitation** (see Section 5.5), an action which reuses scripts describing dialogue between the requirements engineer and a stakeholder with the goal of eliciting new information from that stakeholder. The requirements engineer uses this script as the basis for further communication with the stakeholder, and formalises the response according to the response structure given within the script. The formalised response is the new information which is integrated into the script;

Collectively, these top-level actions form the Analysis-Action cycle described in the previous chapter. These three actions are executed in a predefined order, in a continual loop, until it is determined by means of Analysis that the given requirements model is sufficiently "complete", as shown in Algorithm 1. The order in which requirements tasks are executed is determined by three sequencing criteria:

1. Acquire information efficiently;

2. Maximise information gain;

3. Prioritise calculation approaches over look-up approaches.

A method that acquires information from memory structures which are directly accessible to the processor will be more efficient than a method which must acquire information from external sources. This is because the latter method will incur overheads, which will not be incurred by the former method, as a result of the additional information transfer operations needed to communicate the necessary information from the source to the processor. Given this, one can expect that Inference will prove to be more

```
Data: Context = An Initialised Modeling Context;
Complete = false;
Failed = false;
Result: Target model refined such that all Analysis Rules satisfied
while not Complete and not Failed do
    Context.InformationRequirement = analyse(Context);
    if Context.InformationRequirement != null then
        Success = infer(Context);
        if not Success then
            Success = elicit(Context);
            if not Success then
                log(Context, failed);
                Failed = true;
            end
        end
        refreshWorkingMemory(Context);
    else
        Complete = true;
    end
end
```

Algorithm 1: The Top-Level Procedure for Executing Requirements Tasks

efficient than Elicitation as a means of acquiring information, because both operations require reusable structures to be retrieved and applied, but Elicitation additionally requires the requirements engineer to initiate a dialogue with a stakeholder to acquire the desired information. This prediction is given further weight by its consistency with the common argument in favour of requirements-level reuse, that it will reduce the effort involved in building good quality requirements documents [Sut02, NZ11].

Two additional criteria which are important in practice when choosing a transformation to apply to a target model would be:

- The likelihood that a particular transformation will satisfy an information requirement;
- The likelihood that a particular transformation will produce accurate inferences about the subject of the target model.

These additional criteria address, in brief, the quality of the inferences over a target model which a particular transformation produces. However, this research did not in fact apply these criteria in order to sequence the generic requirements tasks in RORE

because both criteria relate to the *subject matter* (namely the contents) of a transformation. Within RORE, the generic requirements tasks are generic precisely because they assume no inherent knowledge of any particular subject, or application domain, beyond the formalism through which such knowledge is expressed. Knowledge about an application domain — and thus about the particular nature of transformations over a target model — is contained in reusable knowledge structures, and not in the generic tasks. The generic tasks simply describe procedures for applying those different kinds of reusable knowledge structure in order to analyse and transform a target model. As such, no particular generic task is inherently any more, or less, likely to satisfy an information requirement than any other. What is important, from a quality perspective, is the contents of the reusable knowledge structure by which that generic task is parameterised in a given RORE cycle. Accordingly, the quality of inferences which are produced by a generic task was not a determining factor in the sequencing of those tasks. Nevertheless, the author of this thesis recognises that the quality of the inferences which are produced by a generic task would be an important consideration when applying the RORE approach in cases outside of the assumptions which were stipulated in Section 3.2.

Two particular procedures are common to all, or most, requirements tasks in RORE: the matching procedure and the Integration procedure. The matching procedure (described in Section 5.8) retrieves knowledge structures from long-term memory that satisfy the needs of the current modeling context ("modeling contexts" are specified in Section 6.4.2 of the next chapter). This is critical to the reuse-oriented nature of RORE. Integration (described in Section 5.10) is used by both the Inference and the Elicitation tasks. Integration takes the set of facts produced by productive requirements tasks (Inference and Elicitation), and either appends them to the target model, or uses them to replace existing facts within the target model.

5.3 Analysis

Analysis is an evaluative task which does not itself modify any models, but evaluates models in order to support decisions about what further refinements, if any, are necessary. To achieve this aim, the requirements engineer retrieves Analysis Rules from long-term memory and tests one rule at a time against the target model. Each rule should assert some standard which the model should satisfy in order to be complete. If a model fails an Analysis rule then the implication is that the standard asserted by

that rule has not been met and an Information Requirement is generated to this effect. If, however, all retrieved rules are passed during Analysis then the model is deemed complete and no further refinement is required.

Analysis has two sub-tasks:

- **Completeness Analysis** is responsible for determining the degree of completeness for a model and thereby deciding whether or not further requirements are necessary. Completeness is defined in terms of a set of analysis rules, which are stipulated by the knowledge engineer within long-term memory;
- **Quality Analysis** is responsible for evaluating the quality of a model against a set of analysis rules — which have been specified within the KE perspective — with a view to deciding specifically in what way the model is limited and thus what kind of refinement should be applied next. The particular metrics against which the quality of a target model is judged are encoded within the set of analysis rules by which that model is tested.

These sub-tasks should be performed in this order, in accordance with the procedure described by Algorithm 2, so that Quality Analysis is only performed in the event that a model is deemed, by Completeness Analysis, to be incomplete.

```
Data: Context = A Modeling Context;
Complete = false;
Result: Complete AND Null, OR Not Complete AND Information
        Requirement Returned
Complete = analyseForCompleteness(Context);
if not Complete then
  | return analyseForQuality(Context);
else
  | return null;
end
```

Algorithm 2: The top-level procedure for Analysis

Although the two sub-tasks are very similar in their structure, they are distinguished in two regards. Firstly, they are distinguished by the output which they produce. Completeness Analysis produces a Boolean value indicating the completeness of a model, whereas Quality Analysis produces an Information Requirement which specifies the conditions that the next refinement should meet. Secondly, the two sub-tasks draw the rules on which they are based from two distinct libraries. This allows optimised sets of rules to be optimised for each of the two tasks.

5.3.1 Completeness Analysis

During Completeness Analysis, *all* of the completeness rules retrieved from long-term memory which are appropriate to the current modeling context are tested one at a time against the current target model until either a rule fails or there are no more rules to test. The product of this process is a single decision (yes or no) which indicates whether or not the model is complete. The model is complete if all were tested and none failed. The procedure for making this decision is described by the algorithm 3.

```
Data: Context = A Modeling Context;
Rule = null;
Complete = true;
Result: All rules satisfied and Complete == true
Context.RetrievedStructures = match(Context, AnalysisRules);
while Rule = pop(Context.RetrievedStructures) and Complete do
    if not fireAnalysisRule(Rule, Context) then
        Complete = false;
    end
end
return Complete;
```

Algorithm 3: The procedure for performing Completeness Analysis

5.3.2 Quality Analysis

If, and only if, the model is not complete, then the requirements engineer next performs Quality Analysis. To perform Quality Analysis, the requirements engineer retrieves *all* relevant quality rules from long-term memory which are relevant to the current modeling context. As in Completeness Analysis, these are tested on at a time against the current target model until either a rule fails or there are no further rules. In the event that a rule fails, the requirements engineer produces an Information Requirement according to the procedure described in Section 5.6.2. If, however, all retrieved rules are fired without any rule failing, then it is assumed that the completeness rules were insufficiently well-defined and that the model is, in fact, complete. That being the case, the requirements engineer notes this case, and terminates the Analysis-Action cycle.

The procedure for Quality Analysis is described by Algorithm 4.

```
Data: Context = A Modeling Context;
Context.RetrievedStructures = match(Context, AnalysisRules);
Rule = null;
InformationRequirement = null;
Result: Context.InformationRequirement is not null
Context.RetrievedStructures = match(Context, AnalysisRules);
while Rule = pop(Context.RetrievedStructures) AND InformationRequirement
== null do
    if not fireAnalysisRule(Rule, Context) then
        InformationRequirement = constructInformationRequirement(Rule)
    end
end
return InformationRequirement
```

Algorithm 4: The procedure for Quality Analysis

5.4 Inference

Inference is the first of two productive actions defined within RORE. The goal of this task is to produce information which satisfies the information requirement that was generated during Analysis. In particular, Inference does this based on the reuse of an appropriate knowledge structure which is retrieved from long-term memory. In accordance with the three problem-solving steps of heuristic classification, the requirement engineer first matches the current modeling context to an appropriate knowledge structure in long-term memory, then fires this knowledge structure to produce a refined, yet abstract, set of facts and then reifies and integrates these facts into the target model. In this way, Inference advances the state of the target model in order to produce a more "complete" model, where completeness is defined in terms of the analysis rules in long-term memory. However, as discussed in chapter 3, without adequate stakeholder validation — process for which are not built in to the current version of the RORE framework – there is no guarantee that any inferences drawn by RORE will be correct.

Inference can utilise two different kinds of knowledge structure in order to guide the production of new information to support refinement. These two kinds of knowledge structures offer the desired mix of declarative and procedural reuse strategies. The two kinds of knowledge structure are:

- **Model Chunks** (see Section 6.2.2 in the next chapter), which are declarative structures representing abstract facts. These abstract facts describe some aspect of an application domain or its requirements and can be directly integrated into a model to significantly increase the information content of that model;

- **Production Scripts** (see Section 6.2.3), which are procedural knowledge structures that describe how new facts can be produced by transforming facts in the source model. No facts are explicitly specified by a production script, and the output of a production script will tend to be finer-grained than the output of a model chunk.

During inference, the requirements engineer will attempt to use both kinds of knowledge structure to resolve an information requirement until the information requirement has either been resolved or no more relevant knowledge structures can be identified. Two sub-tasks are defined:

1. **Chunk-based Inference** (see Section 5.4.1), which attempts to utilise model chunks to resolve an information requirement;

2. **Rule-based Inference** (see Section 5.4.2), which utilises production scripts to resolve an information requirement.

Because this thesis anticipates that model chunks will typically produce more facts than production scripts, and therefore yield a greater information gain for a similar degree of effort, the two sub-tasks are executed in this order. The top-level procedure for Inference is given below in Algorithm 5.

```
Data: Context = A Modeling Context;
Success = false;
Result: Success == true
Success = inferByChunk(Context);
if not Success then
  | Success = inferByRule(Context);
end
return Success;
```

Algorithm 5: The Top-Level Procedure for Inference

5.4.1 Chunk-based Inference

During Chunk-Based Inference the requirements engineer attempts to satisfy an information requirement by using an appropriate model chunk, if any exists, to produce the information which the information requirement mandates. Model chunks are specified in full in Section 6.2.2 of the following chapter. Because a model chunk is a declarative

structure which represents an abstraction of the facts that will ultimately be integrated into the target model, chunk-based inference does not require any reasoning mechanism to be fired - unlike other requirements tasks. However, the requirements engineer may need to reify the chunk to satisfy the specific needs of the modeling context. To this end, model chunks also specify an adaptation script which is similar in form to a production script (see Section 6.2.3). This adaptation script specifies a procedure for reifying the model chunk.

Chunk-based inference involves three main steps. Firstly, the requirements engineer attempts to retrieve an appropriate model chunk by matching the source and target models, and the information requirement to candidate model chunks. Secondly, if an appropriate model chunk could be retrieved, the requirements engineer reifies this chunk to fit the particular modeling context described in working memory at that point in time. Thirdly, and finally, the result of this reification is integrated into the target model. The full procedure for chunk-based inference is given in algorithm 6.

```
Data: Context = A Modeling Context;
Success = false;
Chunk = null;
Result: Chunk is integrated into Context.TargetModel ⇒ Success = true
Context.RetrievedStructures = match(Context, ModelChunks);
if size(Context.RetrievedStructures) > 0 then
    Chunk = Choice of chunk from Context.RetrievedStructures;
    Chunk = reify(Chunk);
    Integrate(Chunk, Context.TargetModel);
    Success = true;
end
return Success;
```

Algorithm 6: The procedure for Chunk-based Inference

5.4.2 Rule-based Inference

Rule-based inference is so called because it uses sequences of production rules, known as production scripts (see Section 6.2.3 of the next chapter for a full specification). Production scripts can be seen as direct applications of heuristic classification to solve the problems specified by an information requirement. Production scripts retrieve facts from the source model and transform these in order to produce new facts which refine the target model. Because production scripts directly transform information from the

source model, they are inherently rooted in the current modeling context and so the facts which they produce do not need to be reified.

Like chunk-based inference, rule-based inference consists of three main steps. As with all requirements tasks, the first step is to match the modeling context to a set of candidate knowledge structures. In the case of rule-based inference, these knowledge structures are production scripts. Once an appropriate set of production scripts has been identified, the requirement engineer selects the script which they deem most likely to resolve the information requirement. This script is then executed, or "fired", according to the procedure described in Section 5.11.2. Briefly, this process involves retrieving a set of source facts from the source model, and applying each production rule in the script in turn to iteratively transform these source facts until all production rules in the script have been completed. The result of a production script will, therefore, be a set of transformed facts which can be integrated into the target model.

Algorithm 7 shows the complete procedure for rule-based inference.

```
Data: Context = A Modeling Context;
Success = false;
Facts = null;
Result: Facts are integrated into Context.TargetModel ⇒ Success = true
Context.RetrievedStructures = match(Context, ProductionScripts);
if size(Context.RetrievedStructures) > 0 then
    Rule = Choice of rule from Context.RetrievedStructures;
    Context.GeneratedFacts = fire(Rule, Context);
    Integrate(Facts, Context.TargetModel);
    Success = true;
end
return Success;
```

Algorithm 7: The procedure for Rule-based Inference

5.5 Elicitation

Elicitation is the only requirements task described by RORE which attempts to resolve an information requirement by interacting with external information sources. As such, it is used primarily as a last resort when all other means of resolving an information requirement have been exhausted, although it can also be used to provide a mechanism by which a requirements engineer is requested to validate portions of model. During

Elicitation, the requirements engineer retrieves structures known as "elicitation stimuli" (see Section 6.2.4). Each stimulus describes a question which has been proven to be an effective way of eliciting a given kind of information from stakeholders. The requirements engineer enacts the elicitation stimulus by interacting with relevant stakeholders to ensure that they understand the question posed by the stimulus, and to elicit from those stakeholders an appropriate response to the question. Within the RORE framework, it is assumed that a requirements engineer who is applying the approach understands the abstractions within the long-term memory knowledgebase that they are utilising. Therefore, the requirements engineer is expected to validate the appropriateness of a stakeholder's response to an elicitation stimulus. The reuqirements engineer should treat an elicitation stimulus as a prompt to themselves to indicate the kind of information which is required to refine the target model in line with the current information requirement. They are, therefore, free to use their own skill and creativity to elicit an appropriate response from the stakeholder, but this is not a formal part of the RORE process. RORE will also perform a certain degree of validation of the response during the next round of analysis by determining whether or not the current information requirement has been satisifed. If the requirements engineer determines the stakeholder's response to be appropriate, they then formalise the response into a set of facts according to a chunk structure (see Section 6.2.2) specified by the stimulus. The resulting set of facts is then integrated into the target model.

The requirements engineer follows four main steps during Elicitation. Firstly, they must retrieve an appropriate elicitation stimulus according to the usual matching procedure. Next, they enact the Elicitation Stimulus by presenting the question to relevant stakeholders and eliciting an appropriate response. Thirdly, they must formalise this response according to the chunk structure specified by the Stimulus itself. The final step is dependent on the Elicitation Stimulus itself. One of two courses of action are possible. If specified by the Elicitation Stimulus, then the resulting set of facts can be directly integrated into the target model as-is. However, this may not always be appropriate, and so an Elicitation Stimulus can also indicate that a set of facts should be used as the source facts for a round of inference. In this case, the usual inference procedure is invoked, but the facts produced by elicitation are temporarily substituted for the source model.

The procedure for Elicitation is specified in algorithm 8.

```
Data: Context = A Modeling Context;
Success = false;
Stimulus = null;
Result: Facts are integrated into Context.TargetModel ⇒ Success = true
Context.RetrievedStructures = match(Context, InferByRule);
if size(Context.RetrievedStructures) > 0 then
    Stimulus = Choice of stimulus from Context.RetrievedStructures;
    Context.GeneratedFacts = fire(Stimulus, Context);
    Integrate(Facts, Context.TargetModel);
    Success = true;
end
return Success;
```

Algorithm 8: The procedure for Elicitation

5.6 Control Procedures

The requirements tasks described by RORE are knowledge-driven procedures which operate over several knowledge structures stored in working memory. These knowledge structures persist for different periods of time and must be managed carefully in order for the RORE process to run smoothly and efficiently. To this end, three additional procedures are specified by RORE which are responsible for managing the knowledge structures within working memory. These procedures are:

- **Activity Initialisation** is performed by the requirements engineer in order to prepare working memory for the enactment of a particular activity. It is the first action which the requirements engineer carries out when they begin a new modeling activity;

- **Generation of Information Requirements** is the procedure by which information requirements are produced based on the results of a particular test during Analysis;

- **Refreshing Working Memory** involves ensuring that working memory is cleared at the end of each Analysis-Action cycle to ensure that the requirements engineer's workspace is kept clutter free.

Collectively these actions manage the resources within working memory over which each Analysis-Action cycle operates. As such, these actions are intrinsically linked to the requirements tasks described above, but are not directly relevant to producing requirements models. It is the control procedures which serve as the entry point to a

RORE session, and it is these procedures which co-ordinate resources and other activities through that session. The top-level control procedure is described in Algorithm 9.

```
Data: LTM = A RORE Long-Term Memory Definition;
Phase = A Phase defined within that LTM to enact;
Activity = An Activity associated with Phase;
SourceModel = Any Model instantiating ModelType indicated by Activity;
TargetModel = Any Model instantiating ModelType indicated by Activity;
Context = null;
Result: Model is refined
Context = initialise(LTM, Phase, Activity, SourceModel, TargetModel);
if Context == null then
    log(Context, failed);
else
    invokeRequirementsTasks(Context);
end
```

Algorithm 9: The top-level Control procedure which invokes the Analysis-Action cycle

First, the requirements engineer initialises a RORE session. This action involves deciding what modeling task needs to be performed and identifying the source and target models for that session. New models are created if necessary. Next, the controller commences an Analysis-Action loop. This loop continues until Completeness Analysis dictates that the model is complete. During this loop, each requirements task is executed in turn, with Quality Analysis deferring to the controller in order to produce Information Requirements as required. Once each requirements task has completed, the controller refreshes working memory and the loop commences for a further iteration.

5.6.1 Activity Initialisation

In RORE, activity initialisation is the process of preparing working memory for a particular activity. As noted in Section 6.5 of the next chapter, an "Activity" in RORE is a unit of behaviour which involves the refinement of a particular kind of model. Activities are characterised by both a source and a target model type, and they involve the transformation of the source model into a complete model of the target type. Three general kinds of activity can be identified:

- **Construction** activities are those which build a new target model from scratch. In this case no source model is identified;
- **Refinement** activities are those which build a new target model of the same type as a specified source model. The target model will be a new — more mature — version of the source model;
- **Transformation** activities are those which build a new target model that is of a *different* model type to the source model. This is analogous to traditional model transformation systems.

Each activity specifies the type of both the source and target models (if any), and during Activity Initialisation the requirements engineer must instantiate both the source and the target models by specifying an existing source model (for refinement or transformation activities), and either specifying an existing target model or instantiating a new (empty) target model of the given type. Table 5.1 summarises these scenarios.

Activity Type	Source Model	Target Model
Construction	-	Type_x
Refinement	Type_x	Type_x
Transformation	Type_x	Type_y

Table 5.1: Initialisation conditions for activity types

Once the desired source and target models have been specified for the RORE session, these models must then be correctly organised in working memory so that the RORE cycle can begin. In order to avoid the need for the RORE procedures to make a special case of construction, RORE assumes that both a source and a target model will be present in every case. Since, in the case of construction, no source model is specified, therefore, the requirements engineer should treat the newly instantiated target model as both the source and the target model. Thus once the target model has been instantiated, in the case of construction activities, a pointer should be created in the source model slot of working memory to the target model.

Aside from the source and target models, the requirements engineer also needs to load other aspects of the initial modeling context into working memory before the RORE cycle can begin. In particular, the requirements engineer will use knowledge about the location of the current modeling context within a broader requirements engineering process to match the modeling context to appropriate knowledge structures. Accordingly, the requirements engineer should also load both the activity knowledge

structure and its associated phase knowledge structure (see Section 6.5 of the next chapter for details) into working memory.

With these points in mind, the initialisation procedure is given below in Algorithm 10.

```
Data: LTM = A RORE Long-Term Memory Definition;
Phase = A Phase defined within that LTM to enact;
Activity = An Activity associated with Phase;
SourceModel = Any Model instantiating ModelType indicated by Activity;
TargetModel = Any Model instantiating ModelType indicated by Activity;
Context = null;
Result: Either Context is instantiated OR Context is not instantiated
if TargetModel == null then
    log(TargetModel, failed);
else
    if TargetModel.Type == Activity.TargetModelType then
        if SourceModel == null then
            if Activity.SourceModelType == Activity.TargetModelType then
                SourceModel = TargetModel ;
            else
                log(SourceModel, failed);
                return null;
            end
        end
        if SourceModel.ModelType == Activity.SourceModelType then
            Context = New Context(LTM, Phase, Activity, SourceModel,
            TargetModel);
        end
    end
    return Context;
end
```

Algorithm 10: The Procedure for Initialising a RORE Session

The requirements engineer first chooses from long-term memory a requirements engineering process which they wish to employ, and then identifies the appropriate phase and activity of that process. These knowledge structures are loaded into working memory. Next, the requirements engineer specifies a target model of the type mandated by the chosen activity. The requirements engineer may either specify a pre-existing model as the target model, in which case this model will be refined further, or they may create a new target model. In either case, the model must match the type mandated by the chosen activity. Finally, if the activity requires it, the requirements engineer must

select a pre-existing model as the source model.

5.6.2 Generating Information Requirements

Information Requirements are critical to the RORE process. They describe specifications of a specific problem with the target model and so represent a specific goal which must be satisfied during the next Analysis-Action cycle. Their structure is specified fully in the next chapter, Section 6.4.2. The requirements engineer produces a new Information Requirement during each cycle in response to the failure of the target model to satisfy a particular quality analysis rule, and it is from information about this failure that new information requirements are produced.

The version of RORE presented in this thesis, Analysis Rules contain all of the information that is needed in order to produce a new information requirement. Thus the process of generating requirements is a simple process of instantiation based on properties defined over the Analysis Rule, as shown in algorithm 11.

Data: Rule = An Analysis Rule;
IR = null;
Result: IR is instantiated
IR = new InformationRequirement();
IR.RequiredFactType = Rule.EnforcedFactType;
IR.GoalState = Rule.Antecedent;
return *IR*;

Algorithm 11: The procedure for creating an Information Requirement

5.6.3 Refreshing Working Memory

At the end of each Analysis-Action cycle, the requirements engineer should perform some basic housekeeping tasks. In particular, they will need to dump any knowledge structures produced during the cycle, so that these do not confound decisions and procedures during the forthcoming cycle. There are three kinds of knowledge structure that will need to be dumped:

1. The Information Requirement should be dumped at the end of each cycle;
2. Any knowledge structures retrieved by matching should also be dumped;
3. The temporary store containing facts produced by either Inference or Elicitation during the previous cycle will need to be dumped.

As with Information Requirement generation, this is a simple but critical procedure, and it is described by algorithm 12.

Data: Context = A Modeling Context
Result: All temporary data is cleared
Context.InformationRequirement = null;
Context.RetrievedStructures = null;
Context.GeneratedFacts = null;
Algorithm 12: The procedure for flushing working memory at the end of each cycle

5.7 Reuse Procedures

The requirements tasks described in Section 5.2 are each driven by the reuse of knowledge structures from long-term memory. As Section 3.3.3 noted, the reuse model of RORE is heavily influenced by Clancey's heuristic classification, and as such each requirements task broadly consists of three main activities:

- **Matching**, which relates the specific details of the current modeling context to indexes in long-term memory which describe generalised statements of problems and point to potential solutions as described by a range of reusable knowledge structures (rules, model chunks, production scripts and elicitation stimuli) (see Section 5.8);

- **Firing or Reification** during which the requirements engineer performs the task mandated by a procedural knowledge structure, or else reifies a declarative knowledge structure. Both produce a set of facts which fits the specific context of the source and target models (see Section 5.11 for firing procedures, or Section 5.9 for reification procedures);

- **Integration**, where the requirements engineer integrates the facts produced by the current cycle into the target model in order to refine the model, and satisfy the Information Requirement (see Section 5.10).

Each of the major steps of a given requirements task or sub-task invokes one or other of these reuse procedures, as specified in the requirements task procedures. As a result, there is no separate top-level procedure for invoking these reuse procedures; invocation of a reuse procedure is directed entirely by the needs of a particular requirements task.

5.8 Matching Modeling Contexts to Knowledge Structures

The requirements tasks on which the RORE approach is founded are individually driven by reuse. They apply knowledge structures, which are retrieved from RORE's long-term memory, to analyse the quality of models and to refine models according to the results of those analyses. The ability to find and retrieve knowledge structures in long-term memory that are directly relevant to a given modeling context and which will satisfy a given information requirement is critical to the RORE approach. In general, the problem of component retrieval is a long-standing problem for reuse research [Kru92, FK05], and this thesis aims to show that the RORE approach uses a mix of strategies for library organisation and component retrieval which support a practical approach to systematic reuse.

The matching process described in this section has been directly influenced by the Domain Theory's AIR tool [Sut02] in two ways:

- Firstly, like AIR, RORE's matching approach uses procedures which are inexpensive with respect to effort rapidly to narrow down the pool of candidate knowledge structures before employing more time-consuming matching procedures which produce more accurate matches;

- Secondly, like AIR, RORE's matching approach uses both rule-based matching and analogical (see [Gen83, Mai92, MS96]) matching algorithms to identify suitable knowledge structures.

There are also important differences between the two approaches, however.

Firstly, AIR's matching procedure matches directly against the knowledge structures which make up its library, whereas in RORE knowledge structures are each tagged with meta-data against which the match is made. The specific structure of these meta-data indexes is discussed in Section 6.2.5 of the next chapter.

Secondly, AIR organises its reusable knowledge structures into specialisation hierarchies. The analogical matcher is used to match concrete models to general abstractions, while rule-based matching is then used to specialise that match. By contrast, in RORE's long-term memory, no such specialisation hierarchy exists. Instead, the meta-data associated with each structure relates the knowledge structure to the behavioural units (phases and activities), and model types, to which that knowledge structure is

applicable. Furthermore, RORE allows the knowledge base administrator arbitrarily to attach conditions — which may be formal rules or model chunks — to a meta-data index which characterise more precisely the scenario to which the structure should be applied. Each index has pre-, post- and trigger conditions which describe the transformation that the associated knowledge structure represents. All of this meta-data provides a rich source of information to support accurate matching.

The overall structure of the matching procedure is described by algorithm 13, and Sections 5.8.2, refsec:chunkmatching and 5.8.4 drill down into the details of the particular matching procedures which make up this procedure.

5.8.1 Efficient Information Gain Means Efficient Matching

Note that the order in which matching procedures are performed by the requirements engineer is dictated strictly and solely by the principle of efficient information gain which Chapter 3 outlined. Another principle — the quality of a match between a candidate reusable knowledge structure and a target model — is also important. In order to ensure the quality of the target model which is output at the end of a RORE session, it is important also to ensure that the most appropriate knowledge structures (with respect to the information requirement and target model) are retrieved to support transformation of the target model in each Analysis-Action cycle. RORE does indeed take the quality of a match into account and it is for this reason that analogical matching — which has been shown by Sutcliffe and Maiden to be a highly effective means of producing good quality matches between an abstraction and a concrete model (see, e.g., **??** and **??**) — is included as a part of the matching process. However, accurate matching procedures — such as Analogical Matching — are computationally expensive**??** and so the author introduced additional matching procedures to the overall matching process, which are less computationally expensive and which can significantly reduce the number of candidate structures to which analogical matching is applied. Matching procedures which are computationally inexpensive are thus utilied in RORE to filter out as many false positives from the candidate pool as possible, before analogical matching is applied to determine which reusable knowledge structures are highly probable to be appropriate in the sense of:

1. Satisfying the current information requirement;

2. Doing so within the context of the current target model.

```
Data: LTM = A Long-Term Memory definition;
Context = A Modeling Context;
StructureTypeNeeded = Any from AnalysisRule, ModelChunk,
ProductionScript, ElicitationStimulus;
Structures = All ReusableStructures in LTM;
Structure
Result: Structures contains only structures that are relevant to Context
Structures = Filter(Structures, StructureTypeNeeded);
while Structure = pop(Structures) do
    while Condition = pop(Structure.Index.Preconditions) do
        if Condition is a ChunkCondition then
            if not analogicalMatch(Condition, Context.SourceModel) then
                Structures = Remove(Structure, Structures);
            end
        else
            if not ruleMatch(Condition, Context.TargetModel) then
                Structures = Remove(Structure, Structures);
            end
        end
    end
    while Condition = pop(Structure.Index.TriggerConditions) do
        if Condition is a ChunkCondition then
            if not analogicalMatch(Condition, Context.TargetModel) then
                Structures = Remove(Structure, Structures);
            end
        else
            if not ruleMatch(Condition, Context.TargetModel) then
                Structures = Remove(Structure, Structures);
            end
        end
    end
    while Condition = pop(Structure.Index.PostConditions) do
        if Condition is a ChunkCondition then
            if not analogicalMatch(Condition,
            Context.InformationRequirement.GoalState) then
                Structures = Remove(Structure, Structures);
            end
        else
            if not ruleMatch(Condition,
            Context.InformationRequirement.GoalState) then
                Structures = Remove(Structure, Structures);
            end
        end
    end
end
return Structures;
```

Algorithm 13: The top-level matching algorithm

The matching procedures within RORE are thus applied as a composite and are not treated as alternatives, with the composite being designed to ensure a high quality match, and most of the parts being designed only to filter candidates out of the matching process according to different criteria. Thus, while quality was a design consideration for the composite as a whole, efficiency is the primary concern in sequencing the different procedures within that composite. A further consideration is that all of the matching procedures within RORE depend fundamentally on meta-data about reusable knowledge structures which is specified in the design of a long-term memory knowledgebase by the knowledge engineer, and so the quality of the matching procedures can only be as good as the quality of that meta-data.

The principle of Efficient Information Gain, then, was the primary consideration in sequencing the RORE matching procedure. This principle has a particular application to the matching process, because it is the matching process through which information in RORE is predominantly "gained" by the requirements engineer. In order to perform a match, the requirements engineer must specify some input parameters against which candidate structures can be matched. A matching procedure is considered "efficient" if it is able to significantly reduce the size of the pool of candidate knowledge structures based on a small amount of input information. That is, the efficiency of a matching algorithm could be predicted using the Information Gain ratio:

$$Information_{In} : Information_{Out} \tag{5.1}$$

According to this definition, procedures which match against Phases and Activities will be efficient because they rule out all knowledge structures which are not directly relevant to the modeling activity at hand based on a small amount of input information. By contrast, the chunk-based analogical algorithms require at least something of a concrete model to have been specified before it can operate effectively. As such, rule- and chunk-based algorithms are less efficient - because they gain less information based on more input information - than selection based on behavioural units. However, there is a trade-off because simply selecting for a behavioural unit offers a limited guarantee that retrieved knowledge structures will be relevant to the particulars of the current modeling context. It is for this reason, that matching procedures in RORE are ordered so that the most efficient procedures are used to rapidly reduce the size of the candidate pool, and then more accurate — but less efficient — procedures are used to refine the match.

It should be noted that this approach follows the basic approach adopted by Maiden [Mai92], who uses rule-based matching efficiently to identify a pool of candidate abstractions that are likely to match a concrete model before using analogical matching to identify the abstraction which best fits that model. Maiden found this approach to produce accurate, high-quality matches between a concrete model and a domain abstraction. Maiden also validated his automated, analogical approach against matches made by human experts [MMS95]. However, there is a critical difference between Maiden's approach and the approach presented in this thesis, which impacts on the potential quality of a match. Whereas Maiden's approach comes with a built-in set of abstractions and matching rules — all of which have been carefully and empirically validated my Maiden — the RORE approach does not preassume either the abstractions of the matching rules. Instead, these are stipulated within the KE perspective by knowledge engineers. The task of validating both the abstractions and the matching rules therefore falls on the knowledge engineers who have responsibility for designing a long-term memory knowledgebase.

5.8.2 Filtering Candidate Structures

The first procedure in the matching approach is designed to identify, with the minimal possible effort, all of those knowledge structures for which there exists a reasonable likelihood of a relevance to the current modeling context. This is achieved by incrementally narrowing the set of candidate knowledge structures. Initially, all reusable knowledge structures in long-term memory are treated as plausible candidates and the requirements engineer narrows this pool by identifying those criteria which satisfy five criteria. In order, each candidate knowledge structure should match:

- The **Structure Type** (Analysis Rule, Model Chunk, Production Script or Elicitation Stimulus) specified by the requirements engineer as being relevant to the current requirements task;
- The **Phase** associated with the current modeling activity;
- The **Activity** which the current RORE session is enacting;
- The **Desired Fact Type** which the current Information Requirement mandates should be processed by this Analysis-Action cycle.

This procedure is summarised by Algorithm 14.

```
Data: LTM = A Long-Term Memory definition;
Context = A Modeling Context;
StructureTypeNeeded = Any from AnalysisRule, ModelChunk,
ProductionScript, ElicitationStimulus;
Structures = All ReusableStructures in LTM;
Structure = null
Result: Structures contains only those relevant by Type, Phase, Activity and
        FactType
while Structure = pop(Structures) do
    if Structure.Type != StructureTypeNeeded then
        Structures = Remove(Structure, Structures);
    else
        if Structure.Index.Phase != Context.Phase then
            Structures = Remove(Structure, Structures);
        else
            if Structure.Index.Activity != Context.Activity then
                Structures = Remove(Structure, Structures);
            else
                if Structure.Index.FactType !=
                Context.InformationRequirement.RequiredFactType then
                    Structures = Remove(Structure, Structures);
                end
            end
        end
    end
end
return Structures;
```

Algorithm 14: The Filtering Procedure which is part of the overall Matching Procedure

Each of the filtering tests applied by this initial matching procedure requires only that a direct comparison be made between a single property of knowledge structures in working memory and a single property of the index associated with each candidate knowledge structure. The last, cycle goal match, criterion is less trivial, however. This criterion tests to see whether or not the post-condition specified by the index of each knowledge structure would satisfy the goal specified by the Information Requirement. The criterion therefore requires that the requirements engineer evaluate whether or not one condition would hold if another condition would also be true. Formally:

$$(P(S) \Rightarrow G(C)) \Rightarrow GoalMatch(S,C) \tag{5.2}$$

Where $P(S)$ is the postcondition associated with the structure currently being evaluated, $G(C)$ is the goal of the current modeling context, and $GoalMatch(S,C)$ is a predicate asserting that S matches C according to the cycle goal criterion.

Current versions of OWL (Lite, DL and Full) and their associated rule language (SWRL), however, only allow conditions to be tested against sets of OWL axioms (facts in RORE's terminology), and not against other conditions. This thesis does not, therefore, offer detailed specifications on how this criterion should be evaluated (investigating a general approach to this problem would have been a significant effort in its own right) and so the requirements engineer must use their discretion to evaluate this criterion.

5.8.3 Chunk-based (Analogical) Matching

We have adopted analogical matching in RORE because it has tried-and-proven applicability for the reuse of knowledge structures in requirements engineering specifically [Fin88, Mai92, MVL97]. In particular, analogical reasoning is attractive as a means of retrieving knowledge structures in a reuse context because it provides a means of applying knowledge structures beyond the scope of the domains from which they were originally abstracted [Sut02]. As such, analogical reuse is one of the primary strategies adopted by the Domain Theory to achieve a greater degree of generality than the prevalent domain-specific approaches [SM98], and which RORE employs as one strategy for achieving this same goal.

Analogical matching algorithms have been influenced by the work, in Cognitive

Science, of Gentner [Gen83, GM97] and others [Mai92] who have attempted to understand the cognitive processes which underpin the formation of analogies in the human mind. According to Gentner's theory, analogical reasoning involves transferring knowledge from a source domain to a target domain based on *structural* comparisons between the two domains. The particular entities from which the domain is composed are not considered significant in an analogical match, but rather it is structural relationships between those entities that are considered important. Gentner and Markman contrast this with similarity-based reasoning in which the entities, rather than structure, are considered important [GM97]. According to this structure-mapping theory, the atom is analogous to a solar system because both share a similar structure: that of small bodies orbiting a larger, central body.

The procedure which is described below for analogical matching essentially replicates the original structure mapping algorithm described extensively by Maiden [Mai92] and Maiden and Sutcliffe [MS96]. However, it was necessary to adapt the algorithm to take into account two significant factors:

- Firstly, the analogical algorithm described by Maiden and Sutcliffe [Mai92, Sut02] assumes their specific schema for representing domain knowledge. Within this schema, fact types are ordered according to the impact that each fact type has on an analogical match. For instance, within the Domain Theory's modeling schema, matches between state transitions in the source and target domain are considered to be more indicative of an analogical match than matches between objects and their so-called secondary states [SM98]. Thus the Domain Theory's analogical procedure focuses on matches between these higher-ranking fact types and uses rule-based matching to refine the match according to lower-ranking fact types [Mai92]. However, this is a decision taken predominantly to ensure efficient matching by avoiding the need to check matches between all facts in every model [MS96]. Because RORE does not assume a specific modeling schema, the analogical procedure presented in this thesis cannot rely on such heuristics, and so instead it uniformly checks structural relations between facts of any kind. This thesis assumes this will impact on the efficiency of the procedure presented by this thesis, but predicts that it will not impact on effectiveness. However, this prediction is not directly relevant to the research questions laid out in Chapter 1 and so this thesis does not extensively validate the prediction. This independence from any particular model schema also means that, in contrast to

AIR's model-specific matching algorithm, RORE's matching procedure is capable of matching across model types. Chapter 9 considers the extent to which this prediction holds.

- Secondly, the implementations described by Maiden [Mai92] and Sutcliffe [Sut02] were based respectively on Prolog and ConceptBase. By contrast, the formalisation of RORE is based on the OWL knowledge representation language. The procedures which previous authors describe, therefore, needed to be modified to suit this new choice of formalism. The open-world assumption made by OWL is a significant change from the closed-world assumption which underpins ConceptBase and this fact needed to be taken into account.

In RORE's analogical matching procedure, model chunks specified as conditions in the index for a knowledge structure are matched against either the source or target model as specified. Algorithm 15 presents the top-level procedure for analogical matching:

The algorithm for determining whether a candidate satisfies the matching threshold is given in Algorithm 16.

The top-level algorithm checks for a match between a Model and a ChunkCondition in two stages. Firstly, it enacts a cheap algorithm to identify "Local Mappings" between fact pairs (one from the model, one from the chunk condition) in order to determine whether it is possible that a match exists. It then looks for a structural match between facts in order to confirm whether or not the condition matches the chunk analogically.

Local Mapping compares all facts from the Model against all facts from the Condition Chunk, and identifies a Local Map if the two facts match semantically (see Algorithm 17):

Identifying the Local Mappings which exist between a Chunk and a Model involves determining whether two facts match semantically. This is so if they both instantiate a shared Fact Type (see Algorithm 18):

If Local Mapping determines that there is the possibility of a match between the Condition and the Model, then a full structural match is checked for. A structural match is considered to exist if 70% of locally-mapped pairs are confirmed as structural matches (see Algorithm 19). The 70% threshold is replicated from Maiden's original analogical algorithm, which Maiden determined through empirical validation to fine-tune his own approach:

```
Data: Condition = A Chunk Condition;
Model = A Model;
ChunkFacts = {};
ModelFacts = {};
PossiblePairs = {};
ConfirmedPairs = {};
MatchingThreshold = 70;
SatisfiesThreshold = false;
Result: True, if a match exists
ChunkFacts = All Facts in Condition.Chunk;
ModelFacts = All Facts in Model;
PossiblePairs = getLocalMapping(ChunkFacts, ModelFacts);
if size(PossiblePairs) > 0 then
    ConfirmedPairs = getStructuralMappings(ConfirmedPairs);
    if ) then SatisfiesThreshold(ConfirmedPairs, PossiblePairs,
    MatchingThreshold
        return true;
    else
        return false;
    end
else
    return false;
end
```

Algorithm 15: The top-level matching procedure first checks for local mappings, and if any exist, performs a full structure match

```
Data: ConfirmedPairs = A set of fact pairs from a Model;
PossiblePairs = A set of fact pairs from a Model;
MatchingThreshold = A numerical value;
Satisfies = false;
Result: True, if the number of confirmed pairs as a percentage of possible pairs
        satisfies the threshold
Satisfied = calculateMatchScore(ConfirmedPairs, PossiblePairs) ⪰
MatchingThreshold ;
return Satisfied;
```

Algorithm 16: The procedure for determining whether or not the matching threshold is satisfied by a candidate knowledge structure

```
Data: ModelFacts = A set of facts from a Model;
ModelFact = null;
ChunkFacts = A set of facts from a Condition Chunk;
ChunkFact = null;
PossiblePairs = {};
Result: PossiblePairs is either empty or contains pairs of facts
while ModelFact = pop(ModelFacts) do
    while ChunkFact = pop(ChunkFacts) do
        if factsMatchSemantically(ModelFact, ChunkFact) then
            add(PossiblePairs, ChunkFact, ModelFact);
        end
    end
end
```

Algorithm 17: Local Mapping is a cheap procedure for ruling out cases where analogical matches could not exist

```
Data: ModelFact = A fact extracted from a Model;
ModelFactType = null;
ChunkFact = A fact extracted from a Condition Chunk;
ChunkFactType = null;
Result: True if the two facts share a parent fact type;
while ChunkFactType = pop(ChunkFact.instantiatedTypes) do
    while ModelFactType = pop(ChunkFact.instantiatedTypes) do
        if ChunkFactType == ModelFactType then
            return true;
        end
    end
end
return false;
```

Algorithm 18: A procedure for checking whether or not two facts match semantically

```
Data: PossibleFactPairs = A set of possible fact pairs, established by Local
      Mapping;
PossibleFactPair = null;
ConfirmedFactPairs = {};
MatchingThreshold = 70;
Result: True, if a structural match exists
if size(PossibleFactPairs) == 0 then
  | return false ;
end
while PossibleFactPair = pop(PossibleFactPairs) do
  | if scoreFactPair(PossibleFactPair) ¿= MatchingThreshold then
  |   | add(ConfirmedFactPairs, PossibleFactPair);
  | end
end
return ((size(PossibleFactPairs) / (sizePossibleFactPairs)) * 100) ¿ 70;
```

Algorithm 19: Structural matching checks the structural relationship that exists between each fact pair

The score for a fact pair is calculated by evaluating the proportion of mappings that can be established between the neighbouring facts of the two facts in the fact pair (see Algorithm 20):

The ability to calculate the score for a fact pair is dependent on the ability to determine whether two binary relationships are semantic matches. This is true if the two relations share the same property type (see Algorithm 21):

5.8.4 Rule-based Matching

Whereas chunk-based matching uses model chunks to evaluate the applicability of a knowledge structure to a given modeling context, rule-based matching uses conditional tests similar to those specified by analysis rules. The rule-based procedure is not designed as an alternative to, but rather complements, the chunk-based matching procedure. As Sutcliffe and Maiden note [SM98], rules are useful for fine-grained matching, whereas chunks are useful for coarser-grained matching of overall structure.

Rules, which may be arbitrarily specified as pre-, post- or trigger conditions in a meta-data index, are tested against either the source or target model. The result will be a Boolean (true or false) value indicating the applicability, or not, of the knowledge structure to the current modeling context. The procedure is significantly more straightforward, but nonetheless significant for RORE's matching procedure, than the chunk-based procedure.

```
Data: ModelFact = Any fact from a Model;
ModelProperty = null;
ModelChildFact = null;
ChunkFact = Any fact from a Condition Chunk;
ChunkProperty = null;
ChunkChildFact = null;
ChunkPropertyTuple = {};
ModelPropertyTuple = {};
PossibleNeighbours = 0;
ActualNeighbours = 0;
Result: ActualNeighbours / PossibleNeighbours * 100
while ChunkProperty = pop(ChunkFact.Properties) do
    while ChunkChildFact = pop(ChunkProperty.Values) do
        while ModelProperty = pop(ModelFact.Properties) do
            while ModelChildFact = pop(ModelProperty.Values) do
                possibleNeighbours += 1;
                ChunkPropertyTuple = {ChunkFact, ChunkProperty,
                ChunkChildFact};
                ModelPropertyTuple = {ModelFact, ModelProperty,
                ModelChildFact};
                if propertiesMatchSemantically(ChunkPropertyTuple,
                ModelPropertyTuple) then
                    actualNeighbours += 1;
                end
            end
        end
    end
end
return (actualNeighbours / possibleNeighbours) * 100
```

Algorithm 20: Fact pairs are scored for their structural match by determining how many structurally-matching properties the chunk fact shares with the model fact

```
Data: ChunkPropertyTuple = A tuple containing the parent, property and child
      from a chunk;
ModelPropertyTuple = A tuple containing the parent, property and child from a
model;
ParentsMatch = null;
ChildrenMatch = null;
ChunkPropertyType = null;
ModelPropertyType = null;
Match = true;
ShareType = false;
Result: True, if the type of the parent and child match across the two property
        tuples
while ChunkPropertyType = pop(ChunkPropertyTuple.Property.PropertyTypes)
do
    ShareType = false;
    while ModelPropertyType =
    pop(ModelPropertyTuple.Property.PropertyTypes) do
        if ChunkPropertyType == ModelPropertyType then
            ShareType = true;
        end
    end
    if not ShareType then
        Match = false;
    end
end
ParentsMatch = factsMatchSemantically(ModelPropertyTuple.Parent,
ChunkPropertyTuple.Parent);
ChildrenMatch = factsMatchSemantically(ModelPropertyTuple.Child,
ChunkPropertyTuple.Child);
return ParentsMatch and Match and ChildrenMatch;
```

Algorithm 21: A Procedure for Checking the Semantic Match Between Two Properties

Essentially, the requirements engineer applies rules for a given index, and uses one of the firing procedures identified in Section 5.11.1 to test each rule against the appropriate model. If all rules are satisfied by the modeling context, then the overall set of conditions returns true.

Data: Rule = Any analysis rule;
Model = A model over which to test the AnalysisRule;
return *satisfiesRule(Model, Rule)*;

5.9 Reifying Solutions

The model chunks which are used by chunk-based inference to produce new facts tend to be abstract and therefore need to be reified. To achieve this, the requirements engineer must reify the chunks by modifying key elements of the chunk using information acquired from the source model. RORE uses a reification approach based on the concept of a requirements template [LMV97]. Facts within a model chunk can be designated - by the use of special tags in their names (see Section 6.2.2) - as being abstract labels. The requirements engineer can then employ two strategies to reify the chunk by substituting concrete labels for abstract labels.

Firstly, the requirements engineer can use their own discretion to decide which information from the source model should be substituted for each abstract label. RORE specifies no formal procedure for this approach. Secondly, if an adaptation script is specified by the model chunk, then the requirements engineer should follow this script to achieve the necessary substitutions. The adaptation script should retrieve facts from the source model and transform these to produce the concrete labels mandated by the model chunk.

The procedure by which a requirements engineer applies an adaptation script is in essence identical to that required to enact a production script (see Section 5.11.2). There is an important difference, however, which is that the results of the script are integrated into the model chunk and not into the target model itself. It is the reified model chunk which is then integrated.

5.10 Integrating Reified Facts

Integration is the final step of the productive requirements tasks (Inference and Elicitation). During this step, the requirements engineer attaches the new facts produced by the requirements task to the current target model. This changes the state of the target model and allows the model to be reassessed to determine the need for a further round of refinement.

The basic procedure for integrating facts into a model is to map new facts to existing facts within the target model, and then to attach properties of the new facts to the associated existing facts. Facts which are not part of a graph formed by a fact pairing are simply appended to the model as "free" facts. RORE distinguishes two specific procedures for achieving this goal. These procedures are distinguished by the manner in which they treat the paired new and existing facts. The procedure that is used is to be specified by the index of the knowledge structure by which the new facts were produced.

5.10.1 Additive Integration

Additive Integration is a procedure which, as the name suggests, appends the new facts to the existing target model so that all information in the initial state of the target model is preserved after the integration procedure is complete. In this case, the existing properties of each existing paired fact are preserved, and the requirements engineer adds the properties of the matched fact are *added* to the existing fact. The procedure is described by algorithm 22.

5.10.2 Substitutive Integration

Substitutive Integration is an alternative form of integration which allows for existing facts to be superseded by new facts. In this case, the information represented by the old fact is substituted by the information represented by the new fact. For each existing fact that references an existing matched fact, the requirements engineer removes the reference to the existing fact and replaces it with a reference to the associated new fact. The requirements engineer performs this procedure for all matched new facts. Algorithm 23 describes this procedure.

```
Data: TargetModel = A model, into which the source facts are to be integrated;
FactPairs = A set of fact pairs which the requirements engineer has specified;
FactPair = null;
Property = null;
PropertyValue = null;
SourceFact = null;
TargetFact = null;
Result: All child facts of paired facts are copied from the source to the target
while FactPair = pop(FactPairs) do
    SourceFact = FactPair.Source;
    TargetFact = FactPair.Target;
    while Property = pop(SourceFact.Properties) do
        while PropertyValue = pop(Property.Values) do
            add(TargetFact.Property[Property].Values, PropertyValue);
        end
    end
end
```

Algorithm 22: The procedure for additive integration, assuming a set of fact pairs

```
Data: TargetModel = A model, into which the source facts are to be integrated;
FactPairs = A set of fact pairs which the requirements engineer has specified;
FactPair = null;
SourceFact = null;
TargetFact = null;
Result: All target facts are replaced by source facts in target model
while FactPair = pop(FactPairs) do
    SourceFact = FactPair.Source;
    TargetFact = FactPair.Target;
    replace(TargetModel, TargetFact, SourceFact);
end
```

Algorithm 23: The procedure for substitutive integration, assuming a set of fact pairs

5.11 Reasoning Procedures

In RORE, analytical and productive knowledge is represented by a large body of logical rules. In order for RORE to be effective, some mechanism is required to enact or "fire" these rules. One option is for the requirements engineer to enact these rules manually. In the case of conditional rules (Analysis Rules, matching criteria) this can be done manually by treating the rule as a logical theorem and attempting to prove or disprove it from the relevant model. Production rules can similarly be enacted manually by treating them as operations which replace an input set of facts by an output set of facts.

Manual enactment of RORE rules, however, would be time consuming and, given the number of rules that would need to be enacted in a RORE session to build a model, would not be practicable. The main advantage of using an existing knowledge formalism, such as OWL, to formalise knowledge in RORE is that there exists a wide range of reasoning support which a requirements engineer can use to automatically enact rules. Assuming that a mechanism exists for firing individual conditions and productions, therefore, this section describes the higher-level procedures which are required directly by the requirements tasks for firing analysis rules, matching criteria, and production scripts.

5.11.1 Firing Analytical Rules

In the current version of RORE, there is no essential difference between an Analysis Rule and a matching criterion. However, each plays a different role. Analysis rules are reusable structures which are utilised during analysis to evaluate the completeness or quality of a model. Matching criteria are not considered reusable, because they do not directly play a role in any requirements task, and instead are used to evaluate a model for the purpose of matching a reusable knowledge structure to it.

Analytical rules consist of two basic components (see Section 6.2.1 of the next chapter for details). The antecedent describes the condition which must be tested, and the consequent describes the value (true or false) which is to be returned if this condition is positive. Two steps are, therefore, needed to fire either a matching condition or an analysis rule, as described by Algorithm 24.

Firstly, the antecedent - which is represented by a SPARLDL Ask query - is executed over the desired source or target model. The result of this query is then evaluated in order to determine whether the consequent, or its negation, should be returned as the value of the rule.

```
Data: Rule = Any analysis rule;
Model = Any model;
AskQueryHolds = false;
Result: AnalysisRule.Consequent if AnalysisRule holds; else not Consequent
AskQueryHolds = sparqlDlASK(Rule.Antecedent, Model);
if AskQueryHolds then
    return Rule.Consequent;
else
    return not Rule.Consequent;
end
```

Algorithm 24: Procedure for firing an analysis rule

5.11.2 Firing Production Scripts

Production scripts are described in detail in Section 6.2.3 of the following chapter. Structurally they consist of an input query and a linked list of production rules. The input query is used to retrieve the facts from the source model over which the production script will operate. The input query is a SPARLDL select query which retrieve axioms, rather than returning a Boolean value. The requirements engineer performs this query either manually or using an appropriate reasoner, and places the results into the temporary fact slot in working memory.

The requirements engineer then retrieves the first production rule from the production script and iterates over each subsequent rule in turn, applying it to the facts that currently reside in the assigned temporary slot. Each production rule is similar in structure to an Analysis Rule, consisting of an antecedent and a consequent. In the case of a production rule, the antecedent is a further SPARQLDL select query which is performed on the selected source facts to select the specific facts which that production rule will transform. The consequent specifies the production rule using a proprietary language which was designed as part of this thesis specifically for representing RORE productions. The language is described fully in Grammar 6.2.3. It consists predominantly of two kinds of operations - fact creation and assignments - which can be used to instantiate new facts based on source facts and previous productions. It also includes an iterative operation which allows one-to-one mapping between source and target facts. To execute a production rule, the requirements engineer performs the query described by the antecedent over the temporary slot in working memory. The production rule is then applied to the retrieved facts in order to generate new facts which can be integrated into the target model.

This full procedure for executing production scripts is described in algorithm 25.

```
Data: Script = Any production script;
Model = Any model;
InputFacts = null;
TransformFacts = null;
CurrentRule = null;
OutputFacts = null;
Result: A set of facts
InputFacts = sparqlDLSELECT(Script.InputQuery, Model);
while CurrentRule = pop(Script.NextRule) do
    TransformFacts = sparlDLSELECT(CurrentRule.Antecedent, InputFacts);
    OutputFacts = applyProductionExpression(TransformFacts);
    add(InputFacts, TransformFacts);
end
```

Algorithm 25: The procedure for firing production scripts over a model

5.12 Summary

This chapter has described in detail each of the high-level procedures, and the lower-level algorithms, which underpins RORE and the prototype implementation of the tool. At the highest level, RORE defines a set of five procedures to support Requirements Engineering corresponding to the Requirements Engineering Tasks described in Chapters 3 and 5: Analysis, Inference and Elicitation. These procedures are broadly similar in structure in that they: retrieve reusable knowledge structures from long-term memory; apply those structures in order to make inferences about and reason over requirements models; and, in the case of Inference and Elicitation, integrate the product of that reasoning into the target model.

Each of these three generic steps (Matching, Reasoning and Integration) is defined more precisely by an underlying set of algorithms. Matching comprises, in the first instance, a filtering process, in which candidate knowledge structures are ruled out by comparing the Index Description of those reusable structures to the current Modeling Context. This is followed by a combination of two specialised types of positive matching algorithm. Analogical matching compares the structure of Chunk-based Conditions to either the source or target model in order to determine a match between a reusable knowledge structure and a Modeling Context. Rule-based matching fires rules, which are specified through Rule-based Conditions, over either the source or the target model in order to determine a match.

Different procedures are defined for reasoning over models during the second step of each Requirements Engineering Task. This is a result of the fact that different tasks

utilise knowledge structures which express inferential knowledge in different ways. Analysis Rules are evaluated by a SPARQL-DL query engine (as discussed in Chapter 4 which represents a rule-firing, or query resolution, mechanism. Production Scripts are executed through a parse-and-execute interpreter process. Detailed reasoning procedures are not defined for Chunk-based Inference or Elicitation because the reification of a Model Chunk in the first case, and production of information as a response to a stimulus in the latter case, are manual processes.

This chapter has also presented two types of Integration Procedure. While these two types of Integration Procedure are highly similar in structure, they differ in the crucial mechanism by which they integrate new information into the target model. Both Additive and Substitutive Integration match new Facts as produced by Inference or Analysis to facts in the target model, and then traverse the network which branches out from the new Facts in order to transfer new information into the target model. The difference is that Additive Integration appends the new Facts to the existing Facts in the target model, whereas Substitutive Integration replaces the Facts in the target model by the new Facts.

Chapter 6

The Knowledge Structures of RORE

6.1 Introduction

The procedures described in the previous chapter assume a set of knowledge structures which guide the approach, and from which requirements artefacts can be constructed. Figure 6.1 presents an overview of the knowledge structures specified by RORE which formalise the knowledge over which the procedures described in the previous chapter operate.

RORE is a framework for building models of application domains, their requirements and specifications of software systems to satisfy those requirements. The central workpiece in a RORE session, therefore, is the Model (see Section 6.4.1). In order to facilitate unambiguous and precise definition of the procedures discussed in the previous chapter, this thesis should also provide a precise syntax through which Models can be expressed.

However, a key design goal of this framework is to provide systematic support for requirements-level reuse independent of any specific requirements modeling framework. Therefore, whereas many other approaches to requirements-level reuse assume a particular language through which requirements models will be expressed, RORE does not assume any specific schema for representing domain or requirements knowledge. Instead, RORE can support multiple model types, the meta-models for which are expressed formally (see Section 6.3) as a part of RORE's long-term memory. RORE is, therefore, able to reason about the definition of a model type in order to interpret the information contained within a concrete model, rather than assuming this knowledge a priori.

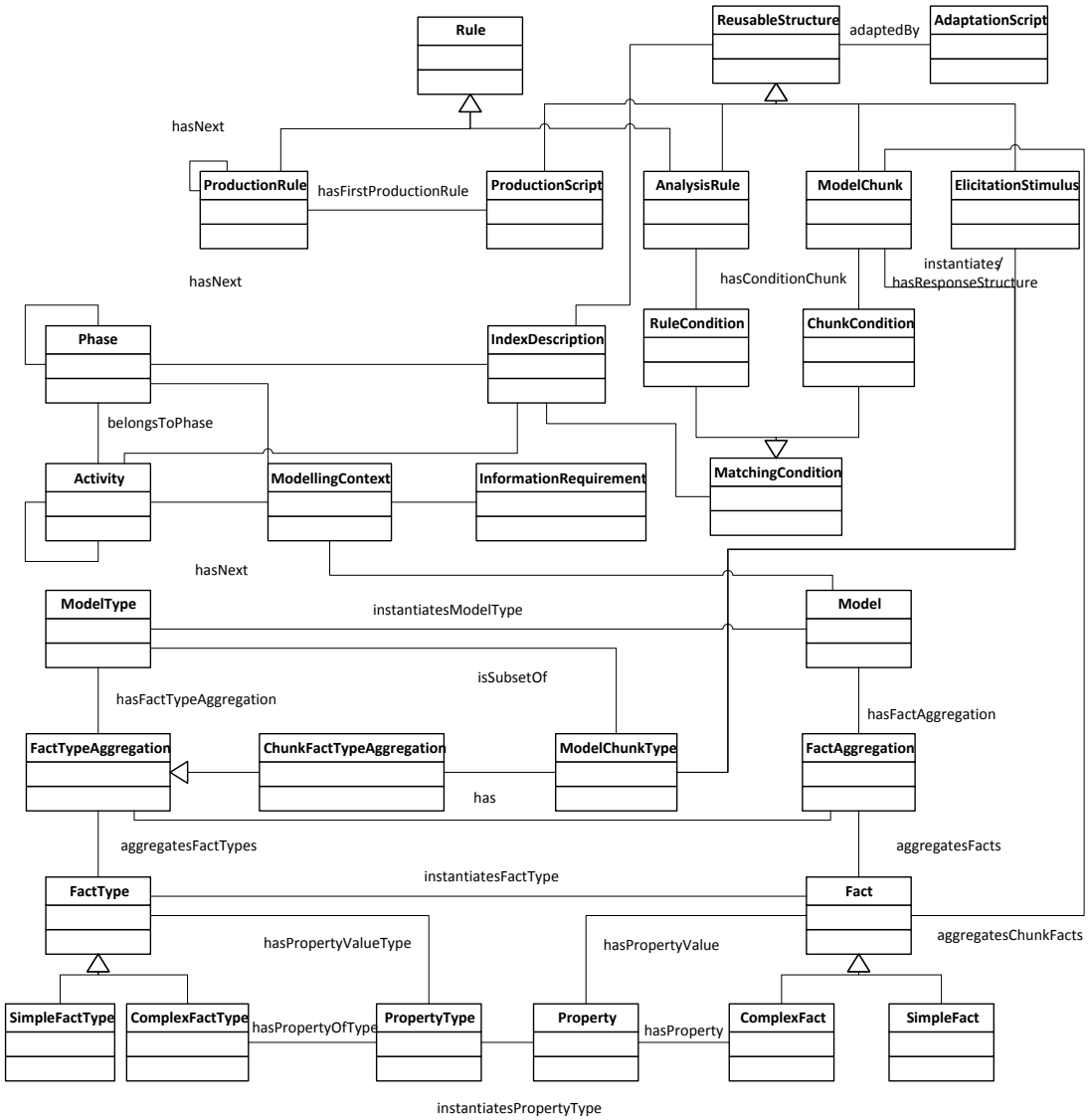

Figure 6.1: Detailed overview of the knowledge structures defined in RORE's immutable layer

This aspect of RORE allows knowledge base administrators to introduce their preferred requirements modeling representation into the RORE framework by aggregating facts types which have already been defined in RORE's long-term memory. Once a model type has been defined, the RORE requirements tasks can be applied to construct instances of that model type by reuse. Models instantiate model types in the sense that the model is an aggregation of facts, such that each fact instantiates a fact type from which the model type is aggregated.

The ability of RORE to construct a model for a new model type is dependent on the availability of reusable knowledge structures within the knowledge library. Reusable knowledge structures in RORE are not, however, confined to a single type of model and instead it is possible in RORE to transfer reusable knowledge between modeling representations. Reusable structures are defined in terms of types of fact, rather than in terms of types of model. A reusable structure therefore is applicable to any kind of model which is aggregated from the fact types over which that structure is defined. While there are differences between the major requirements modeling languages, there is also much overlap with respect to the kinds of fact which they each contain (see Chapter 2). As a result, this thesis claims that it should not be necessary to populate RORE's reuse library entirely from scratch for every new type of model that is introduced.

This chapter describes and specifies the structure of RORE's two knowledge-bases: long-term memory and working memory. This chapter drills down into the details of each knowledge base, specifying the abstract syntax of each of the knowledge structures which make up RORE's knowledge model. A full OWL-DL formalisation is given in Appendix A.

6.2 Reusable Knowledge Structures

Reusable knowledge structures in RORE are defined by knowledge engineers in order to support the adaptation, by parameterisation [Sut02], of RORE's Requirements Engineering Tasks to the needs of a particular Action-Analysis cycle. RORE defines four types of reusable knowledge structure. These are: Analysis Rules, Model Chunks, Production Scripts and Elicitation Stimuli. Each of these is specialised to provide the knowledge which is required to adapt a particular Requirements Engineering Task. Section 5.2 described in detail how reusable knowledge structures in RORE are utilised by the generic Requirements Engineering Tasks to transformation a concrete model in

a particular way. This section summarise the purpose and detailed structure of each reusable knowledge type.

6.2.1 Analysis Rules

Of the four kinds of reusable knowledge structure defined by RORE, Analysis Rules are the only structure type which are not utilised for the production of new information. Instead, their function within RORE is to define tests which can be fired against a model in order to draw conclusions about that model, specifically with respect to the quality or the completeness of the model. The capability provided by Analysis Rules is critical to the decision-making capability of the Analysis requirements task.

No general definition of model "completeness", or "quality" is given by the RORE framework. Instead, it is the rules themselves which inherently encode such definitions. A model is assumed to exhibit a certain aspect of quality if it satisfies a single analysis rule. A model is assumed to be complete if it satisfies all of the quality conditions that are associated with the type of that model.

A common structure for this kind of test in a range of human discourse is the logical rule. In formal logic, for instance, inference rules are used incrementally to prove that one proposition follows from another. Similarly, conditional statements in most high-level programming languages take the form of a rule or condition tested against an underlying set of variables. Given the tried-and-tested nature of this approach, this research chose the rule as the most appropriate representation for modeling conditional (analytical) knowledge in RORE.

Figure 6.2 illustrates the precise structure of an Analysis Rule.

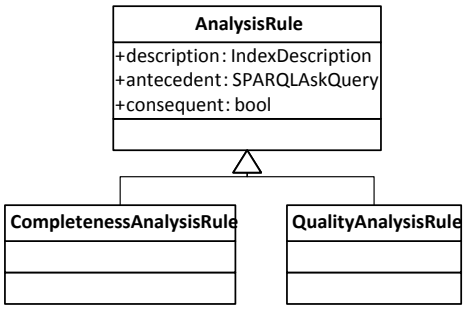

Figure 6.2: The structure of analysis rules

Aside from the *description* attribute, which is common to all reusable knowledge structures in RORE, Analysis Rules have two other attributes.

Antecedent is the attribute which stipulates the condition that an Analysis Rule imposes on a requirements model. This is a test which a model must pass in order to be considered of sufficient quality in a particular respect. The *antecedent* of an Analysis Rule is specified as a SPARQL-DL Ask query [SP07]: a kind of query over a model which returns a Boolean value indicating whether or not the model satisfies that query. Specifically, the *antecedent* is specified using Derivo's syntax [Sys12] for Ask queries. This was a pragmatic decision taken because it is Derivo's SPARQL-DL API through which the RORE prototype executes SPARQL-DL queries.

Consequent is a Boolean value which is the value to be returned in the event that the *antecedent* is satisfied by a model. This was introduced as a result of limitations in the contemporary syntax and semantics of SPARQL-DL Ask queries. Certain kinds of negative assertion cannot easily be expressed as an Ask query. The *consequent* provides a way of handling assertions of this kind by expressing a positive assertion and then negating the result.

6.2.2 Model Chunks

The most common form of reusable content described within the software engineering literature is a chunk of knowledge expressed through some representation, for example reusable source code [Kru92], components [LW05], or patterns [Gam95, Sut02]. The common theme which underpins such paradigms is the reuse of aggregations of sentences, in which each sentence is represented by a chosen (possibly natural) language, in which each sentence represents some fact or goal regarding an application domain or its software solution.

This approach is, as Maiden notes [MMS95], also naturally occurring. Experts reuse knowledge acquired through past experience in order to compose solutions to new problems [Gui90, Sut02]. Along these lines, the CHREST+ cognitive architecture is a theoretical framework, based on Simon's conception of ill-structured problem solving [Sim74] and his EPRAM architecture, which models human expert problem solving as a function of chunk abstraction and reuse [GLC+01]. Similarly, the reuse of sets of facts ("Model Chunks") about an application domain or its software solution facilitates a compositional approach to software design [Sut02]. This is powerful because it is an efficient approach to problem solving since it reduces the need to reason from-scratch about a problem or its solution. Maiden has shown that it can also have advantages for model correctness, because reusable fact sets can serve as prompts to

aid in clarifying rationale [Mai92]. It is for these reasons that this research opted to introduce Model Chunks as a mode of reuse in RORE.

The particular syntax of a Model Chunk is heavily inspired by that given in Sutcliffe and Maiden's Domain Theory [SM98, Sut02], which represents models as aggregations of facts, each instantiating a particular fact type. In RORE, both concrete Models and Model Chunks are represented in this way. However, whereas the Domain Theory identifies a concrete schema through which models are represented, RORE abstracts away from this and deals generically with Fact Types and Facts. It is this abstraction which enables RORE to support a range of requirements engineering modeling notations.

Figure 6.3 presents the structure of Model Chunks in RORE.

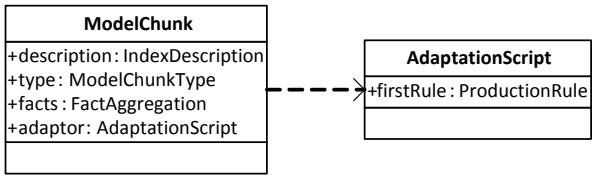

Figure 6.3: The structure of model chunks

Aside from the standard *description* attribute - which all reusable RORE knowledge structures share - Model Chunks have the following additional attributes.

Type links the Model Chunk to the Model Chunk Type which it instantiates.

Facts aggregates the Facts which the Model Chunk comprises. Facts are aggregated through a Fact Aggregation, which provides a layer of indirection between Model Chunks and the Facts themselves. Each Fact Aggregation aggregates multiple Facts of a particular Fact Type. In the case of Model Chunks, the Fact Aggregation must satisfy the cardinality constraint which is imposed by the Model Chunk Type for the corresponding Fact Type.

Adaptor is an optional attribute which allows the knowledge engineer to specify an Adaptation Script which can be applied to reify the Model Chunk for a given set of facts. The Adaptor cannot be utilised independently of the Model Chunk. When applying a Model Chunk to a target model, requirements engineers may need to reify the Model Chunk to fit the particular circumstances of its reuse context. Adaptation Scripts enable knowledge engineers to specify sequences of transformations through

which this reification can be automated, where this is possible. It should be noted that the current RORE prototype does not implement Adaptation Scripts.

6.2.3 Production Scripts

A major goal in designing the RORE framework was to offer support for procedural reuse to complement the declarative reuse of the Domain Theory. This goal was motivated by the claim of this thesis that procedural reuse is inherently more general than declarative reuse. Production Scripts provide an implementation of procedural reuse to satisfy this design goal.

The design of Production Scripts in RORE is heavily influenced by that of POSE problem transformations [HRJ08]. However, POSE operates by matching structures in a concrete problem description to the conclusion of a problem transformation, and then substituting the matched structure by the premise of that transformation. By contrast, RORE defines a number of operators through which is expressed the process by which a transformation can be realised. Compositions of these operators are encapsulated as Production Scripts.

Figure 6.4 illustrates the structure of Production Scripts, and of Production Rules from which Production Scripts are composed.

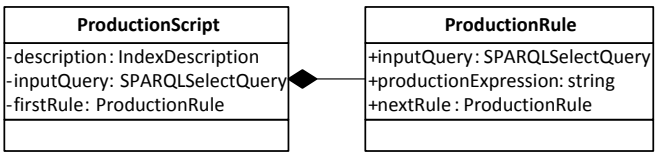

Figure 6.4: The structure of production scripts

Production Scripts themselves have two attributes:

- **inputQuery** is used by the Production Script to select from the source model the set of facts which the Production Script should transform. The *inputQuery* is expressed as a SPARQL-DL Select query [SP07], again using Derivo's syntax [Sys12]. SPARQL-DL Select queries are analogous to SQL Select queries in that they return a set of OWL Axioms which satisfice the query. Section 4.3.2 described the internal design of the prototype RORE Production Engine, which has an internal Production Dictionary. Facts which are selected by the *inputQuery* of a Production Script are copied into this Production Dictionary where they are transformed by the script.

- **firstRule** is a pointer to the first Production Rule in the Production Script: the starting point for the transformation which the Production Script enacts. Production Rules within a Production Script are organised into a linked list.

Production Rules, from which Production Scripts are composed, have three attributes:

- **inputQuery** is used by individual Production Rules to select, from the set of Facts that were retrieved into the Production Dictionary by the *inputQuery* of the Production Script, the specific Facts which that particular Production Rule will transform;
- **productionExpression** provides a textual description (the "source code") of the transformation which the Production Rule represents. The syntax for specifying the *productionExpression* is described below;
- **nextRule** provides a pointer to the next Production Rule in the linked list of Production Rules which defines the body of the Production Script.

Production Expressions in RORE are specified through a specialised and novel transformation language. Each Production Expression comprises three main elements:

- A **label** which is the name of the value, defined in the Production Dictionary, which will be returned as the result of the Production Script;
- A sequence of **declarations** which declare variables which the Production Script will transform and specify the initial value for each. The value may either be a reference to an object in the Production Dictionary, or it may be a literal value, or it may be the result of creating a new Fact;
- A sequence of **productions** which are the operations that collectively comprise the production rule.

Every production in a Production Expression is either an assignment or an iteration:

- **Assignment** operations update labels in the Production Dictionary with a new value;
- **Iteration** operations perform a specified sequence of transformations for each element in a value of type list.

Three kinds of value are defined in the Production Expression language:

- **Literal** values which must be primitives (strings, integers or Boolean values);
- **Objects** which are references to complex knowledge structures such as lists or RORE Facts;
- **Fact Creation** statements which create new Facts of the specified type.

The Backus-Naur grammar for Production Expressions is given below.

⟨*production_exp*⟩ ::= ⟨*label*⟩ ‘`<(`’ (⟨*declaration*⟩ ‘`;`’)* ‘`)<(`’ (⟨*production*⟩ ‘`;`’)* ‘`)`’

⟨*declaration*⟩ ::= ‘`VAR{`’ ⟨*label*⟩ ‘`,`’ ⟨*value*⟩ ‘`}`’

⟨*production*⟩ ::= ⟨*iteration*⟩ | ⟨*assignment*⟩

⟨*iteration*⟩ ::= ‘`[`’ ⟨*label*⟩ ‘`<-`’ ⟨*list-value*⟩ ‘`*:`’ (⟨*production*⟩ ’;’)* ‘`)`’

⟨*assignment*⟩ ::= ‘`ASSIGN{`’ ⟨*label*⟩ ‘`,`’ ⟨*value*⟩ ‘`}`’

⟨*value*⟩ ::= ⟨*label*⟩ | ⟨*fact_creation*⟩ | ⟨*object*⟩

⟨*object*⟩ ::= ⟨*literal*⟩ | ⟨*list-value*⟩ | ⟨*fact*⟩

⟨*label*⟩ ::= ⟨*name*⟩ (‘`.`’ ⟨*name*⟩)*

⟨*name*⟩ := [a-Z]([a-Z0-9_]*)

⟨*literal*⟩ ::= ⟨*boolean*⟩ | ⟨*integer*⟩ | ⟨*string*⟩

⟨*fact_creation*⟩ ::= ‘`CREATE{`’ ⟨*label*⟩ ‘`,`’ ⟨*fact-type-label*⟩ ‘`}`’

⟨*list-value*⟩ ::= ⟨*label*⟩

⟨*special-var*⟩ ::= ‘`RESULTSET`’ | ‘`name`’ | ‘`iri`’ | ‘`factType`’

⟨*fact-type-label*⟩ ::= ⟨*label*⟩

6.2.4 Elicitation Stimuli

The introduction of Elicitation Stimuli into RORE was intended as a means of abstracting the knowledge which underpins the Fact Acquisition Dialogue in Maiden's earliest versions of the Domain Theory [Mai92] and of the Requirements Capturer in later versions of the AIR toolset [SM98]. In the Domain Theory, facts are elicited from the user through a computer-directed dialogue. AIR prompts a user to input facts of a particular kind at key moments in time, and the user responds by using the provided interfaces to input the requested knowledge. Specialised dialogues are used to elicit instances of the particular set of fact types and reusable abstractions which the Domain Theory prescribes [Mai92, Sut02].

However, RORE has no in-built knowledge of any pre-defined fact types, beyond those discussed in this chapter, and assumes no reusable knowledge structures. Consequently, if RORE is to provide a procedure for communicating with information sources outside of the RORE framework itself (the requirements engineer is considered internal to the RORE framework, as they are responsible for enacting its processes), then a procedure is also required to reify the computer-directed dialogue for the needs of a specific context.

Elicitation Stimuli are a kind of reusable abstraction which hold the declarative knowledge needed to inform this reification. The concept of an "elicitation stimulus" was introduced by [SW90], who provide a detailed cognitive account of the knowledge acquisition process. According to this account, a knowledge engineer uses "elicitation stimuli" to prompt a response from domain experts, and so to elicit knowledge from them. These stimuli are derived from that knowledge which has already been acquired during prior elicitation cycles.

The structure of Elicitation Stimuli is given in Figure 6.5.

Figure 6.5 shows three specialised types of Elicitation Stimulus: the Fact Editing Stimulus, the Chunk-based Stimulus and the Multiple Choice Stimulus. The original specification of RORE defined just one type of Elicitation Stimulus which had the same structure as the Chunk-based Stimulus. However, evaluation of the RORE prototype indicated the need to support different types of response structure and so the three specialised types of Elicitation Stimulus, shown in Figure 6.5, were developed.

All three types of Elicitation Stimulus share in common two attributes, in addition to the standard Index Description attribute:

- **contextualSummary** is a string attribute which provides a textual description of the Modeling Context to which the Elicitation Stimulus is applicable. Through

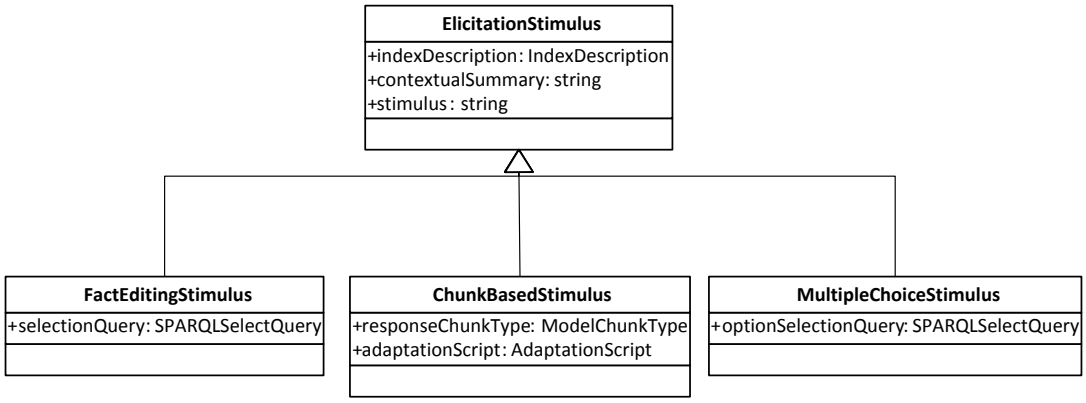

Figure 6.5: The structure of elicitation stimuli

this attribute, the Elicitation Stimulus provides to the requirements engineer broad contextual knowledge which may be useful in responding to the stimulus.

- **stimulus** is a string attribute which provides a textual description of the stimulus itself (usually a question about the target model or the requirements scenario being modelled).

In addition to these common attributes, each type of Elicitation Stimulus defines additional attributes which configure the structure of the response.

The Fact Editing Stimulus presents to the requirements engineer a set of Facts, selected from the source model, and asks the user to reify them to suit the target model. The Fact Editing Stimulus has just one additional attribute - *selectionQuery* - which is a SPARQL-DL Select query which stipulates the Facts which should from the source model that are to be presented to the user.

The Chunk-based Stimulus requests a user to provide a response in the form of a Model Chunk. The Chunk-based Stimulus defines two addition attributes:

- **responseChunkType** stipulates the Model Chunk Type to which the response Model Chunk must conform. It defines the structure of the Model Chunk, and is also used in the RORE prototype to configure the Chunk Manager to provide an intuitive interface through which to specify the response chunk.

- **adaptationScript** points to an optional Adaptation Script (which reusable Model Chunks also optionally stipulate). The Adaptation Script stipulated by a Chunk-based Stimulus is used to perform any post-processing on the response chunk, as deemed appropriate by the knowledge engineer when specifying the Chunk-based Stimulus.

The Multiple Choice Stimulus presents to the requirements engineer a set of Facts as optional responses to the *stimulus* question. The *optionSelectionQuery* attribute specifies a SPARQL-DL Select query through which are selected the Fact options that will be presented to the requirements engineer.

6.2.5 Index Descriptions

Index Descriptions capture meta-data about the range of requirements Modeling Contexts to which a reusable knowledge structure in RORE is likely to be applicable. They formalise statements about each of these facets of a reusable knowledge structure so as to support the efficient retrieval of reusable knowledge structures that will likely be applicable to resolving a particular Information Requirement.

The question of how to organise RORE's knowledge bases to support both effective and efficient knowledge retrieval has been a significant challenge in designing RORE. Recent advances in search algorithms (such as the analogical algorithms [Fin88, Mai92] which Chapter 2 reviewed, and recommender systems [AT05] have mitigated against the need for classification schemas for organising knowledge bases because, as search algorithms have evolved, the focus has shifted from requiring users to select components manually towards using intelligent algorithms to retrieve context-relevant components. Taxonomies may still be relevant because they provide a computationally cheap way of narrowing down the pool of candidates over which more precise search algorithms will operate, as in the Domain Theory [Mai92, SM98, Sut02]. However, taxonomies in modern reuse libraries are intended for computational, rather than human, consumption.

In designing the retrieval process for RORE and, accordingly, the indexing system this research took all of these historical developments into account. As 5 discussed, RORE adopts the filtering approach taken by the Domain Theory's AIR tool. The goal of retrieval in RORE is to identify reusable knowledge structures which will be applicable to satisfy an information requirement in the context of a given source and target model. Thus this thesis sought to identify properties of a Modeling Context (see

Section 6.4.2) which could be used to rapidly filter out all reusable structures which are *not* appropriate in the current context.

This review of requirements reuse presented by Naish and Zhao [NZ11] identifies three main aspects of a Modeling Context are particularly salient for the retrieval of a reusable knowledge structure:

- The Requirements Task which is currently being enacted by the requirements engineer;
- The Problem Domain which is currently being modelled;
- The level of abstraction at which the reusable structure is described.

Figure 6.6 illustrates the structure of an Index Description.

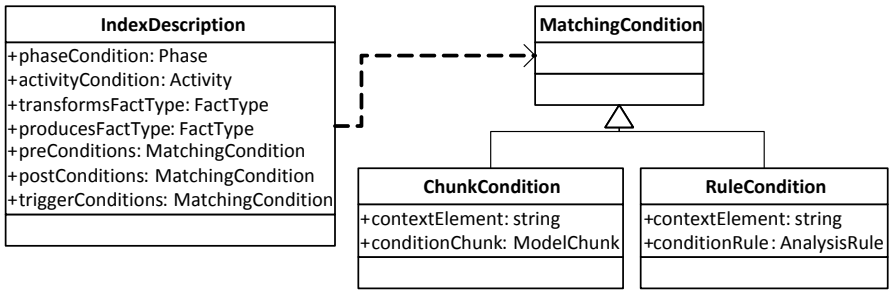

Figure 6.6: The structure of index descriptions

Index Descriptions define seven key attributes. The Matching procedure tests each of these in turn against the current Modeling Context and Information Requirement in order to determine whether or not a reusable knowledge structure matches that context. The first four attributes support very lightweight equality checks (direct string comparisons between the Index Description and the Modeling Context). This allows unlikely matches to be ruled out rapidly and cheaply:

- **phaseCondition** indicates that a reusable knowledge structure is applicable to a particular Phase of a requirements engineering method and so allows a match to be dismissed on the grounds that a reusable knowledge structure is not appropriate in the current Phase;
- **activityCondition** links a reusable knowledge structure to a particular Activity so that it can be ruled out as a match in the event that it is not applicable to that Activity;

- **transformsFactType** stipulates the types of Fact which a reusable knowledge structure seeks to transform (recall that reusable knowledge structures transform input Facts into new sets of Facts). This allows, as a kind of simple precondition, the Matching Engine to check that Facts of the necessary type are available in the source model to enable a Transformation to be applied successfully.

- **producesFactType** specifies the types of Fact which a reusable knowledge structure produces. This is matched against the Information Requirement to ensure that the the reusable knowledge structure will satisfy the goal of the current Analysis-Action cycle.

The final three attributes (pre-, post-, and triggerConditions) allow for the fine tuning of a match, but are more costly:

- **preConditions** describe conditions over a Model which must hold true in order for application of the reusable knowledge structure successfully to satisfy the Information Requirement;

- **postConditions** describe conditions over a Model which describe what *will* hold true if the reusable knowledge structure was applied successfully;

- **triggerConditions** describe conditions over a Model which indicate that this reusable structure should be applied in the current context.

Conditions which are assigned to these final three attributes can be expressed through one of two condition types:

- **Chunk Conditions** express conditions over Models in the form of Model Chunks. Chunk Conditions are evaluated against a Model using the *Analogical Matcher* described in Section 5.8.3. The *conditionChunk* attribute of a Chunk Condition indicates a Model Chunk which should be analogically valid for the given Model in order for the Model to satisfy the Chunk Condition.

- **Rule Conditions** express conditions over Models in the form of Analysis Rules. Rule Conditions are evaluated against a Model in the same way that Analysis Rules are. The Analysis Rule defines the rule which a Model should satisfy in order to satisfy the condition as a whole.

Both types of condition also specify a *contextElement* attribute which indicates whether the condition should be tested against the source or target model.

6.3 Knowledge Structures for Metamodeling

The main purpose of RORE as a whole is to produce requirements models by reuse. However, as discussed in Chapter 3, RORE abstracts away from the details of any specific requirements modeling method in order to support a range of different notations. However, RORE requires some knowledge modeling formalism through which reusable chunks of declarative requirements knowledge can be (semi-)formally specified. RORE therefore introduces a metamodeling layer through which requirements RORE can be extended to support specific modeling notations without building knowledge of that notation into the RORE framework itself. It is at this layer that the Model, Chunk and Fact Types are defined, and it is this layer of RORE with which the Knowledge Engineering perspective is predominantly concerned.

This thesis sought to adapt existing work to suit the needs of the RORE framework. This work draws in particular on the Telos knowledge representation framework [MBJK90] and on the Object Modeling Group's MetaObject Facility (MOF) [Gro11] which is the meta-modeling framework used to define the UML [Gro09] and other OMG specifications. Figure 6.7 shows the RORE knowledge structures through which metamodeling is achieved.

The first knowledge structure which is defined in the metamodeling layer of RORE is the Model Type. The Model Type aggregates together all of the Fact Types through which knowledge can be expressed within a particular requirements modeling notation. The Model Type comprises two attributes:

- A **name** which corresponds to the name of the requirements modeling notation which this Model Type represents (e.g. i* or KAOS);

- **comprisesTypes** is a list of Fact Type aggregations through which the Model Type is associated with the Fact Types which it comprises.

Fact Type Aggregations provide a layer of indirection between the Model Chunks and Facts themselves, and comprise a single attribute (*aggregatesType*) which is a pointer to the Fact Type which they aggregate.

Mirroring the schema through which Model Types are defined, the metamodeling layer of RORE also provides support for defining Model Chunk Types. Model Chunk Types broadly mirror Model Chunks in structure (the *comprisesFactType* attribute of a Model Chunk Type plays the same role as the Model Type's *comprisesType* attribute),

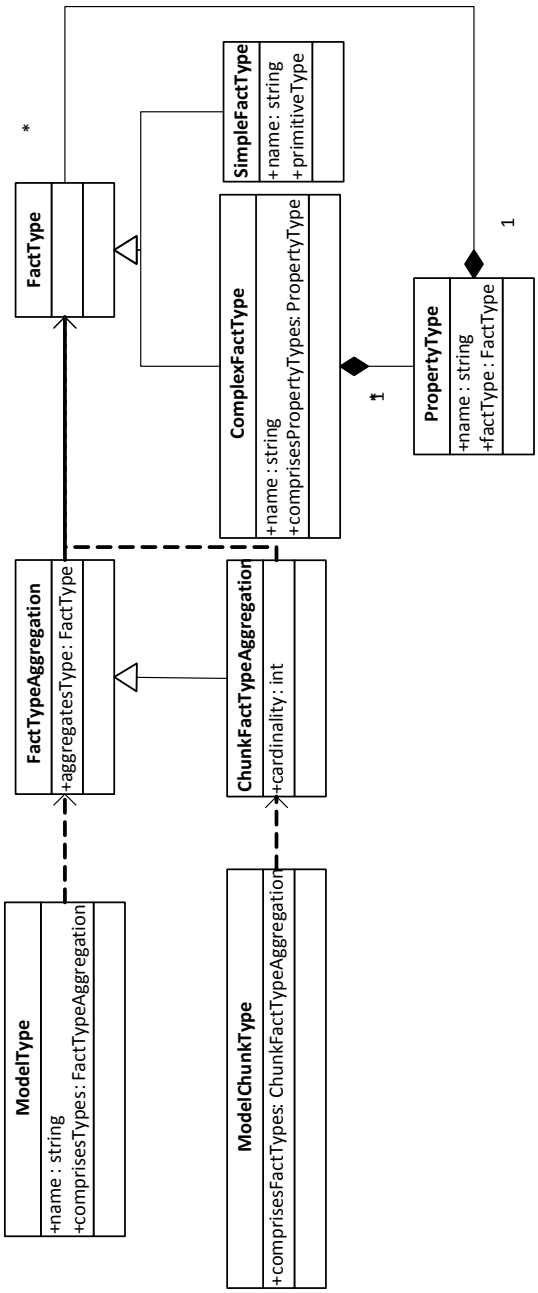

Figure 6.7: Knowledge structures for metamodeling

but are distinguished from Model Types as they entail additional constraints: particularly a cardinality constraint which limits the granularity of a Model Chunk. This cardinality constraint is stipulated for each Fact Type in a Model Chunk Type through the *cardinality* attribute of the Chunk Fact Type Aggregation. The Chunk Fact Type Aggregation is a specialisation of the Fact Type Aggregation.

Both Fact Type Aggregations, and Chunk Fact Type Aggregations, aggregate Fact Types through their *aggregatesType* attribute. Fact Types are divided into two subclasses: Complex Fact Types and Simple Fact Types. Complex Fact Types aggregate Simple Fact Types into larger structures of arbitrary complexity. They comprise two attributes:

- A **name** which defines the public identifier of the Fact Type;
- **comprisesProperyType** which identifies the Property Types that the Complex Fact Type aggregates.

It is through Property Types that Complex Fact Types aggregate less complex Fact Types. Property Types are used to provide a name to the binary relationship which holds between a Complex Fact Type and another Fact Type. A Complex Fact Type may be related to any one Fact Type through multiple Property Type relations. Property Types define two attributes:

- A **name** which identifies the Property Type;
- **factType** which identifies the Fact Type of the value of this Property Type.

Ultimately, all Complex Fact Types are hierarchical aggregations of Simple Fact Types. Simple Fact Types have just one value which must a primitive value (string, Boolean, or integer). Simple Fact Types have two attributes:

- A *name* which identifies the Simple Fact Type;
- A *primitiveType* which points to the kind of primitive value that instances of this Fact Type may have.

6.4 Structures for Representing Requirements Knowledge

6.4.1 Facts and Models

Requirements Models in RORE are instantiations of the Model Types which are defined at the meta-level as discussed in the previous section. Models are the primary concern of the Requirements Engineering perspective. Figure 6.8 illustrates the structure of Models in RORE.

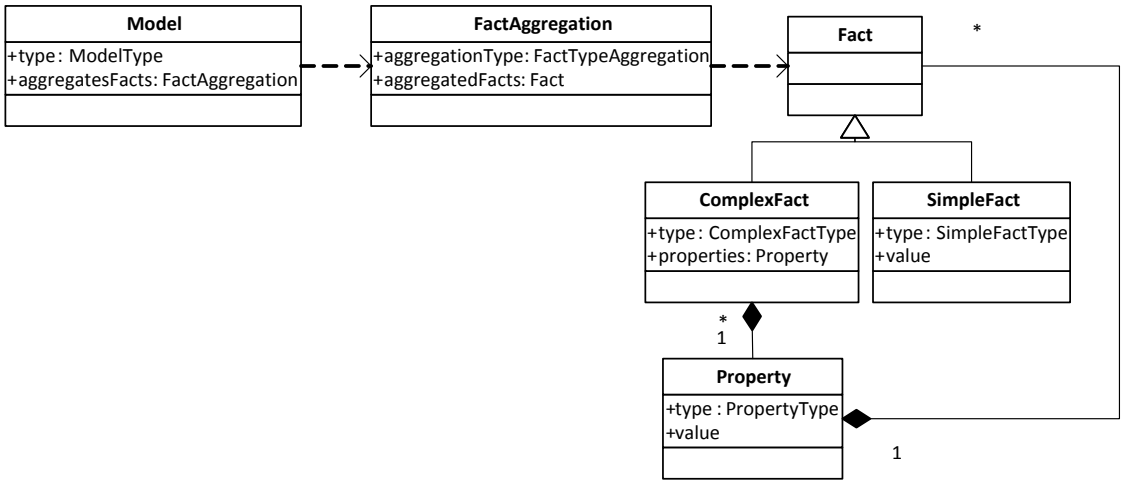

Figure 6.8: Knowledge structures for modeling

The structure of Models in RORE was dictated primarily by that of Model Types. The hierarchy of individuals which form a Model precisely reflects the hierarchy of types which form a Meta-model in RORE (see Figure 6.9).

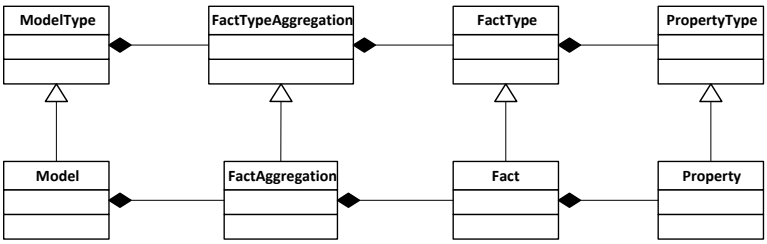

Figure 6.9: Symmetry between the structure of RORE's meta-model and model layers

Thus, for each type class (ModelType, FactType, PropertyType) defined at the meta-level, an individual class is defined at the Model-level. The exception is the Model Chunk, which is defined in the reuse library.

Models in RORE comprise aggregations of Facts. Models have two attributes:

- **type** is a pointer to the Model Type which this Model instantiates. The Model Type defines the general structure for the Model;
- **aggregatesFacts** is a pointer to a set of Fact Aggregations through which this Model aggregates Facts.

Fact Aggregations also have two attributes:

- **aggregationType** is a pointer to the particular Fact Type Aggregation which this Fact Aggregation instantiates. The Fact Type Aggregation which a Fact Aggregation instantiates must be valid for the Model Type which is instantiated by the Model to which that Fact Aggregation belongs.
- **aggregatedFacts** is an attribute which points to the list of all Facts that are aggregated by this Fact Aggregation. All of the Facts which are aggregated by a single Fact Aggregation must be of the same Fact Type, and that Fact Type must be the one which is specified by the Fact Aggregation type.

Facts in RORE fall into two subtypes: Complex Facts and Simple Facts, mirroring the distinction made in the metamodeling layer. Complex Facts have two attributes:

- **type** is a pointer to the Complex Fact Type which this Complex Fact instantiates;
- **properties** is a pointer to each of the Property instances which this Complex Fact Type aggregates.

Properties at the Model-level instantiate the Property Types which are associated with a Complex Fact Type at the meta-level. Instantiation of a Property Type involves assigning a value to that Property Type for a particular Complex Fact. Properties describe these instantiations. They comprise two attributes:

- **type** is a pointer to the Property Type which this Property instantiates;
- **value** is a pointer to the Fact which is assigned as the value of this Property.

Finally, Simple Facts are the primitive knowledge structures from which Complex Facts are aggregated, mirroring the aggregation of Simple Fact Types into arbitrarily Complex Fact Types. They are defined by two attributes:

- **type** which points to the Simple Fact Type that this Simple Fact instantiates;
- **value** which points to the primitive value of this Simple Fact.

6.4.2 Defining the Modeling Context

In RORE, the terms "Modeling Context" and "Contextual Knowledge" are used to refer collectively to the sum total of knowledge which consistutes the state of the requirements engineering process at a given instant in time. The Modeling Context is therefore primarily relevant to the Requirements Engineering perspective. This includes both domain knowledge, captured in the form of the source and target model, and information about the requirements engineering process itself, in particular the current Information Requirement. Contextual Knowledge is stored in working memory, and largely instantiates the knowledge which is defined in Long-Term Memory. With the exception of the Information Requirement, the Modeling Context contains no knowledge for which this is not true.

A Modeling context, therefore, is an aggregate knowledge structure which aggregates the knowledge that a requirements engineer uses to undertake a particular modeling activity. This is illustrated in Figure 6.10

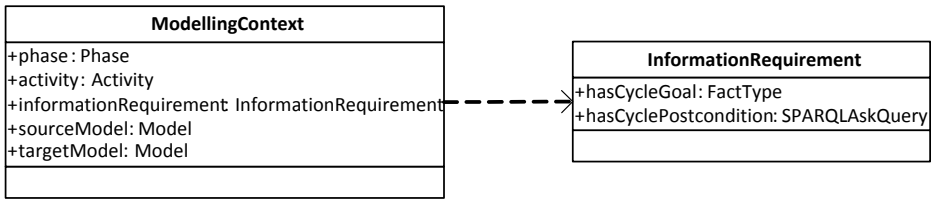

Figure 6.10: The structure of modeling contexts

A Modeling Context aggregates five attributes which collectively define the current state of working memory:

- **phase** represents the current Phase in which the requirements engineer is engaged within the Requirements Engineering perspective. *phase* points to a particular Phase which is defined in long-term memory;

- **activity** represents the current Activity of the requirements engineer at a given moment in time. It is a pointer to an Activity which is part of a particular requirements method that is defined in long-term memory;

- **informationRequirement** points to the Information Requirement which was generated during the current Analysis-Action cycle. Information Requirements specify the goal of a RORE cycle. Their structure is defined more fully below;

- **sourceModel** points to the Model from which information is to be extracted to support refinements during the current requirements engineering Activity;
- **targetModel** points to the Model in working memory which the current requirements engineering Activity aims to refine.

The structure of Information Requirements is also illustrated in Figure 6.10. Information Requirements have two attributes:

- **hasCycleGoal** which points to the Fact Type which the current Analysis-Action cycle is primarily concerned with producing. This attribute is compared against the *producesFactType* attribute of Index Descriptions during matching;
- **hasCyclePostcondition** which records the SPARQL-DL Ask condition of the Analysis rule from which the Information Requirement was generated. The *postCondition* of an Index Description satisfies an Information Requirement if it is logically equivalent to this attribute of the current Information Requirement.

6.5 Modeling the Requirements Engineering Process

Section 6.2.5 discusses how different properties of an index description are used during knowledge structure retrieval to rapidly reduce the size of the pool of candidate knowledge structures. Procedural modeling in RORE is motivated by this filtering process, as it provides a landscape within which a particular modeling context can be situated. Index descriptions point to the behavioural units for which associated reusable structures are relevant. The modeling context also references the phase and activity associated which are currently being enacted by a particular RORE session. This arrangement facilitates the matching of reusable structures to modeling contexts.

It is common in the literature on software methodology to define requirements methods at two levels of abstraction. The Booch method, for instance, distinguishes macro-processes, which consist of sequences of engineering phases, from micro-processes RORE models, which are lower-level activities that are enacted iteratively to realise the goals of a phase. RORE models requirements engineering processes at three levels of abstraction:

- **Phases** are coarse-grained, high-level behavioural units which produce an artefact, or set of artefacts, of a particular kind;

- **Activities** are finer-grained, lower-level activities which interact are carried out collectively to realise the goals of a phase;

- **Refinements** are the lowest-level form of procedural specification representing individual changes to requirements models. Refinements are represented by the reusable structures described in Section 6.2 and are enacted by RORE's requirements tasks so they are not discussed further here.

Phases and Activities are treated in RORE as specialisations of the more general concept of a "Behavioural Unit". This decision was made to support extensibility in future versions of RORE, to allow modeling of process at arbitrary levels of abstraction. Figure 6.11 shows the structure of Phases and Activities in RORE:

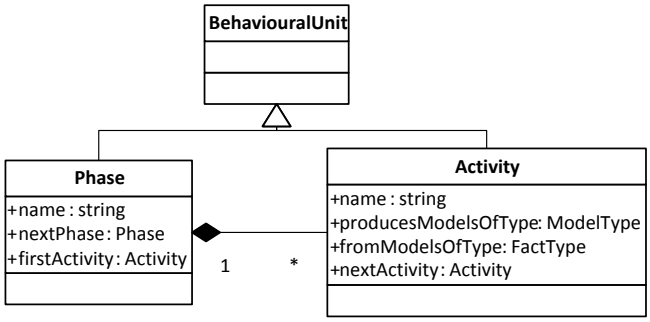

Figure 6.11: Knowledge structures for modeling the RE process

Phases have three attributes:

- A **name** by which the Phase is known to requirements engineers;

- **nextPhase** indicates the Phase which follows this Phase in the requirements engineering method of which this Phase is a part;

- **firstActivity** which indicates the Activity that starts the process by which this Phase is realised.

Activities are defined by the following four attributes:

- A **name** by which the Activity is known to requirements engineers;

- **producesModelsOfType** indicates the type of target models which this Activity aims to produce or refine;

- **fromModelsOfType** indicates the type of source models from which this Activity produces target models;
- **nextActivity** indicates the Activity which immediately follows this Activity in the Phase of which this Activity is a part.

6.6 Summary

This chapter described the knowledge structures through which requirements knowledge is represented in RORE. Knowledge specifications in RORE are described at three layers of abstraction, as discussed in Section 3.5.1. At the highest level of abstraction (the Immutable Layer), the RORE schema defines the structures through which knowledge is specified in the lower layers of abstraction. The Meta Layer instantiates a subset of the knowledge structures which are defined in the Immutable Layer in order to support the Knowledge Engineering view of RORE. At this layer, knowledge structures are instantiated in order to describe meta-models for the requirements modeling notations and requirements engineering methods which RORE will support, as well as the construction of libraries of reusable requirements knowledge structures. Finally, the lowest layer - the Modeling Layer - instantiates a subset (non-overlapping with the knowledge structures that are instantiated at the Meta Layer) of the Immutable Layer knowledge structures in order to support the definition of concrete, project-specific requirements models. The Modeling Layer also supports the instantiation of knowledge structures which define the current state of working memory at a particular point in time during a particular RORE session. All knowledge structures in RORE - at every layer of abstraction - are fully formalised using the Web Ontology Language (OWL) knowledge formalism [BVHH+04].

Chapter 7

Representing Reusable RE Knowledge: The KE Viewpoint

7.1 Introduction

This chapter describes a novel procedure for producing ISMs (Information System Models) from OSMs (Object System Models). The procedure uses a combination of heuristics (procedural reuse) and patterns (declarative reuse) to achieve the transformation. Because of the mix of reasoning styles on which the procedure depends, this thesis considers it to be the prototype of an effective RORE procedure. This chapter demonstrates how a requirements activity and the knowledge which underpins that activity can be formalised using the RORE approach. This chapter first presents a refined version of the Domain Theory's meta-schema for describing OSMs. Next, this chapter presents a novel representation schema for describing software specification knowledge in the form an ISMs. Finally, this chapter describes the procedure for transforming between these two requirements modeling notations. These three components have each been formalised using RORE, and the resultant long-term memory knowledge base is presented in full in Appendix B.

7.2 Representing Software Requirements

This case study uses a refined version of Sutcliffe and Maiden's Domain Theory meta-schema and OSM library to model requirements. Chapter 2 briefly summarises the Domain Theory as Sutcliffe and Maiden defined it [SM98, Sut02]. This section presents the refined version of the modeling formalisms which underpin the theory. Figure 7.1 presents the refined meta-schema:

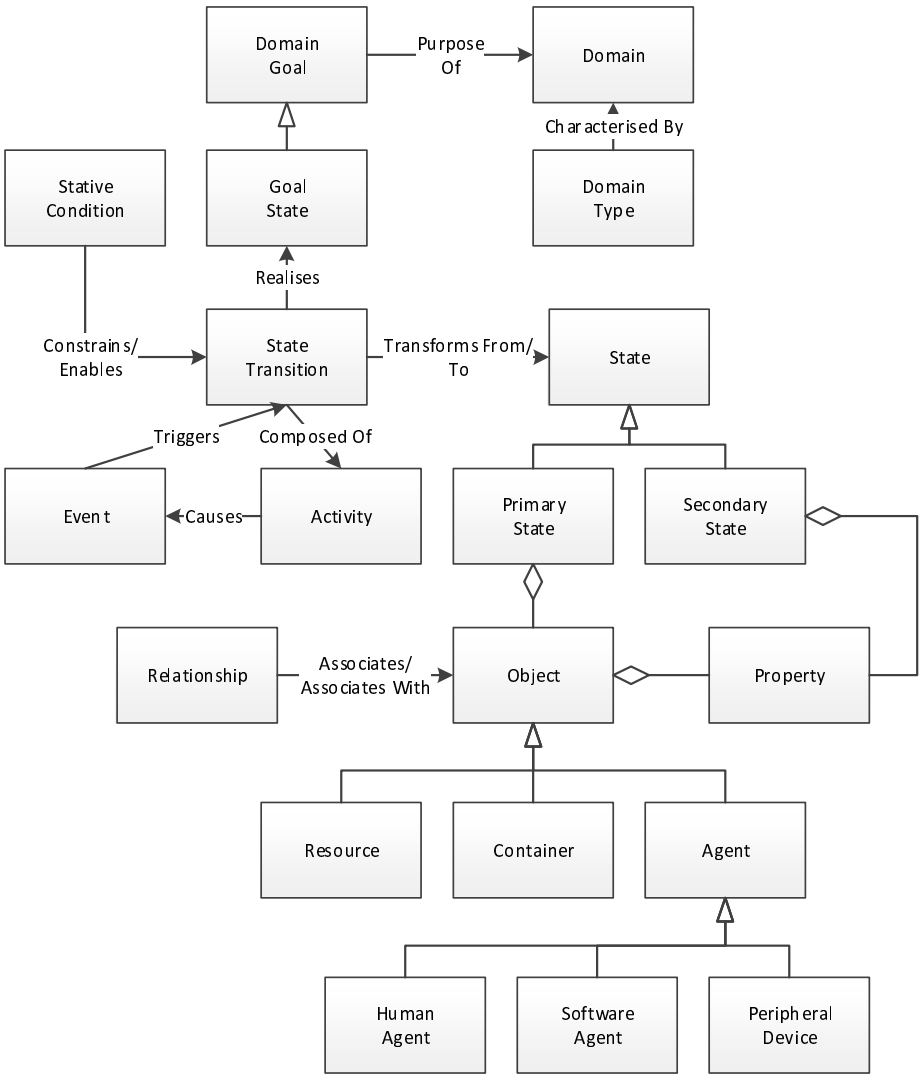

Figure 7.1: The Refined Domain Theory Meta-schema

The most radical refinements occurred at the highest-level, although this research also made some lower-level refinements. The significant changes are as follows:

- The introduction of a *Domain* fact type which aggregates all of the facts for an

individual concrete domain. A complex application domain model will aggregate multiple domains;

- The introduction of a *Domain Type* fact type which enumerates the kinds of domain (Physical, Conceptual, Financial) that an OSM might model;
- The introduction of a *Domain Goal* fact type which enumerates a set of reusable, pre-defined goal states from which a domain can be composed;
- The redefinition of the *State Transition* fact type so that state transitions can transform either primary or secondary states;
- The generalisation of the *Secondary State* fact type so that secondary states aggregate properties and their values;
- The introduction of three new *Agent Subtypes*: Human, Software, Peripheral Device.

These refinements are clear when the modified metaschema (as in Figure 7.1) is compared to Sutcliffe and Maiden's original metaschema as illustrated in Figure 2.1 and reproduced in Figure 7.2.

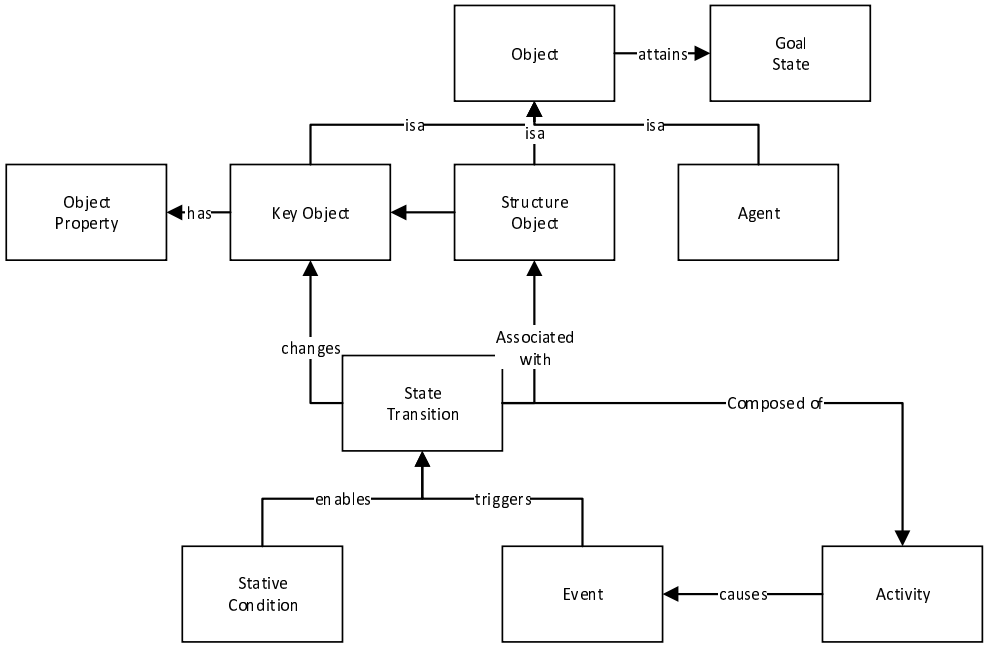

Figure 7.2: The Domain Theory's meta-schema for expressing domain knowledge as described in [SM98]

For those fact types which were defined in Sutcliffe and Maiden's meta-schema for representing domain knowledge [SM98], the definition of the fact type remains unchanged unless specified otherwise.

The remainder of this section gives detailed definitions for those fact types which are novel in the refined OSM meta-schema (Domain, Domain Type, Domain Goal, Agent Subtype).

7.2.1 Domains

Domains are the highest-level fact type within the restructured meta-schema. Each Domain has a single Domain Goal, through which the Domain aggregates the facts that describe its structure and behaviour. They also have a Domain Type attribute which denotes whether the domain is physical, conceptual or financial. Finally, they have an attribute "Monitored" which has a Boolean value and denotes whether or not the to-be software system is to gather information about, and report on, this domain.

7.2.2 Domain Goals

Goal Type	Description
Allocation	An activity in which a planned (but currently unrealised) association is created between a set of key objects and a target structure.
Control	An activity in which one key object representing a message is passed to a second key object, representing a resource, causing that resource to adjust its behaviour.
Composition	An activity in which one set of key objects is aggregated to produce a new set of composite key objects.
Decomposition	An activity in which a set of composite key objects is disaggregated to produce a set of component key objects.
Manipulation	An activity in which a controlling agent performs actions in order to transform the secondary state of a set of key objects.
Sensing	An activity in which some Agent must detect and monitor changes in either the primary or secondary state of a key object.
Transfer	An activity in which a key object is transferred in some sense from a source structure to a target structure.

Table 7.1: Domain Goals, derived from top-level OSMs

Domain Goals are reusable abstractions, analogous to OSMs in that they represent high-level abstractions which describe commonly occurring goals and structures across a range of domains. This research identifies eight atomic Domain Goals which are derived from the top-level OSM families identified by Sutcliffe and Maiden [SM98] (see Figure 2.2). The Domain Goals are described in Table 7.1.

Each Domain Goal is defined by a single goal state, which in turn consists of an initial state, a goal state and a set of transitions which take the system represented by the Domain Goal from the initial state to the goal state. Domain Goals, then, have all of the structure which the original OSM library defined. However, Domain Goals contain no information about the type of domain within which they occur.

Goal Type	**Defining Concept**	**Representation**
Allocation	Relationship to Slot	Structural Relationship
Control	A Behaviour	-
Composition	A Composition	Relationship Between Key Objects
Decomposition	A Composition	Relationship Between Key Objects
Manipulation	A Secondary State	A Secondary State
Sensing	A State	Secondary State
Transfer	Location	Structural Relationship

Table 7.2: Domain Goals are characterised semantically by Defining Concepts

Within the refined Domain Theory, each Domain Goal is associated with a defining concept which is the key concept which is which must be grasped in order to acquire a semantic understanding of the goal. Defining concepts are the elements of a domain which are transformed by the state transitions within that domain and so define the subject matter of that domain. The defining concepts for each Domain Goal are shown in Table 7.2.

7.2.3 Domain Types

We identify four Domain Types. These are presented in Table 7.3:

Domain Types provide additional context for Domain Goals by providing semi-formal definitions for the defining concepts of each Domain Goal. Each Domain Type is associated with a library of reusable object properties and relationship types from which these definitions are constructed. Table 7.4 presents the definition for each defining concept where these can be fixed a priori.

The primary advantage of the Domain Type library in the refined version of the Domain Theory is that it supports the introduction of a rich body of knowledge about

Domain Type	Description
Physical	A domain which is grounded in the physical, material world
Conceptual	A domain which is rooted, at some level, in the physical world, but which is abstract and consists predominantly of informational structures and resources
Physical Financial	A domain grounded in the physical, material world but in which objects are of interest primarily because they have some economic value
Electronic Financial	A conceptual domain in which conceptual objects have financial value. This includes domains in which financial transactions are processed electronically

Table 7.3: Domain Types, derived from key object types, define the nature of entity to which objects within a domain pertain

different kinds of domain. This knowledge is in addition to the general knowledge which is provided by the Domain Goal, and the context-specific knowledge which is provided by the Requirements Engineer. As Section 7.4 discusses, this additional knowledge can be utilised by reuse designers and model generators in order to reason about software solutions to the domain problems which Domain Goals represent. Furthermore, this knowledge can be assumed to be relatively stable. This is because each Domain Type, in essence, represents a particular kind of system, each of which has been studied from a scientific perspective over several centuries. Table 7.5 shows the academic and scientific disciplines which have studied the types of system represented by each Domain Type:

Because of the nature of scientific discourse, each of these disciplines has developed a relatively stable ontological understanding of the systems which they each study. It is from these scientific bodies of knowledge that this research identified the reusable properties which are associated with each Domain Type. The International System of Units (SI) was a particularly useful resource for defining the Physical Domain Type as it provides a readily-available list of standard properties for different aspects of chemical and physical entities. However, this research also drew on a range of other literature from the disciplines identified in Table 7.5 in constructing the refined Domain Type library. Table 7.6 summarises the most commonly useful properties for each Domain Type.

Defining Concept	**Physical**	**Conceptual**	**Physical Financial**	**Electronic Financial**
Slot	Physical Location	-	Physical Financial Location	Electronic Financial Location
-	-	-	-	-
Composition	Physical Proximity	Mathematical/ Logical Relationship	Addition/ Subtraction	Addition/ Subtraction
Secondary State	Defined by Physical Properties	Defined by Conceptual Properties	Defined By Physical Financial Properties	Defined By Electronic Financial Properties
Location	X, Y and Z Positions	Logical Path and Physical (IP) address	Bank Account, Customer, Merchant, Till	Bank Account Number, Sort code

Table 7.4: Domain Types provide more concrete definitions for the defining concepts associated with each Domain Goal

Domain Type	Related Disciplines
Physical	Physics, Chemistry
Conceptual	Information Science, Computer Science, Mathematics
Physical Financial	Economics, Business, Financial Science
Electronic Financial	Economics, Business, Financial Science

Table 7.5: Domain Types represent types of system which are studied by a range of scientific disciplines

Domain Type	Property	Values	Unit
Physical	Length Height Width X Position Y Position Z Position Mass Temperature	Real Number Real Number Real Number Real Number Real Number Real Number Real Number Real Number	Metre Metre Metre Point Point Point Kilogram Kelvin, Celsius, Fahrenheit
Conceptual	Quantity Physical Location Logical Location Structure	Real Number IP Address File Path Attribute Type List	Bit - - -
Financial	Value Currency	Double EUR, GBP, USD...,	Currency -
Electronic Financial	Account Number Sort Code	Formatted String Formatted String	- -

Table 7.6: Domain Types are associated with libraries of pre-defined reusable properties which add semantic value to a domain model

7.2.4 Agent Types

Agent Type	Related Disciplines
Human	A human actor manually enacts the state transitions within a domain
Software	A software system directly enacts the state transitions within a domain
Peripheral Device	A software system controls a peripheral device to enact the state transitions within a domain

Table 7.7: Agent Types indicate the kind of actor which enacts the state transitions within a domain

The restructured Domain Three introduces three Agent Types as presented in Table 7.7. These Agent Types add pieces of information to a domain model which begin to describe the requirements for the to-be system. They denote, in conjunction with the "Monitored" attribute of a Domain fact, the degree of responsibility which the to-be-implemented software system has within that Domain. This knowledge is particularly useful in the later stages of system specification when it is necessary to know in what way the software system should interact with the domain. The primary distinction between the "Monitored" attribute and the Agent type is that the "Monitored" attribute indicates the data and reports which the to-be system must gather and report on, whereas the Agent Type indicates the extent to which the to-be system will be responsible for transforming the Domain. Table 7.8 illustrates the generic requirements that are implied by each Agent Type. These generic requirements are central to the transformation process, which is outlined in Section 7.4.

Agent Type	Generic Requirement
Human	*None*
Software	Manipulate the domain to realise the goal state
Peripheral Device	Control the Peripheral Device Agent to realise the goal state

Table 7.8: Agent Types indicate the level of interaction which a software system may have with the activities within an application domain

There are strict rules governing the Domain Types to which each Agent Type is applicable. These rules arise from the fact that in order to enact a state transition, an Agent must have the faculties which are necessary to allow it to enact that transition. Table 7.9 illustrates which Agent Types are applicable to which Domain Types. In

general, Software is designed to manipulate information structures which are represented by conceptual domains, but can also manipulate Electronic Financial domains. Software has no way, however, to interact with the Physical world, except through Peripheral Devices (such as printers, card readers, robotic arms, and so on). Thus Peripheral Devices can interact with Physical Domains, where Software cannot. In principle, Peripheral Devices can also interact with conceptual structures, but empirically this research found no examples of domains in which this actually occurred. Human actors have both physical faculties which allow them to manipulate Physical Domains, and conceptual faculties (brains) which allow reasoning over Conceptual domains. However, Human interaction with Electronic Financial domains tends to be through software systems and peripheral devices, so the Human Agent cannot be considered a primary actor in these Domain Types.

Domain Type	Human	Software	Peripheral Device
Physical	✓	-	✓
Conceptual	✓	✓	-
Physical Financial	✓	-	✓
Electronic Financial	-	✓	✓

Table 7.9: An overview of the types of agent which can enact an activity in different types of domain

7.3 Information System Models: An Overview and Rationalisation

The original formulation of the Domain Theory introduced the concept of an Information System Model (ISM) [SM98, Sut02]. According to Sutcliffe [Sut02]:

> [ISMs] represent processes that feed off, report on and provide external representations of information contained within OSMs...The essential model of the business...would be modelled in an OSM...The information system that then tracks the progress [of activities within the OSM]...is handled by [an ISM].

Sutcliffe identifies a library of four highly generic classes of information system, each of which is specified informally [Sut02]. The approach outlined by Sutcliffe to

modeling information systems, however, has two significant limitations. Firstly, neither Sutcliffe [Sut02] nor Maiden [Mai92, SM98] describe a detailed schema through which information system specifications can be represented. A significant aim of requirements engineering, however, is to incrementally formalise knowledge about the application domain so that detailed and precise specifications for a software solution can be written [NE00, Jac01b, Sut02, CA07]. Ideally, these detailed specifications should indicate the algorithms and data structures which the software system will ultimately comprise. The Domain Theory's approach to Information System Modeling is inadequate, therefore, because it does not provide any schema through which such precise specifications of a software system can be represented.

A second limitation lies in the range of software systems which Sutcliffe's conception of an ISM can model. Sutcliffe considers only that class of software system - the Information System - which gathers and reports on information [Sut02]. However, as Jackson notes [Jac01b], a wide range of different kinds of interaction are possible between a machine and its context. In order for the Domain Theory to have general applicability, it should be possible to use the framework to model requirements (and specifications) for software systems which extend beyond simple information gathering and reporting.

Given these limitations, this research was motivated to revisit the ISM concept and to provide a detailed schema - similar to that which Sutcliffe and Maiden define for modeling Object Systems [SM98] - which is capable of representing in detail the specifications for a software system. This schema is presented in Figure 7.3. The schema assumes an object-oriented approach to software development, in line with the assumption which fundamentally underpins the original formulation of the Domain Theory.

An Information System Model (ISM) is designed to provide a detailed software system specification for a single OSM domain. It consists of four sub-models which provide an international (Goal Model), structural (Object Model), behavioural (Process Model) and component (Resource Model) view of the software system. This section now presents each of these models in turn.

The schema consists of four types of model. However, only three of these actually represent the software specification itself:

- **Object Models** describe the software units (object-oriented classes) from which the Model layer of the new software system will be composed;

- **Process Models** describe the business activities which the software system will

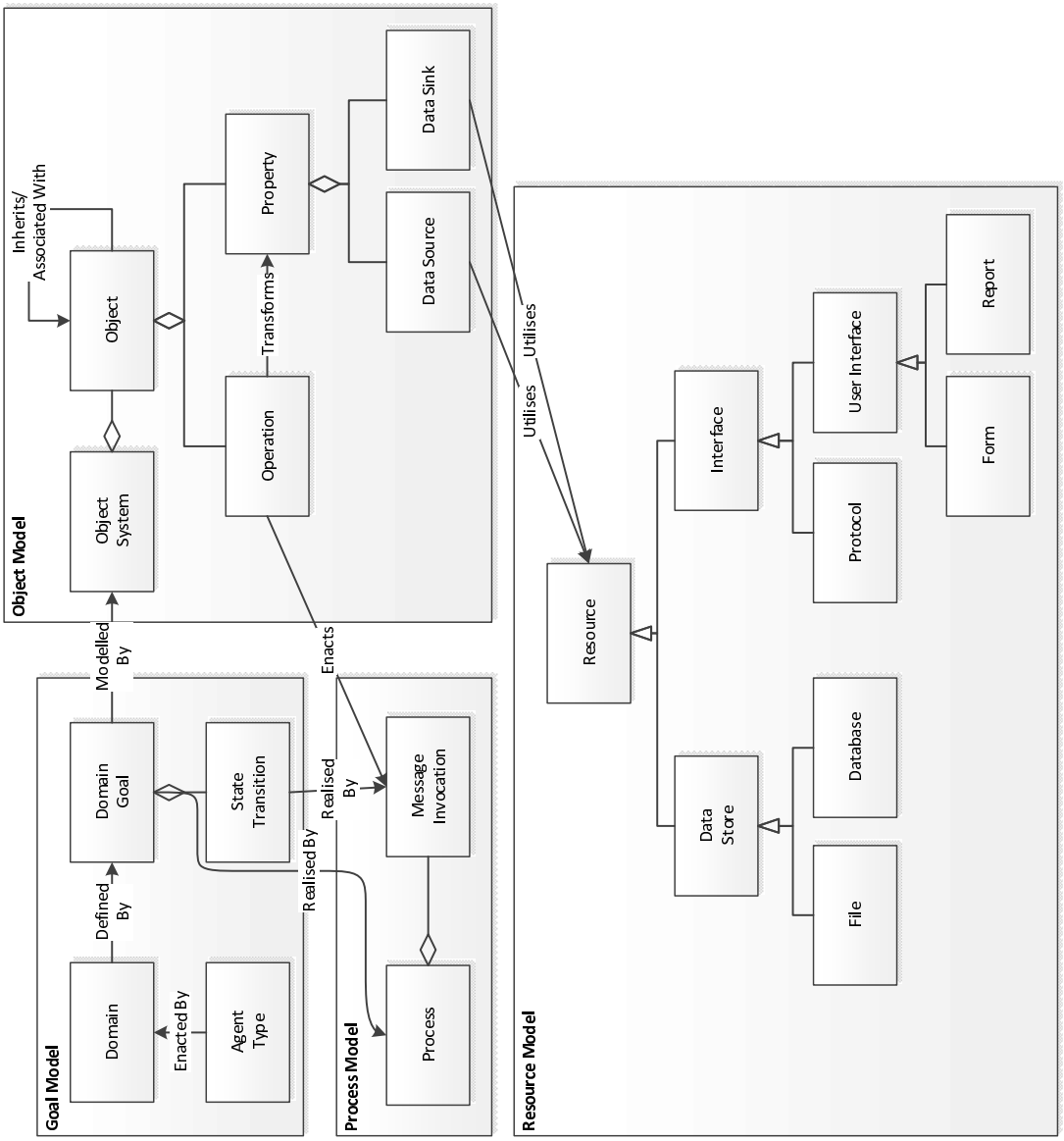

Figure 7.3: A Novel Schema for Modeling Information Systems Specifications

implement in terms of operations over classes defined in the Object Model;

- **Resource Models** specify the low-level resources from which the software system will acquire the data which it processes, and to which the software system will write data.

The fourth model within the ISM schema, the **Goal Model**, is actually a subset of the restructured OSM schema. Each Object System which is defined within an Object Model, and each Process, is associated with precisely one Goal Model. This Goal Model provides traceability between an OSM and an ISM, indicating which ISM specifies software support for each OSM. However, the Goal Model has another role. According to Jackson, software specifications consist in statements about the relationship between a machine and the domains which that machine supports [Jac01b]. In line with this approach, the restructured version of the Domain Theory assumes that OSMs strictly model the application domain and its activities, whereas ISMs model the software system which supports those activities. The **Goal Model** provides the link between these two specifications by defining the relationship that should exist between the application domain (modelled by the OSM) and the software system (modelled by the ISM). The relationship is stated in terms of Agent Types, and in terms of the "Monitored" attribute of the domain fact. Figure 7.4 illustrates this idea.

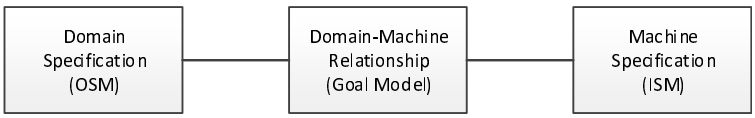

Figure 7.4: The relationship between OSMs and ISMs in the refined Domain Theory

7.3.1 The Goal Model

The Goal Model (see Figure 7.5) is a subset of the restructured OSM schema which Section 7.2 discucssed. This thesis does not, therefore, discuss each fact type in detail. The primary purpose of the Goal Model is to provide a link between the software specification which is represented by an ISM and the domain model which is represented by an OSM. A Goal Model is the first of the models within an ISM to be generated by the novel transformation procedure as it simply involves extracting significant information from an OSM. Furthermore, the Goal Model contains the knowledge that is needed to construct a structural model as well as two pieces of information (the Agent Type and

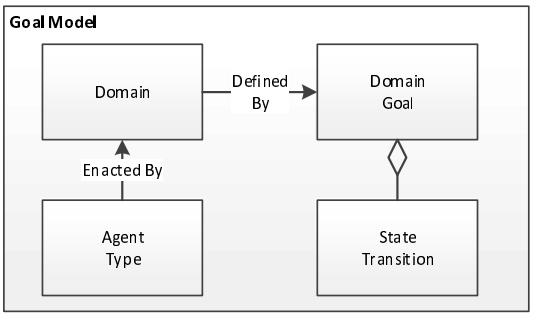

Figure 7.5: The Schema for a Goal Model

the "Monitored" attribute of a domain) which tell the transformation procedure what functionality needs to be specified. The "Monitored" attribute indicates the extent of reporting that needs to be done on the domain represented by a Goal Model, whereas the Agent Type indicates the nature of any additional functional requirements which the transformation procedure should specify. The main function of the Goal Model, therefore, is to aggregate the information which is needed to bootstrap other models within the ISM.

The Goal Model consists of four fact types:

- The *Domain* itself, as discussed in Section 7.2, which contains information about a single atomic aspect of a complex application domain;
- The *Domain Goal* which characterises the predominant behaviour within that atomic domain;
- A set of *State Transitions* which describe in more detail the behaviour represented by the Domain Goal;
- An *Agent Type* which specifies generic functional requirements which must be specified during the transformation procedure.

7.3.2 The Object Model

Figure 7.6 illustrates the schema for an Object Model, which is part of the more general ISM schema. An Object Model is, in effect, a simplified version of a UML class diagram and represents the low-level software units from which the new software system shall be constructed. Its main function within an ISM is to provide a low-level structure for the grouping of data and operations which transform those data. The transformation

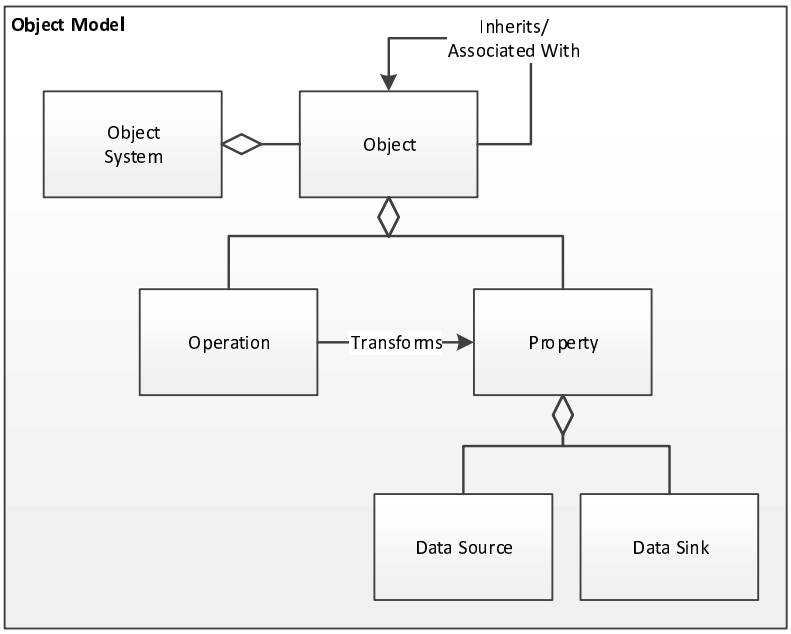

Figure 7.6: The Schema for an Object Model

procedure which is described by Section 7.4 is a domain-driven approach to software specification and so the Object Model in an ISM will, in some sense, reflect the object structure within the OSM from which that ISM was generated. Each object in an ISM Object Model is, therefore, a software class which represents some real or imagined entity in the application domain. Collectively, Objects in an Object Model form the software model which will form the basis of an implementation of the reports and other functional requirements that are mandated by an OSM.

An Object Model consists of five inter-related fact types:

- An **Object System** is a purely structural concept which aggregates the Objects that are derived from each OSM;
- An **Object** models a software class as defined by a high-level object-oriented programming language such as Java;
- *Objects* may be related to one another in one of two ways: by inheritance, which represents a sub-typing relationship between software classes; by association, which represents either functional or data dependency between software classes;
- Each *Object* aggregates a set of **Properties** which represent data that describes that class;

- Each **Object** aggregates a set of **Operations**, or "Methods", which operate over, and thus transform, the *Properties* of that same software class;
- Each *Property* defines a set of **Data Sources** and **Data Sinks**. A *Data Source* indicates where the *Property* gets its initial value from. A *Data Sink* indicates where the data contained within a *Property*. *Data Sources* and *Data Sinks* define mappings between *Properties* of an object, and *Resource* which are defined in the resource model.

7.3.3 The Process Model

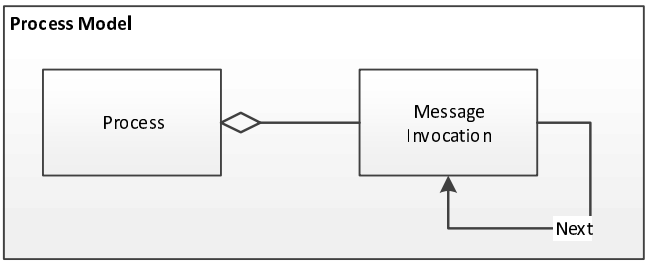

Figure 7.7: The Schema for a Process Model

Figure 7.7 illustrates the schema for the Process Model, which is also a part of the general ISM schema. The low-level operations, from which will be composed the functionality of the to-be software system, are defined in the Object Model. However, the operations which are defined by classes within a software system must typically be composed into coarser-grained, higher-level processes. Within the Object Model itself there is no construct for representing control flow between different class methods. Rather, this is the function of the Process Model, which aggregates the low -level operations that are defined within Object Models into higher-level Processes. This is necessary for two reasons.

Firstly, when an OSM is fully enacted by a Software agent, the OSM represents high-level domain behaviour which may be implemented by several low-level operations as defined within an Object Model. The Process Model provides a mechanism for specifying how these low-level operations should be co-ordinated in order to realise the high-level domain behaviour. Secondly, OSMs can themselves be composed to model more complex domain activity [Sut02]. Process Models thus provide a mechanism through which can be aggregated interactions between the operations which are defined for several OSMs to define ever-more complex domain behaviour.

Process Models are essentially simplified versions of UML sequence diagrams and consist of sequences of messaging passing between operations defined within an Object Model. They consist of two inter-related fact types:

- **Processes** represent high-level units of activity whose behaviour is defined in terms of a sequence of lower-level operations;
- **Message Invocations** represent calls to methods within an Object Model. They consist of a pointer to the operation which is to be invoked, as well as of a pointer to the next *Invocation* in the process.

7.3.4 The Resource Model

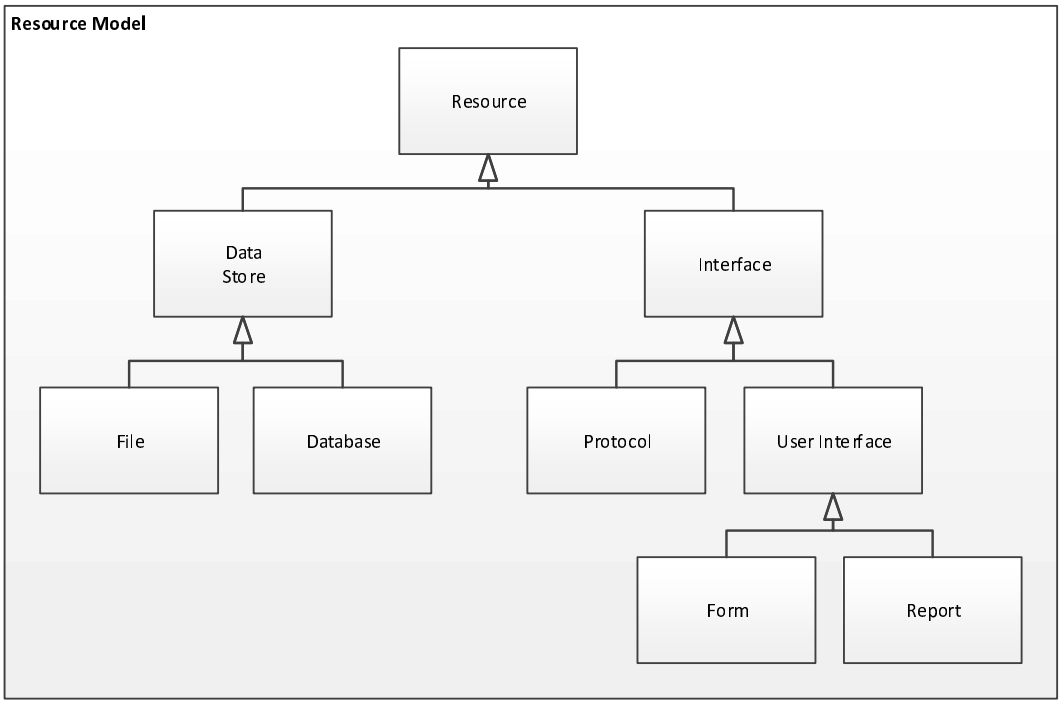

Figure 7.8: The Schema for a Resource Model

Figure 7.8 illustrates the schema for a Resource Model. A Resource Model consists of dictionary of Resources. A Resource is a component of a software system which is capable of persisting, processing or generating data. Resource Models were motivated by the desire to provide some fundamental constructs through which the operations defined in the Object Model could be given a truly formal definition. This thesis notes that fundamentally software consists of two basic kinds of operation:

- **Local Operations** which are typically mathematical and logical operations (as in the Intel 64 processor[Cor12]) directly implemented by the Central Processing Unit (CPU) of the platform on which the software is executed;
- **External Operations** which are implemented by a piece of hardware which is external to the CPU, and is accessed via some interface.

These operations transform data which are loaded into the CPU from some external storage device, and from where the results of the operations are output. Every software system mirrors this basic structure. By defining resources from which a software system will be composed, therefore, Information System Models provide an unambiguous basis through which the operations defined in the Object Model can be given meaning. This is a useful precursor to the task of generating code (which is not attempted in this thesis).

We define two high-level types of Resource:

- **Data Stores** are components which persist data externally to a software system and over an extended period of time;
- **Interfaces** are functional components which either produce new data by some mechanism, or transform data in some way.

We distinguish two types of *Data Store*:

- **Files** are *Data Stores* which represent information in the form of a (possibly well-structured) flat file. This includes XML, CSV and Fixed-Length files. They are defined by a name, and an EBNF specification of their structure;
- **Databases** are *Data Stores* which store information in a Relational Database Management system, and so provide powerful functionality for querying and retrieving data. They are defined by a name, and a set of tuples indicating the structure of each table within the database.

We also distinguish two types of *Interface*:

- **Protocols** are software interfaces to external systems. External systems may include network systems such as Application or Web Servers, or peripheral devices such as Printers, Scanners, or domain-specific peripherals. *Protocols* are specified as a set of operations which are provided by the driver for the external

system, as well as the input and output parameters, and the pre- and postconditions for each operation. This provides a rich set of data about the structure of a Protocol which indicates to a software designer how the *Protocol* can be utilised to implement some desired behaviour;

- **User Interfaces** are Graphical or Command-Line User Interfaces which support interaction between the software system and a human user. Information System Models distinguish two types of *User Interface*: the **Form**, which allows a user to act as a data source for the software system; and the **Report**, which outputs processed information to the user. Both are defined by the mapping from a data structure to a set of user interface components.

7.4 The Transformation Procedure

The procedure for transforming restructured OSMs into ISMs is shown in Figure 7.9. The procedure assumes, as input to the process, a complex domain model comprising a set of Domains, represented through the restructured Domain model.

7.4.1 Specify Object Properties

The first step in the transformation process is to ensure that Object in the input model has the necessary properties specified. This step is necessary for two reasons:

- There is no guarantee that any process for generating OSMs will require the user to specify the properties of objects as, within the Domain Theory's matching approach, structure and not secondary states are the significant concern;

- Subsequent stages of the novel transformation procedure depend on certain properties having been specified, depending on the Domain Goal and Type of the OSM.

To this end, an Elicitation Stimulus is defined which asks the requirements engineer to specify properties for each Object in the each Domain in the input model. The requirements engineer should elicit this knowledge from relevant stakeholders, and so has a degree of discretion in respect of the properties which they can specify.

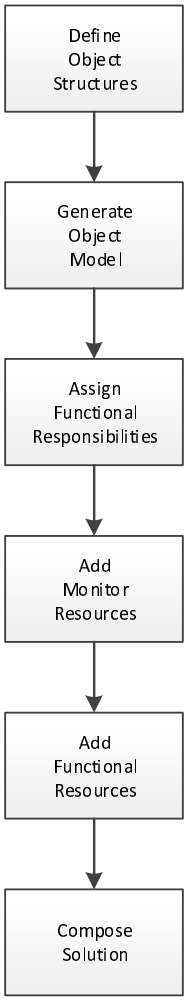

Figure 7.9: The Procedure for Transforming Restructured OSMs into ISMs

7.4.2 Generate Object Model

The next step in the process is to generate an initial form of the Object Model from the current state of the complex domain model. This is done by applying three simple heuristics in the following order:

1. Each *Object* in the input Domain Model becomes an *Object* in the corresponding ISM Object Model;
2. Each *Object Property* for a given *Object* in the input Domain Model becomes a *Property* of the corresponding *Object* in the ISM Object Model;
3. Each *Relationship* connecting two *Objects* in the input Domain Model becomes a relationship connecting the two corresponding *Objects* in the target ISM Object Model.

These heuristics are applied in this order for each *Object* in each *Domain* in the input model.

7.4.3 Assign Functional Responsibilities

For each *Domain Goal* in the input model, a single *Operation* is added to an *Object* in the ISM Object Model. *Operations* are added to the ISM *Object* which represents the OSM *Key Object* whose state is affected by the realisation of the *Domain Goal*. Depending on the kind of *Domain Goal* which the *Operation* enacts, the *Operation* will have one of a number of possible pre-defined *Operation Structures* as specified in Table 7.10:

Additionally, one *Operation* is added to each *Object* representing an *Agent* for each *Domain Goal* which that *Agent* is responsible for enacting. Agent Operations are shown in Table 7.11.

7.4.4 Add Monitor Reports

This step produces a set of *Reports* which support the monitoring of each *Key Object* in each *Domain* which has its "Monitor" attribute set to *true*. For each *Key Object* in each such *Domain* a *Report* resource is added to report on the state of the *Key Object*. The *Report* is structured such that it displays a list of instances of the *Key Object* and, for each instance, displays both the primary and secondary state of that instance.

Goal Type	Operation Name	Input Parameters
Allocation	allocateTo	*targetStructure* - An ISM Object representing a structure object in the corresponding OSM
Control	issueCommand	*command* - An ISM Object representing a key object which, in turn, represents a command to the controlled object
Composition	addComponent	*component* - An ISM Object representing a key object which, in turn, represents the component that is to be added to the composition
Decomposition	removeComponent	*component* - An ISM Object representing a key object which, in turn, represents the component that is to be removed from the composition
Manipulation	manipulateProperty	*property* - An ISM Property representing the property of the ISM object whose value is to be manipulated; *value* - Any object which is a valid value for *property* and represents the value which should hold true for the to-be-manipulated property once the manipulation is complete
Sensing	sensePropertyValue	*property* - An ISM Property which is the property of the sensed key object whose value is to be sensed
Transfer	transferTo	*targetStructure* - An ISM Object representing a structure object in the corresponding OSM

Table 7.10: Rules for adding operations to ISM objects representing key objects which are manipulated in order to realise Domain Goals

Goal Type	Operation Name
Allocation	allocateObjects
Control	commandObjects
Composition	composeComponents
Decomposition	decomposeComposition
Manipulation	manipulateObjects
Sensing	senseObjects
Transfer	transferObjects

Table 7.11: Rules for adding operations to ISM objects representing agent objects which manipulate Domain Goals

7.4.5 Select Generic Algorithm

As part of the restructuring process, this thesis also identified a set of generic algorithms. During the Algorithm Selection step of the transformation procedure, a generic algorithm is matched to each *Domain* whose *Agent Type* is either a "peripheral device" or "software". Facts within a *Domain* are tested by rule against candidate algorithms in order to identify the algorithm which best satisfies the specific problem which that *Domain* represents. The *Generic Algorithm* is then selected and linked to the functional operations within that *Domain*.

Generic Algorithms are selected by rule-based classification which uses facts within an input OSM to determine which Generic Algorithm would best satisfy the requirements which that OSM represents. The permutation of *Domain Goal* and *Domain Type* which the OSM embodies significantly narrows down the range of candidate structures, while state transitions and properties over key facts narrow candidates down to a final selection. Each Generic Algorithm defines the conditions under which it is applicable.

7.4.6 Configure Functional Resources

Some Generic Algorithms may introduce *Resources* into the ISM Resource Model. However, it will typically not be possible automatically to specify the structure of *Resources* that are generated by *Generic Algorithm*. This is because when designing a *Generic Algorithm* it is often possible to identify the *type* of *Resource* with which an algorithm must interact, but the particular structure of that *Resource* will generally vary dependent on context. *Resources* that are not fully specified (because they lack important structural properties) need to be configured in order for the ISM to be complete. To this end, a set of Elicitation Stimuli are defined which allow the user to more fully specify incomplete *Resources*. Different stimuli are defined to support the specification of different kinds of *Resource*.

7.4.7 Specify Data Sources and Sinks

The previous three steps have focused on applying a range of heuristics to generate *Data Sources* and *Data Sinks* for a range of *Objects*. However, it may be necessary for the requirements engineer to specify sources and sinks which were not introduced by these heuristics. Therefore, an Elicitation Stimulus is defined to enable the requirements engineer to specify additional *Sources* and *Sinks* for each *Object* in the complex domain model which lacks these resources.

7.4.8 Compose Process Model

Once the Object and Resource Models have been defined for the input Domain Model, the operations within the Object Models can be composed into a coherent process. Process Models are composed from the operations which are defined over *Agent*-representing *Objects* in the Object Models. Jackson proposes an approach to the composition of domains based on phenomena shared between domains [Jac01b], and the transformation adopts this notion through an approach to composing OSMs based on the passing of *Key Objects* between OSMs. An Elicitation Stimulus is provided which asks users to indicate the passing of *Key Objects* from one OSM to another. The response to this Elicitation Stimulus is in the form of a 2-tuple in which the first attribute of the tuple represents an OSM which passes a *Key Object* to the OSM represented by the second attribute. The object-passing dictionary which is defined by a requirements engineer in response to this stimulus is used to produce the Process Model. The Agent operations for each OSM are added to the process model, and then for each entry in the object passing dictionary, the Agent operation which is defined for the second OSM in the entry is set as the "next" operation for the Agent operation which is defined for the first OSM in the entry. In this way, the Process Model is iteratively constructed.

7.5 Summary

This chapter has demonstrated the knowledge engineering perspective of RORE by showing how a requirements engineering activity, and its support modeling notations, can be formalised in the Meta Layer of RORE. Firstly, this chapter presented a refined version of Sutcliffe and Maiden's schema for representing domain knowledge [SM98, Sut02]. Secondly, this chapter presented a novel modeling schema for expressing software system specifications: Information System Models. Information System Models as presented in this thesis expand on, and give structure to, the concept as discussed by Sutcliffe [SM98]. Appendix B illustrates how both modeling notations have been formalised using the knowledge structures which are defined in the Immutable Layer (see Appendix A). As such, the examples presented in this chapter - as realistic, and complex examples of requirements modeling notations - provide a demonstration of how RORE can be used to formalise requirements modeling notations.

Finally, a step-by-step procedure was described for transforming refined Object System Models into Information System Models. Each step of this process was described, and a brief description was given of how each step could be modeling as a

reusable knowledge structure in RORE. Appendix B presents the formalisation of these steps as RORE knowledge structures, using OWL syntax. This chapter also shows, therefore, how reusable knowledge structures can be used in RORE to formalise requirements engineering activities as a sequence of reuse-driven transformations.

Chapter 8

Applying Reusable RE Knowledge: The RE Viewpoint

8.1 Introduction

Chapter 7 demonstrated the Knowledge Engineering perspective (see Section 3.5.2) by showing how RORE can be used to build a long-term memory by formalising a procedure for transforming Object System Models into Information System Models.

This chapter complements that demonstration by illustrating the Requirements Engineering perspective. This chapter demonstrates how the long-term memory knowledge base that was generated in Chapter 3.5.2 can be used by a requirements engineer to generate new model instances by reuse. To this end, this chapter presents three concrete examples taking both from the software engineering literature and from this author's experiences of software development, and show how RORE's Requirements Task Assistant can be used to produce ISMs for each scenario. The three scenarios are:

1. The Autopilot Example;
2. The Order Management Example;
3. The Online File Transfer Example.

The following sections present each of these examples in turn. Each example is presented in four parts:

1. A brief textual description of the scenario is given;
2. A graphical representation of the refined OSM is shown for that scenario;

3. A step-by-step description of the application of the transformation procedure (see Chapter 7) to the OSM is given;

4. The resultant ISM is presented and briefly summarised.

The graphical representations of both the OSMs and ISMs for each of the examples in this Chapter were generated automatically by a prototype visualisation feature of RORE. This feature generates basic graph-based representations of RORE models which illustrate facts as nodes and properties of facts as arcs. Because of the complexity of some of the models which were input to, and produced by, RORE, the visualisation of some of these diagrams is not clear. Nonetheless, these diagrams are presented in this thesis because they provide visual validation of the input to and output from the RORE process in each case. As a result of the difficulty in presenting these models clearly within the space constraints of this thesis, a textual description is given — in order to assist readers — for diagrams which are not clearly visible.

8.2 The Autopilot Example

8.2.1 The Autopilot Scenario

The Autopilot Scenario is taken from Coad [CNM95]. Andi's small business repairs and upgrades small aircraft, and is looking to introduce a novel, low-cost autopilot system into the market for her hobbyist clientle. Andi has done some research into autopilot systems, and has discovered that an autopilot system consists of two basic kind of component:

- A gyroscope which measure the angle of an object (a plane in this case);
- A controller which interprets instructions in order to manipulate mechanical aspects of the plane.

A plane moves in three basic directions; along each of three axis. Accordingly, the autopilot system must collect information about the angle of the plane along each of these axis, and adjust that angle in line with some assumptions about the desired flight plan. The autopilot system therefore requires three gyros - one for each axis along which the plane rotates - and three controllers. Each gyro should feed information to the appropriate controller which should then adjust the angle of the plane accordingly. For performance reasons which are discussed in Section 9.3.4.1, however, this example models just the X and Y axis of the plane.

8.2.2 The Autopilot OSM

The prototype RORE implementation is capable of automatically generating basic graphical representations of formal RORE models. Figure 8.1 illustrates the auto-generated Object System Model for the Auotpilot Scenario.

This model stipulates that for two axis of an aeroplane (X and Y) there exists a gyro and a controller. The same basic pattern is repeated for each axis, and so the Z axis was not included in this example. There are two Domain facts for each axis which is represented: a Gyro domain (XAngleGyro, YAngleGyro) and a Controller domain (XAngleController, YAngleController). All four Domains share the Physical domain type. The two Gyro domains share the Sensing goal type and the two Controller domains share the Control domain type. Each of the four domains has a single Agent object, of type Peripheral Device in each case, which represents: the Gyro itself, for Gyro domains; and the Controller for the Controller domains. A key object, representing the Plane, is also shared across all four domains and is associated with two properties which represent the XAngle and YAngle of the plane respectively. The Initial state for the OSM captures the initial values of both XAngle and YAngle. For each of the Gyro domains, state transitions capture the sensing of a change in the angle of the plane along the corresponding axis (X or Y respectively), and Goal states capture the result of this angle change. For each of the Controller domains, state transitions capture the dispatch of commands to the Controller agent in order to change the plane angle along the corresponding axis, an Event captures the reception of this command and a Goal state captures the fact of the resulting change in the angle of the Plane.

The four domains map onto a refined OSM context as follows:

- A ⟨**Sensing, Physical, Peripheral Device**⟩ domain, each modeling the gyro for a single axis. These domains have a key object which represents the aeroplane. The key object has a single property representing its angle of rotation along each axis. Events from the gyro to the plane notify the plane of its angle change;

- A ⟨**Control, Physical, Peripheral Device**⟩ domain, each modeling the controller for a single axis. These domains also have a key object representing the plane with properties representing the angle of rotation along each axis. In this case, however, events from the plane to the controller inform the controller of a change in angle so that the controller can respond accordingly.

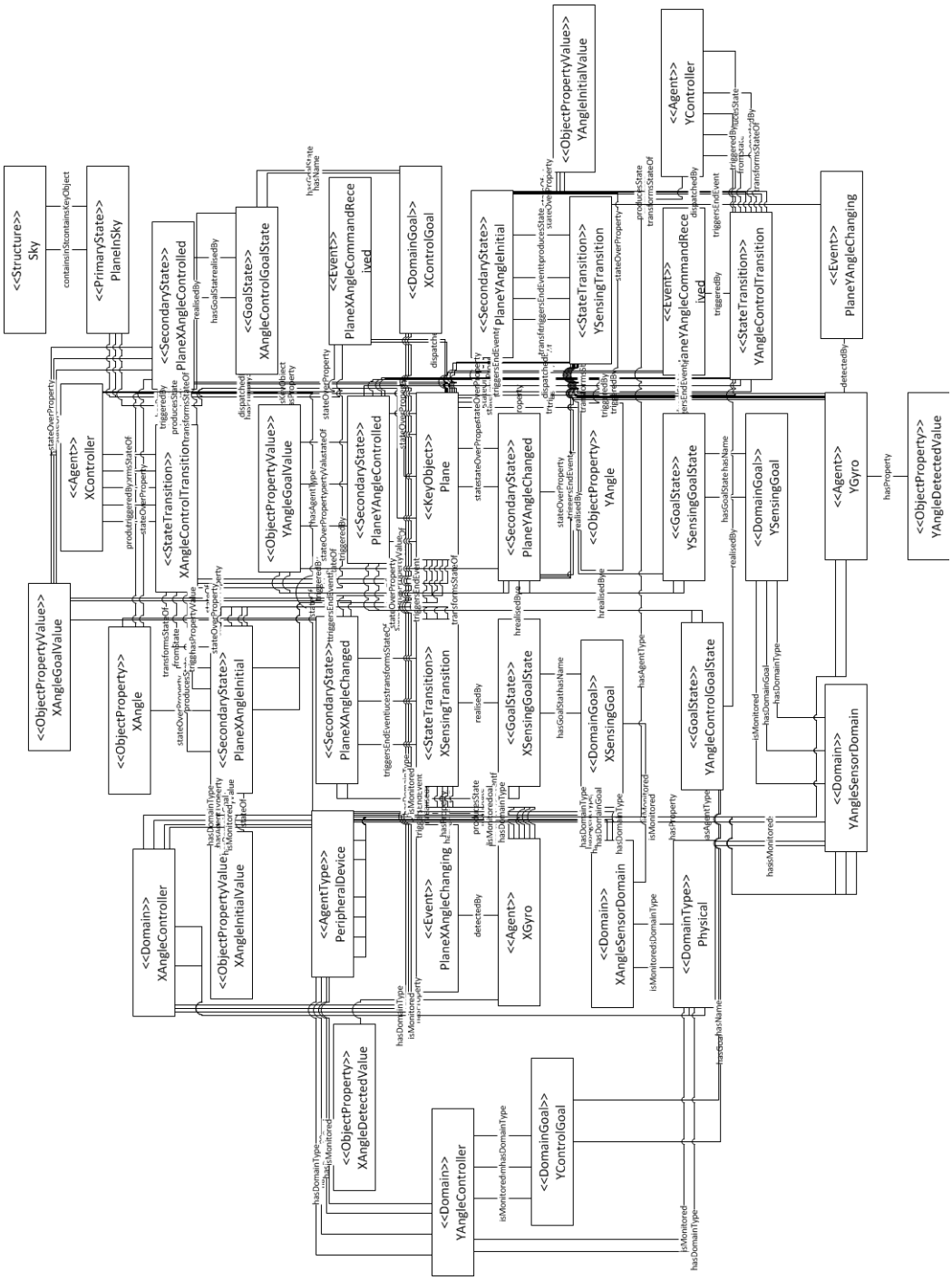

Figure 8.1: The Auto-generated Diagram of the RORE-formalised Autopilot OSM

8.2.3 Producing the Autopilot ISM

This section now briefly summarises the steps by which RORE transformed the Autopilot OSM into an Autopilot ISM. The tool generates detailed logs of its activity, and the Autopilot Spec Generation log is presented in Appendix C.

Cycle One: Generating System Classes Analysis failed on the first test: "HasISMObjectsRule". This rule is a broad rule which tests to see whether any ISM objects (representing system classes) exist within the target model. Accordingly an information requirement was generated which mandated that cycle 1 should focus on producing system classes as input to the target model. Chunk-based Inference was attempted and model chunks were identified as potentially relevant. However, closer manual inspection confirmed that they were not ideally applicable to the empty model, as they contained floating operations (unattached to any containing class). RORE therefore progressed to Rule-Based Inference where matching identified one potentially-relevant Production Script ("GenerateISMObjects"). RORE fired this, as manual inspection confirmed that it would satisfy the Information Requirement (IR), and the result was that a single system class was generated for each object in the source model, and for each property of a source object a corresponding target property was also generated. This produced the basic structure of the ISM Object Model.

Cycle Two: Generating Monitor Reports During completeness analysis the "HasISMObjectsRule" was refired to confirm that the information requirement from cycle 1 had successfully been satisfied and the test was indeed passed. However, analysis now failed on a second rule "HasMonitorReportRule" which checks to see whether reports exist in the ISM for each domain within the input OSM for which the monitor tag is set to "true". Reports are user interface forms which report on the state of data within the software system. On the failure of the target model to satisfy this rule, an information requirement was generated mandating the generation of such reports for the input source model. Chunk-based inference was skipped in this case because the user was already aware of a production script which would resolve this information requirement. The user therefore instructed RORE to retrieve relevant production scripts and opted to fire the "GenerateMonitorReports" script. The input source model contains a single key object - the Plane object. The production script therefore generated two reports:

- The first report, **PlaneListReport**, represents a list of all planes about which the

software system has information;

- The second report, **PlaneReport**, reports on the state of a single aircraft from this list.

Cycle Three: Assigning Functional Responsibilities In addition to the "HasISMObjectsRule", the "HasMonitorReportRule" was also fired successfully in cycle 3. This time, however, the "HasFunctionalResponsibilitesRule" (the term is taken from Larman [Lar04]) - which checks to see whether operations have been appropriately assigned to system classes in the ISM - failed. Accordingly, an information requirement was specified mandating that operations should be added to system classes in the ISM as necessitated by the goals defined in the OSM. In order to satisfy this information requirement, chunk-based inference was attempted. However, while this returned several relevant model chunks, no chunk would fully satisfy the specified information requirement. RORE therefore progressed to rule-based inference, which identified one relevant production script: "AssignFunctionalResponsibilities". This production script consists of two production rules which add relevant command operations [Mey88] to the ISM classes which represent respectively the Key Objects and Agent Objects within the OSM. The command method for each system class representing a Key Object commands instances of that class to change state in the manner specified by the goal in the associated source OSM domain. The command method which is assigned to each system class representing an Agent Object invokes the Key Object command over all instances of classes representing key objects. Accordingly, for the Autopilot domain, the following operations were generated for each class:

- **PlaneClass**: issueCommandToPlane(), which accepts a command from an agent-representing system class to modify the specified property of the plane in the specified way; sensePropertyOfPlane(), which detects the value of the specified property for the given plane;

- **XGyroClass**: sensePlaneObjects(), which retrieves the value of the XAngle property from each Plane object in the autopilot system by invoking the sensePropertyOfPlane() method;

- **XControllerClass**: commandPlaneObjects(), which commands each Plane object to adjust its XAngle by invoking the issueCommandToPlane() operation;

- **YGyroClass**: sensePlaneObjects(), which retrieves the value of the YAngle property from each Plane object;

- **YControllerClass**: commandPlaneObjects(), which commands each Plane object to adjust its YAngle.

Cycle Four: Selecting a Generic Algorithm During this cycle, the target model passed all of the analysis rules which have failed in previous cycles. However, the target model failed the "HasDataSources" test which checks whether or not the data sources for properties of system classes have been properly specified. An information requirement was generated mandating that the current cycle should attempt to link properties to appropriate data sources in the ISM, and to this end the user attempted a round of chunk-based inference. One model chunk was retrieved (the "DataSampleSensingAlgorithm") which appeared a plausible candidate to resolve the information requirement. RORE fired this but did not make any modifications in order to refine the algorithm. Instead, during the integration phase this research paired the "senseAllProperties" method in the chunk to the existing "sensePlaneObjects" method in the ISM, and the "senseProperty" method to the ISM method "sensePropertyOfPlane". This integrated generic algorithms for each of these two ISM methods into the ISM.

Cycle Five: Selecting a Generic Algorithm The generic algorithm which was applied in the previous cycle did not, however, resolve the information requirement. In the fifth round, therefore, RORE fired the "HasDataSources" rule again and again it failed. The user therefore returned to chunk-based inference for a second attempt at resolving the resultant information requirement. This time RORE applied the "NotifiedSampleControlAlgorithm" which represents event-based Control operations, where commands are dispatched in response to discrete events. This resulted in a ControlInterface resource being integrated into the ISM to represent the communications protocol between the autopilot system and the X and Y controller devices. A NotifiedSampleCommand class was also introduce to the ISM object model to represent the commands which are issued by the autopilot system to these components. Finally, the user paired the "dispatchAllNotifiedSampleCommands" operations to the "commandPlane" method, and the "dispatchNotifiedSampleCommand" operation to the "issueCommandToPlane" operation. This resulted in generic algorithms for each method being integrated into the existing ISM.

Cycle Six: Configuring Functional Resources Again, the chunk-based inference method introduced valuable new information into the ISM but did not succeed in satisfying the information requirement. The user therefore fired the "HasDataSources" rule for a third time and again it failed. The user attempted both chunk- and rule-based inference, but in neither case was any further component identified which looked promising and which had not previously been fired. RORE therefore progressed to elicitation where matching identified the "FunctionalResourceConfiguration" stimulus. This elicitation stimulus is a fact-editing stimulus which can be used to configure the structure of resources. However, the user did not make any significant modifications to the current structure of either the SensorInterface or ControlInterface resources.

Cycle Seven: Specifying Data Sources and Sinks Still the "HasDataSources" test was not passed by the autopilot ISM model. As in cycle six, no appropriate model chunk or production script was identified. The user therefore attempted Elicitation as a means to resolve the information requirement. Matching identified a single relevant elicitation stimulus - "DataSourceAndSinkStimulus" - which is a fact-editing stimulus that supports the editing of the data sources and sinks of properties over objects defined within the ISM. The user used this stimulus in order to set the data source and sinks of the XAngle and YAngle properties of the Plane class. The data source in each case was the SensorInterface resource, whereas the data sink was the ControlInterface. Finally, RORE integrated these new facts into the ISM model by pairing the PlaneClass fact which was produced by executing the stimulus with the PlaneClass fact in the ISM.

On a further round of analysis, all tests were passed and so no further action was taken.

8.2.4 The Autopilot ISM

The result of this process is a substantial (but incomplete) Information System Model of the resources, software system classes and algorithms that would make up the Autopilot software solution. This illustrates the role which RORE can play in bridging the gap between requirements and architectural design. An overview is given in Figure 8.2:

It should be noted that the auto-generated diagrams do not display simple (string, integer, or boolean) facts as while these generally represent useful low-level information, they contain a level of detail that is not conducive to a diagram which is intended to provide a brief overview of the structure of a specification. For this reason, while

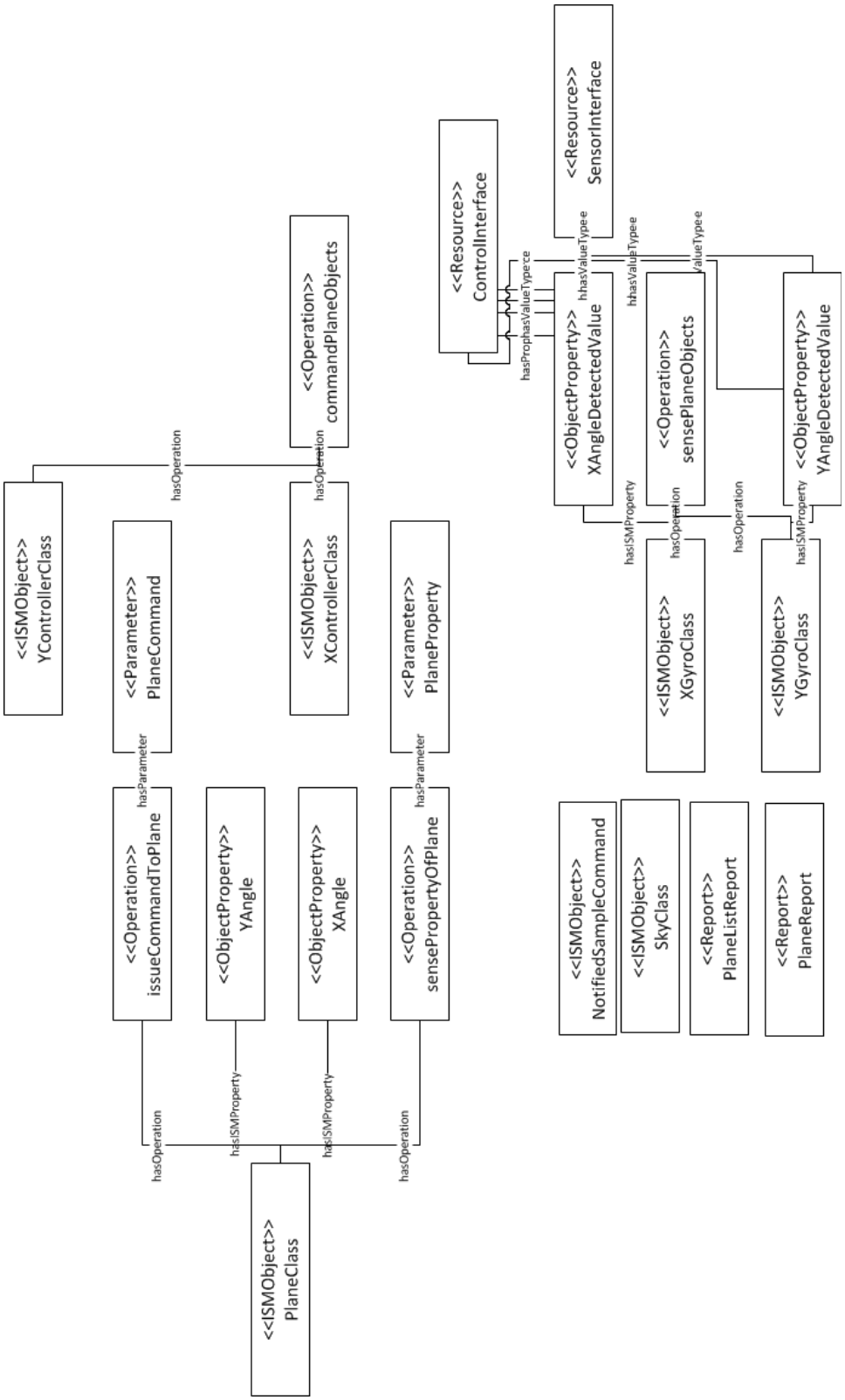

Figure 8.2: The Auto-generated Diagram of the RORE-formalised Autopilot ISM

key facts such as generic algorithms are described in the logs and full formalisations, they are not displayed in the model.

8.3 The Online File Transfer Example

The following brief example is a common-enough problem, but provides just a brief glimpse into some of the difficulties encountered during the production of this thesis.

8.3.1 The Online File Transfer Scenario

James is a Doctor of Philosophy (PhD) student at a well-established British University. As such, he spends much of his time producing documents addressing a range of fascinating - if arcane - topics for consumption by a similarly-minded international audience (he hopes). Spending, as he does, much of his life indoors and at a computer screen, James appreciates the occasional change of scene. He is, therefore, in the habit of working variously from a small home office and a better-sized office on his University's campus. However, on those occasions that James sees fit to make a switch from one office to another, he encounters a problem. His thesis - and related materials and code - amount to several gigabytes of data, and he will usually refer to a sizeable portion of this data during any single thesis-writing session. It is, therefore, imperative that James have some means to transfer his wealth of knowledge between offices as necessary. He has, in the past, attempted to use rewritable CDs and DVDs to this end. However, James' home computer only has the ability to burn such discs, and he was therefore compelled to use a brand new disk each time he changes location - a costly solution, given the recent termination of his research funding! Similarly, he has considered the use of so-called "pen drives". However, confessing as he does to a degree of absent-mindedness, this also proved too costly a solution as such devices needed frequent post-misplacement replacement. James therefore turned to an old friend for help: the Internet.

Fortunately, James has access to a not-insubstantial volume of online webspace left: a relic of his ill-fated venture into freelance website development. Both he, and the University, also have a reasonably speedy internet connection. James envisions that his data-transfer woes could be resolved if only there were some way for him to package up his thesis-relevant files and transfer them to his webspace, from whence they can be downloaded on arrival at his next destination. He vaguely recalls that his webhost

allows him to use the File Transfer Protocol (FTP) - a standard protocol for its stated purpose - to upload files to, and download files from, his webspace. However, he also recalls this being a time-consuming process for large operations if he uses his existing FTP client. He would therefore like an application which was better specialised to his particular thesis-related needs. In particular, he would like a tool automatically, at the touch of a button, to grab files automatically from pre-defined "thesis-relevant" local directories and upload them to relevant directories on his webspace. His PhD work just might be able to help him capture the requirements for such a tool!

8.3.2 The Online File Transfer OSM

Figure 8.3 illustrates the Object System Model for James' online file transfer problem.

The domains in the File Transfer example map onto the following OSM contexts:

- The first domain in each case is an ⟨**Allocation, Conceptual, Software**⟩ domain in which the software determines by applying some pre-specified schema which files are "relevant" to James' thesis, and thus which files should be downloaded. In doing so, the software system *allocates* files in the source location to space in the target location. For the upload process, the source is the local file system and the target is the remote one. For the download process, this scenario is precisely reversed. These two domains consist of a set of key objects representing the files to be transferred, as well as two structure objects representing the source and target file systems. The agent object represents the software system itself which will be responsible for forming this allocation;

- The second domain in each case is a ⟨**Transfer, Conceptual, Software**⟩ domain which represents the transfer of each file from its source to its allocated target space. This process realises and so fulfils the *allocation* which was determined in the first step. The structure of these domains is strongly similar to the structure of the *allocation* domains. The *transfer* domains also contain a set of key objects representing the files to be transferred, and two structure objects representing the source and target file systems. The agent for the *transfer* domain also represents the software system which is responsible for the transfer. However, the *transfer* domains are distinguished from the *allocation* domains by a single event which indicates the completion of the transfer operation.

The upload process thus consists of an *allocation* operation during which local files are allocated to space on the remote file system. This is then followed by the *transfer*

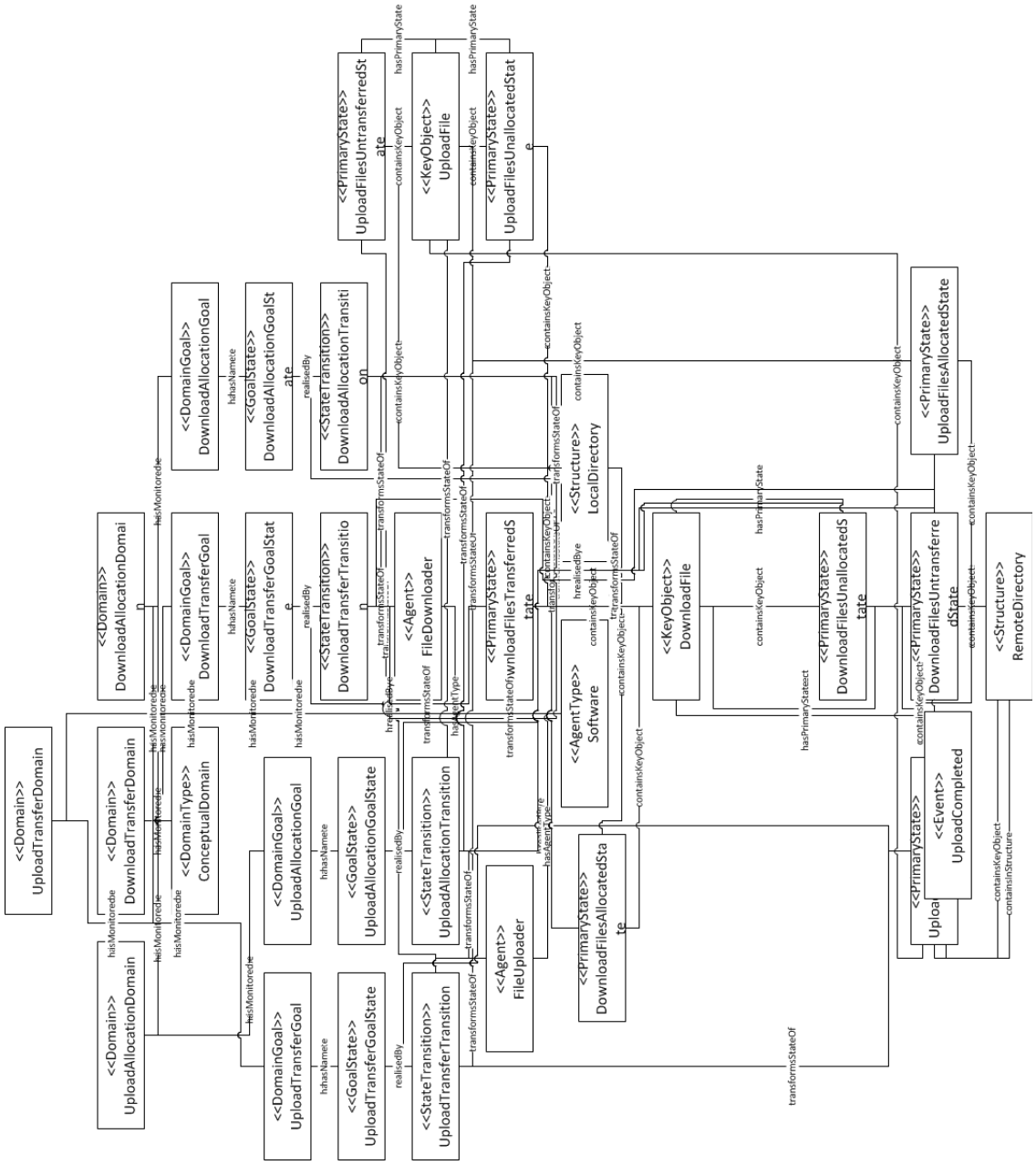

Figure 8.3: The Auto-generated Diagram of the RORE-formalised File Transfer OSM

operation during which the local files are actually uploaded to their allocated remote space.

In inverse, the download process consists of an *allocation* operation which allocates remote files to space on the local file system. This is followed by the transfer operation during which the files are downloaded to their allocated local space.

8.3.3 Producing the Online File Transfer ISM

In sum, the steps which were taken in order to produce the Online File Transfer ISM specification from the input Online File Transfer OSM requirements model are as follows.

Cycle One: Generating ISM Objects The "HasISMObjects" analysis rule was fired and failed, thus indicating that the goal of cycle one should be to generate the system classes that should comprise the basis for the ISM Object Model. Chunk-based Inference was attempted and several model chunks were returned. However, careful manual inspection of these model chunks indicated that while they defined a number of resources and operations, they did not define the ISM Objects associated with those operations. As such, none of the returned chunks would satisfy the information requirement for this cycle. RORE therefore progressed to attempt rule-based inference. This returned the "GenerateISMObject" production script, which was executed and generated the following objects (one for each object in the input File Transfer OSM):

- DownloadFileClass;
- RemoteDirectoryClass;
- UploadFileClass;
- FileUploaderClass;
- FileDownloaderClass;
- LocalDirectoryClass.

With no existing facts in the ISM, these objects were integrated directly.

Cycle Two: Generating Monitor Reports On the second cycle the "HasISMObjects" rule passed, but the "HasMonitorReports" rule failed. This generated an information requirement to the effect that a set of reports should be generated in the ISM for each key object in the input OSM. The user attempted chunk-based inference, but no chunk produced a report and chunk-based inference was not suitable for resolving the information requirement therefore. RORE progressed to rule-based inference. The matcher identified one production script - "GenerateMonitorReports" - which was ideally suited for the task. RORE fired it, but no Monitor Reports were produced. Manual inspection of the production script revealed that this was because monitor reports were generated only for those domains whose "Monitor" tag was set to "true".

Cycle Three: Assigning Functional Responsibilities We therefore returned to analysis. Inspection of the "HasMonitorReport" rule indicated that the implementation of the report mismatched the "GenerateMonitorReport" production script because the former examined only whether reports existed within the target ISM, and did not take into account the status of the "Monitor" tag. The user therefore chose to ignore this production rule throughout the remainder of this scenario. Instead RORE progressed to fire the "HasFunctionalResponsibility" rule, which failed. This resulted in the generation of an information requirement which mandated that the current cycle should focus on assigning responsibility to classes within the ISM for the implementation of the various state transitions modelled within the input File Transfer OSM. While operations were defined within the various model chunks that were returned during Chunk-based Inference, these operations were not associated with any classes, and so would not satisfy the information requirement. RORE therefore progressed to rule-based inference, during which the matcher returned the "ObjectModelGeneration" script. The user fired this and two operations were generated to represent the major state transition for each domain in the input OSM:

- **DownloadFileClass**: allocateToDownloadFile, transferDownloadFileTo;
- **UploadFileClass**: allocateToUploadFile, transferUploadFileTo;
- **FileDownloaderClass**: allocateDownloadFileObjects, transferDownloadFileObjects;
- **FileUploaderClass**: allocateUploadFileObjects, transferUploadFileObjects.

During integration the user paired each of the classes which were returned by the production script with their corresponding (namesake) classes in the target ISM and applied the integration, thus associating the generated operations with the appropriate ISM classes.

Cycle Four: Selecting a Generic Algorithm The "HasISMObjects" and "HasFunctionalResponsibility" rules passed, and the user again ignored the "HasMonitorReports" rule as previously stated. RORE was therefore instructed to fire the "HasDataSources" rule which failed. The user attempted chunk-based inference and identified two generic algorithm chunks which would introduce the resources which would ultimately act as the data sources and sinks for the key data within each domain. The user selected to apply the first of these chunks, the "ClassificationAlgorithm" which represents rule-based allocation. The result was to produce a new ISM object - the AllocationRule - and a new resource fact - the AllocationRuleDataSource resource object. Generic algorithm pseudocode was also produced for each of the allocation operations in the target ISM, and by pairing the operations within the model chunk as shown in the log, the user was able to assign the correct pseudocode to the correct operation in the target ISM model.

Step Five: Selecting a Generic Algorithm We again attempted the "HasMonitorReports" analysis rule, which again failed. RORE thus progressed onto chunk-based inference and on this occasion identified a model chunk - "CompleteRemoteTransferAlgorithm" - which represents the transfer of complete data structures or files across a remote network. The previous cycle had generated pseudocode algorithms for the allocation element of the File Transfer application, but not for the actual data transfer aspect of the application. The user therefore applied this CompleteRemoteTransferAlgorithm in order to generate such code. The result was a new resource - the RemoteDataTransferProtocol - as well as pseudocode for each of the transfer operations that already exist in the ISM domain. The user paired the operations from the model chunk with the transfer operations in the target ISM as shown in the log for this scenario, and the result was that the pseudocode defined in the model chunk was assigned to each of the file transfer operations in the ISM.

Step Six: Configuring Functional Resources We attempted to fire the "HasDataSources" rule for a third time, but again it failed. No further model chunks or production scripts were identified that were relevant to the current RORE context, and so

RORE progressed to elicitation. One Elicitation Stimulus - "ConfigureFunctionalResources" - looked potentially promising. RORE fired this, and the user was prompted to reify each of the resource objects within the ISM. However, the user did not identify any significant reifications that needed to be made at this stage, and so simply reintegrated the facts presented by the stimulus as-is back into the ISM.

Step Seven: Specifying Data Sources and Sinks We fired the "HasDataSources" analysis rule again and again it failed. Again, no further model relevant chunks or production scripts were identified and so RORE progressed to Elicitation again. This time the user attempted the "ConfigureDataSourcesAndSinks" stimulus. This provided an opportunity to set the data source and sink for each property that was defined in the current state of the ISM. The user made the following assignments:

- **AllocationRule**: The user set the data source of the AllocationRuleAntecedent and AllocationRuleConsequent properties to be the AllocationRuleDataSource resource;
- **DownloadFileClass**: The user set the data sink of the "DownloadData" property to be the RemoteDataTransferProtocol;
- **FileDownloaderClass**: The user set the data source of the "DownloadStatus" property to be the RemoteDataTransferProtocol;
- **UploadFileClass**: The user set the data sink of the "UploadData" property to be the RemoteDataTransferProtocol resource;
- **DownloadFileClass**: The user set the data source of the "UploadStatus" property to be the RemoteDataTransferProtocol resource.

By pairing the class associated with each property with its namesake class in the ISM, the user was able to integrate these relationships into the ISM.

We fired all analysis rules again (excluding the "HasMonitorReport" rule) and each rule passed indicating the model was completed (when the user took into account that the HasMonitorReport rule should be ignored).

8.3.4 The Online File Transfer ISM

The Online File Transfer ISM is shown in Figure 8.4.

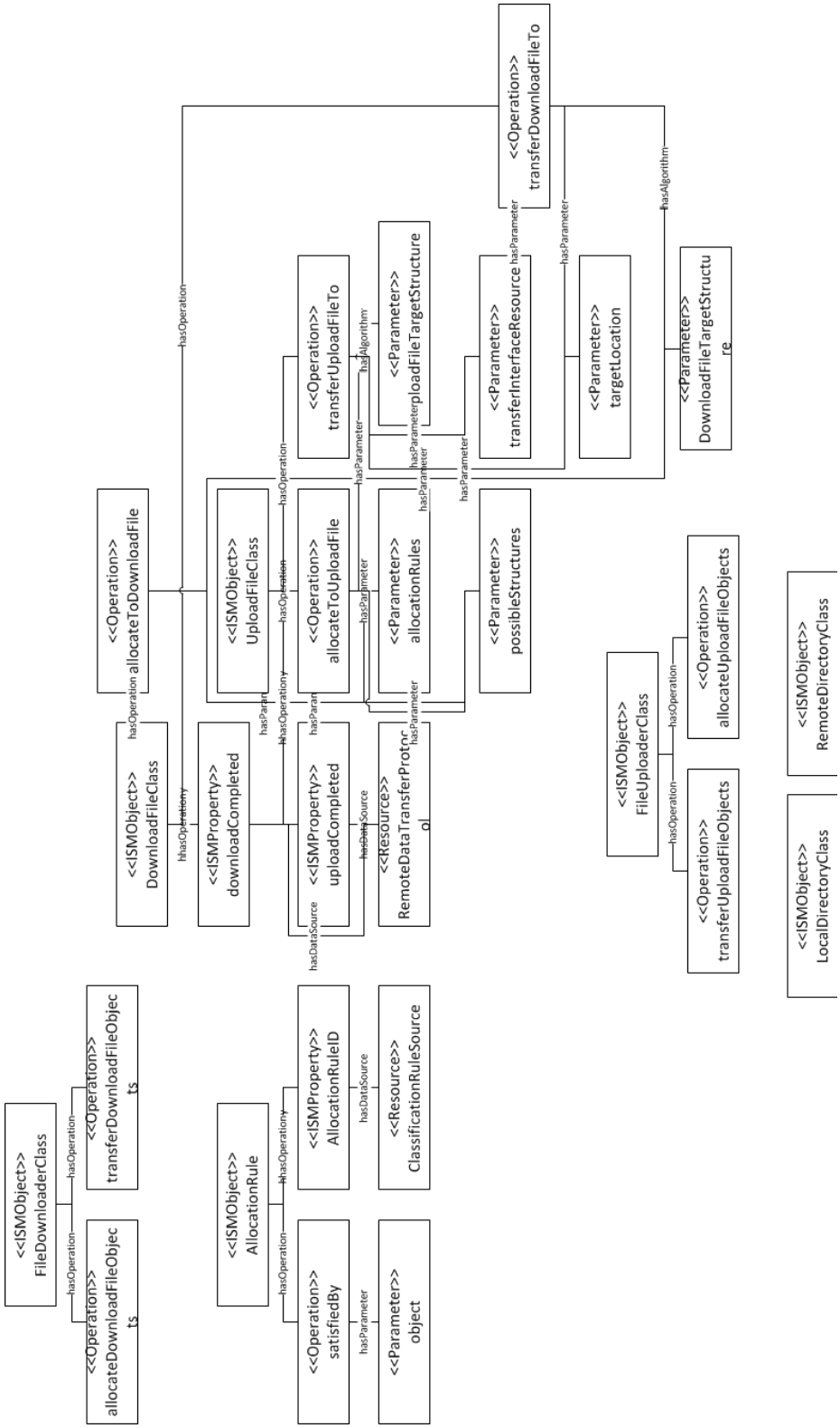

Figure 8.4: The Auto-generated Diagram of the RORE-formalised File Transfer ISM

8.4 The Order Management Example

The Order Management example is taken from a practical project which the author of this thesis has previously undertaken in an industrial context. However, for contractual reasons the specific details of the project cannot be published, and so this section outlines an Order Management scenario in the context of a fictional business.

8.4.1 The Order Management Scenario

Grunnings is a medium-sized-but-growing firm based in Surrey which manufactures electric drills for the domestic Do-It-Yourself enthusiast. In order to consolidate the sales growth recently experienced by the company, as well as to facilitate a continuation of this trend going forwards, Mr Dursley, the Chief Executive Officer (CEO) at Grunnings has introduced a new "lean manufacturing" policy. As part of this policy, the company is in the process of moving from a policy of mass manufacture to a build-to-order process. In the past the marketing and sales directors would collaborate to set quarterly sales targets, and the manufacturing director would then set the manufacturing targets accordingly. However, this has proven to date to be an inexact approach to target setting with the result that in some quarters Grunnings would lose customers to its competitors as it was unable to satisfy greater-than-estimated demand, whereas in other quarters Grunnings would lose profit to waste having over-estimated demand for the quarter. The new build-to-order business model which CEO Vernon Dursley is introducing at Grunnings will resolve these issues by allowing Grunnings to manufacture, each quarter, precisely the number of drills it will be capable of selling.

To support the new build-to-order approach, CEO Dursley has also commissioned the development of a new software system to manage orders end-to-end. The goal is to ensure prompt and timely fulfilment of orders by facilitating efficient order scheduling, providing live picking information to production-line operatives, and providing up-to-date stative information about the fulfilment of an order at all times. At Grunnings a sales system is already in place which allows the sales teams to record details of drill sales to customers. The new order-management software system should connect to this sales system in order to retrieve sales from which orders can be inferred and added to an in-house database. The system should then apply a simple procedure in order to prioritise orders according to the date on which the order was placed, the deadline for the order, the number of previous orders a customer has placed, and the type of drill the customer has ordered. The output of this procedure should be a build schedule for

the production line to cover a one-day period. The system should provide a means for printing scheduled drills out onto "T-Cards", which represent the schedule for a day. The system should also allow production line operatives to scan drills onto the production line to indicate that production of a particular drill for a particular order has commenced, as well as scanning drills offline to indicate that the build of such a drill has now been completed and that the drill is ready for dispatch. The system should provide adequate reports on the state of the drill and its associated order as it progresses through the manufacturing process, although online updates are not a requirement.

8.4.2 The Order Management OSM

Figure 8.5 provides the auto-generated overview diagram for the Order Management OSM:

This diagram consists of four domains as follow:

- **The OrderRetrievalDomain** captures the transfer of order records from the sales system to the order management system and has the "Conceptual" Domain Type. It is acted on by a "Software" Agent — the order retrieval software which connects to the sales database to download the latest order batch. The Key Object (Order) represents orders within the system. The Initial State of the Domain is a Primary State (OrdersNotRetrievedState) indicating that orders are stored in a Structure (OrderServer) representing the interface to the sales system. The Goal (OrderRetrievalGoal) is represented by a primary state (OrderRetrievedState) which stipulates that no orders are stored in the OrderServer and all orders are now contained in a Structure object (OrderManagementSystem) which represents the order database. This transition is captured by the "OrderRetrievalTransition" State Transition, which triggers the "OrderRetrievalCompleted" event once all Orders have been transferred from OrderServer to OrderManagementSystem.

- **The OrderSchedulingDomain** represents the process of organising orders into a schedule for a particular day and has the "Conceptual Domain Type". It is acted on by a "Software" Agent (OrderScheduler) which represents the software algorithm which is responsible for producing order schedules for a given day. The Key Object is the Order object, which is shared with the OrderRetrievalDomain. The Initial State (OrdersNotScheduledState) is a primary state

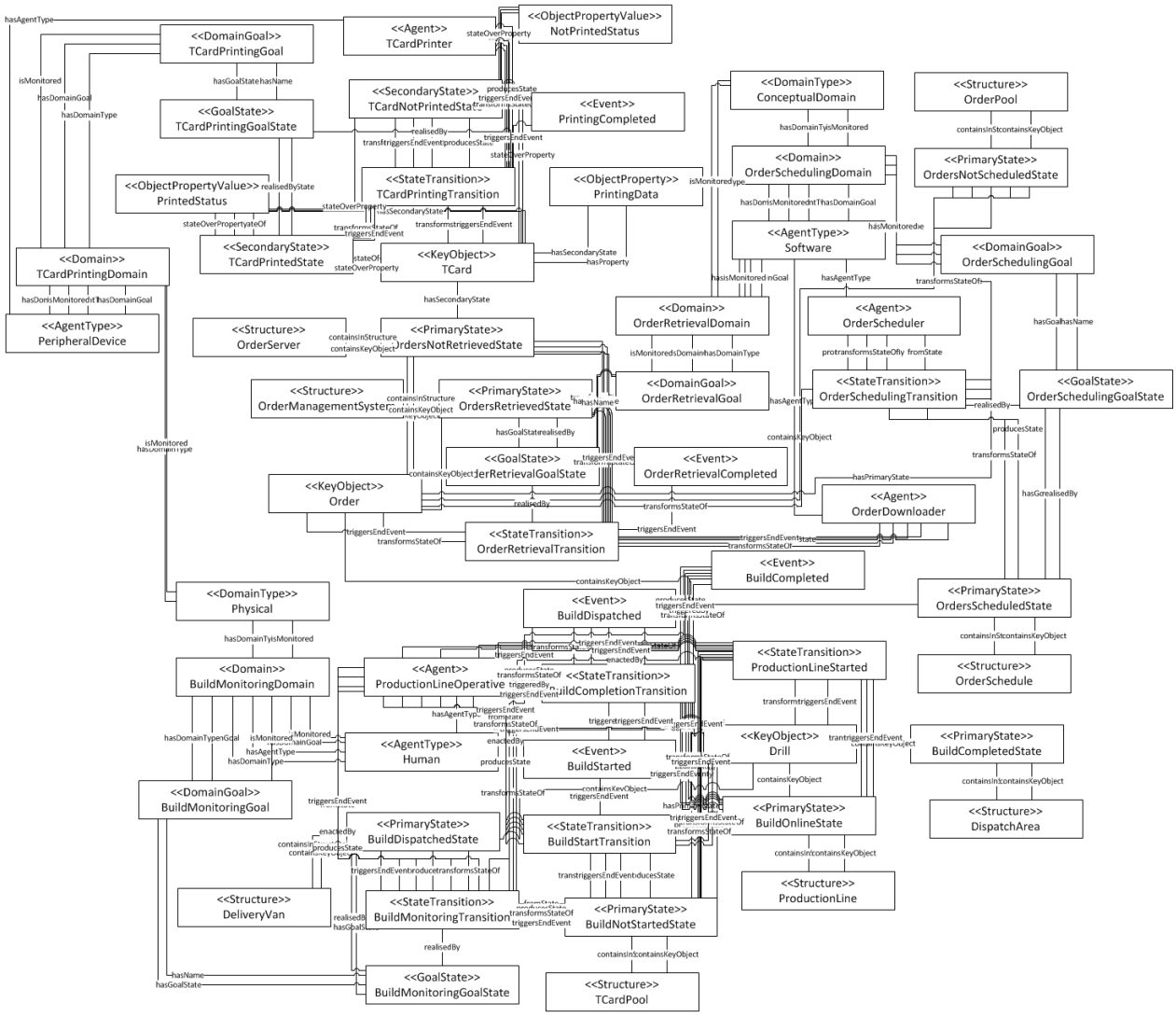

Figure 8.5: The Auto-generated Diagram of the RORE-formalised Order Management OSM

stipulating that Orders exist which are contained within a Structure object (OrderPool), which represents the pool of currently-unscheduled Orders. The Goal State (OrderSchedulingGoalState) stipulates that all orders are contained within a Structure (OrderSchedule) representing an Order Schedule. The scheduling process is represented by the "OrderSchedulingTransition" State Transition.

- **The TCardPrintingDomain** represents the process of printing a build schedule onto a set of T-shaped cards and has the "Physical" Domain Type. It is acted on by an Agent (TCardPrinter), of type "Peripheral Device", which represents the printer that will be used to print orders onto T-Cards. The Key Object (TCard) represents the TCards themselves, and have a Property (PrintingData) through which the status of a T-Card ("Printed" or "Not Printed") is denoted, and which might also contain the Order information that is to be printed onto a particular T-Card. The Initial State of the Domain is a Secondary State (TCardNotPrintedStatus) denotes that all T-Cards in the system have the "Not Printed" status. The Goal State stipulates that all T-Cards in the system have the "Printed" status. The printing process is represented by the "TCardPrintingTransition" State Transition which produces the "PrintingCompleted" Event when the Goal State is achieved.

- **The BuildMonitoringDomain** represents the path of each drill through the production line and to the dispatch area as it is first scanned online, then constructed, and then dispatched. The domain has the Physical Domain Type and has an Agent (ProductionLineOperative) of type "Human" which represents builders on the production line who have responsibility for producing a drill. Drills within this Domain pass through several stages of production: Not Started; Online; Completed; Dispatched. The stage of production which each drill has reached is recorded by a set of Primary States (respectively: BuildNotStartedState, BuildOnlineState, BuildCompletedState, BuildDispatchedState) which describe the containment of the drill in a set of Structure objects representing the area of the factory in which the drill is stored during that production stage (respectively: TCardPool, ProductionLine, DispatchArea, DeliveryVan). The Transition between each of these states is captured by a sequence of State Transitions (respectively: BuildStartedTransition, BuildCompletedTransition, BuildDispatchedTransition) each of which produces an Event on completion (respectively: BuildStarted, BuildCompleted, BuildDispatched). The Goal State states

that each of these events must be captured.

These domains map onto the following OSM contexts.

- The first domain (OrderRetrievalDomain) is a ⟨**Transfer, Conceptual, Software**⟩ domain;
- The second domain (OrderSchedulingDomain) is an ⟨**Allocation, Conceptual, Software**⟩ domain;
- The third domain (TCardPrintingDomain) is a ⟨**Manipulation, Physical, Peripheral Device**⟩ domain;
- Finally, the fourth domain (BuildMonitoringDomain), a ⟨**Transfer, Physical, Human**⟩ domain.

Cycle One: Generating System Classes The "HasISMObjects" analysis rule was fired and failed. Accordingly, an information requirement was generated mandating that cycle one should aim to generate the system classes that will form the basis for the Order Management ISM's object model. In order to satisfy this requirement, the user attempted chunk-based inference. While inference chunks were found, a close inspection of each of these indicated that each was insufficient to fulfil the information requirement satisfactorily. RORE therefore progressed to rule-based inference. As for the previous examples which this thesis has reported, the "GenerateISMObjects" was identified as the most appropriate production script for resolving the current information requirement. RORE fired this production script and a large set of objects was generated, reflecting the complexity of the input OSM. The full list is reported in the log in Appendix E and is shown in the ISM model presented in Figure 8.6.

Cycle Two: Generating Monitor Reports In the second cycle, the "HasISMObjects" rule passed, by the "HasMonitorReports" test failed. The resulting information requirement specified that cycle two of the Order Management example should focus on generating monitor reports for each of the key objects specified in the OSM for domains whose monitor tag is specified as "true". While chunk-based inference was attempted, manual inspection of the retrieved model chunks confirmed that none of the chunks was appropriate at this stage. RORE therefore moved on to attempt rule-based inference. The "GenerateMonitorReports" script was returned at this stage and fired. As a result, four reports were produced to report on the *Order* and *Drill* key objects.

For each of these key objects a ListReport was generated to display a list of all relevant objects within the software system, and a singleton Report was generated to display the details of a specific Order or Drill object respectively.

8.4.2.1 Cycle Three: Assigning Functional Responsibilities

In cycle three both the "HasISMObjects" and "HasMonitorReport" rules passed, while the "HasFunctionalResponsibility" rule failed. The consequence of this outcome was that an information requirement was generated requiring that cycle three focused on the assignment of the functional requirements modelled by the input OSM across the system classes modelled in the output ISM. Chunk-based inference returned a handful of model chunks which specified operations, but did not specify the classes to which these were to be assigned. RORE therefore progressed to rule-based inference. The "FunctionalResponsibilityAssignment" script was retrieved and fired. As in the previous examples, it generated a batch of operations for ISM classes representing both OSM key objects and OSM agent objects. The assignment of functionality determined by this script in the case of the Order Management example was as follows:

- **DrillClass**: transferDrillTo() which updates the location property of a Drill object;
- **OrderClass**: allocateToOrder() which allocates a TCard time slot to a given order; transferOrderTo() which retrieves a particular order from the sales management system to the order management system;
- **TCardClass**: manipulatePropertyOfTCard() which prints a TCard by producing an appropriate instruction for the TCardPrinter;
- **ProductionLineOperativeClass**: transferDrillObjects() maintains the current location of all drills within the order management system;
- **OrderSchedulerClass**: allocateOrderObjects() runs a complete TCard scheduling session by allocating individual TCard slots to orders;
- **OrderDownloaderClass**: transferOrderObjects() retrieves all order objects from the sales system;
- **TCardPrinterClass**: manipulateTCardObjects() prints all TCard objects that are in the current TCard allotment.

Cycle Four: Selecting a Generic Algorithm In cycle four, all three of the previous analysis tests - "HasISMObjects", "HasMonitorResource" and "HasFunctionalResponsibility" passed. The "HasDataSource" analysis rule was therefore fired and failed producing an information requirement mandating the production of data source links between objects and resources in the ISM. Chunk-based inference was attempted at this point, and generic algorithm chunks were identified. The "CompleteRemoteTransfer" chunk was identified, which represents the transfer of a complete data file from a source to a destination across a remote computer network. This chunk introduced a new fact - a resource representing the communication protocol by which data is transferred in this domain. Two operations were also introduced: transferAllCompleteObjects() and transferCompleteObject(), each specifying the generic algorithm pseudocode for their respective methods. These were paired to the transferOrderObject() and transferOrderTo() methods respectively, and all associated facts were integrated into the OrderManagementSpecs model accordingly.

Cycle Five: Selecting a Generic Algorithm In cycle five, the "HasDataSource" analysis rule was, again, failed by the Order Management ISM. Accordingly, the corresponding information requirement was generated. Chunk-based inference was again performed, this time retrieving the "AllocateObjectByClassification" algorithm. This produced a new class - AllocationRule - to implement the logic for an individual allocation rule. The generic algorithm also produced two operations: allocateAllObjects and allocateToOrder. By pairing these with the allocateOrderObjects and allocateToOrder methods in the current ISM model the user ensured that the associated generic algorithm pseudocode would be integrated into the resultant model.

Cycle Six: Selecting a Generic Algorithm Cycle six involved a third round of generic algorithm selection. The third round corresponded to the fact that the Order Management OSM comprises three domains with software or peripheral device agents. Again the "HasISMObjects", "HasMonitorReport" and "HasFunctionalResponsibilities" rules passed, while the "HasDataSources" rule failed. Chunk-based inference was attempted and the "ScriptedDeviceManipulation" chunk was selected. This model chunk aggregates a set of generic algorithms for manipulation of a physical object by a peripheral device which accepts as input from the control software a string representing a script which the device interprets before executing. The chunk introduces

a new resource representing the interface to the communications protocol for the manipulation device, as well as an ISM class which implements a method for generating a script from some input object. The generic algorithm also introduces two methods: a manipulateAllObjects method, which was paired with the manipulateTCardObjects method in the Order Management ISM; and the manipulateObject method, which was paired to the manipulatePropertyOfTCard method. The generic algorithm pseudocode was integrated accordingly.

Cycle Seven: Configuring Functional Resources During the seventh cycle, no further model chunks or production scripts were identified to effectively resolve the "HasDataSource" information requirement. The user therefore attempted Elicitation using the fact-editing "ConfigureFunctionalResources" stimulus, but no significant changes were made to any resource at this stage.

Cycle Eight: Specifying Data Sources and Sinks In the final cycle of the Order Management example, the "HasISMObjects", "HasMonitorReports" and "HasFunctionalResponsibilities" rules passed, while the "HasDataSources" rule failed. Again, no new structures could be retrieved for Inference, and so RORE progressed onto Elicitation. The user undertook elicitation using the fact-editing "ConfigureDataSourcesAndSinks" stimulus. This allowed the user to specify the sources and sinks for each property in the ISM model:

- The RemoteDataTransferProtocol was assigned as the data source for the "currentLocation" property of the Order class;
- The ClassificationRuleSource was assigned as the data source for each property of the AllocationRule class;
- The ManipulationDeviceCommsProtocol was assigned as the data sink for the "printData" property of the ScriptGenerator class, and as the data source for the "printStatusProperty" of the TCardClass.

We paired the various facts that were retrieved and edited using this stimulus to facts in the Order Management ISM as shown in the log (see Appendix E), and then applied Integration to complete the production of the sample ISM specs for the Order Management scenario.

8.4.3 The Order Management ISM

The resulting Information System Model for the Order Management problem is illustrated in Figure 8.6.

8.5 Summary

This chapter has demonstrated the requirements engineering perspective of RORE by showing how the OSM-ISM transformation procedure, which was formalised as a set of RORE reusable knowledge structures in long-term memory in Chapter 7, can be applied in the Model Layer to specific requirements engineering scenarios. Three such requirements engineering scenarios were presented: the Autopilot Example, the File Transfer Example, and the Order Management Example. For each scenario, a textual description was given, and the scenario was also represented as a refined OSM. The prototype Requirements Task Assistant was then used to show, step-by-step, how the reusable knowledge structures that were produced in the previous chapter can be applied in the Model Layer to transform each of these OSM into a corresponding ISM. The transformation procedure as applied to each of the requirements scenarios is described in the logs shown in Appendices C, D and E respectively. This chapter has, therefore, demonstrated how a requirements Activity which is formalised as reusable knowledge structures in long-term memory can be applied to specific requirements engineering scenarios to produce and refine requirements Models.

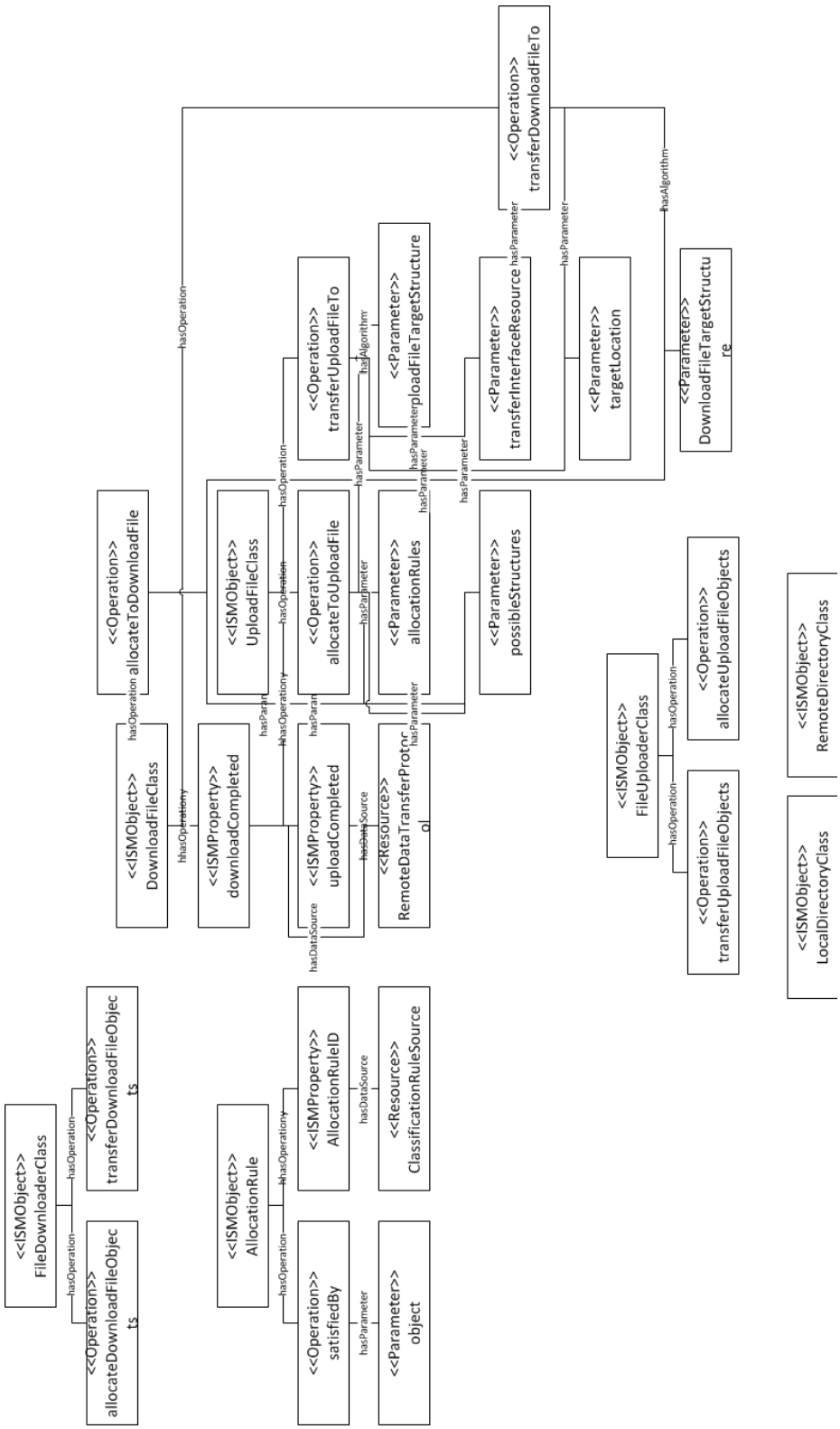

Figure 8.6: The Auto-generated Diagram of the RORE-formalised Order Management ISM

Chapter 9

Research Validation

9.1 Introduction

The introduction to this thesis laid out the following research aim:

> To find a better balance between generality and systematicity than do existing approaches [to requirements reuse] while maintaining a high level of utility.

This chapter evaluates the research presented in this thesis, specifically considering the extent to which the RORE framework satisfies the research aim above. This chapter first defines the evaluation criteria against which RORE will be evaluated. The chapter then applies those criteria to evaluate RORE on its own terms. Finally, this chapter draws on that discussion in order to compare RORE to three of the major approaches to requirements-level reuse: the Domain Theory, Hall et al's Problem-Oriented Software Engineering, and Requirements Patterns as a general approach.

9.2 Evaluation Criteria

The research aim outlined in the introduction of this thesis implicitly identifies four criteria against which the RORE framework is to be evaluated. These are:

- **Generality**: The range of domains, reuse contexts and requirements engineering methods to which a reuse approach can be applied;
- **Utility**: The effort reduction achieved by reusing a reusable artefact versus achieving the same goal manually;

- **Systematicity**: The extent to which reuse is a driving force, rather than an incidental occurrence, in the software development process, and to which such reuse is supported by a repeatable set of procedures and tools;

- **Practicality**: The ability of a reuse approach to satisfy the organisational, economic, legal and technical constraints imposed of a practical setting while still yielding utility.

Before evaluating the RORE framework, this chapter briefly defines each of these criteria, and identify specific qualitative measures which this research will utilise to evaluate the framework in respect of each criterion.

9.2.1 Generality

In evaluating the research presented in this thesis, this chapter adopts and refines the definition of generality given by Sutcliffe [Sut02]: "[g]eneric artefacts have a wider potential target for future reuse by virtue of their abstraction, but pay the penalty of delivering less detailed advice to the designer".

This thesis modifies this definition to take into account the possibility of design tactics for achieving generality, other than abstraction:

> Generality is a measure of the range of domains, reuse contexts and requirements engineering methods to which a reuse approach can be applied.

This definition identifies three types of generality, in accordance with three variables which significant change the overall context to which a reusable approach might be applied:

- **Domain Generality**: The capacity of a component to support problem solving across a range of different application contexts and domains;

- **Task Generality**: The capacity of a component to support a range of different engineering tasks for each application domain to which that component is applicable;

- **Method Generality**: The capacity of a component to be integrated into the notations and techniques comprised by a range of different engineering approaches.

Dimension of Generality	Observable Measure
Domain	Scenario Number and Range
Task	Task Number and Range
Method	Method Number and Range

Table 9.1: Observable Metrics for Evaluating RORE along Dimensions of Generality

This chapter evaluates RORE — in addition to the Domain Theory and POSE — with respect to each of these three types of generality. In order to assess systematically each kind of generality the evaluation will apply the following objective measures (see Table 9.1).

These measures are further defined in the following way:

- **Scenario Range**: A qualitative evaluation of the degree to which two domains that were tested during the case study differ from one another;
- **Scenario Number**: A quantitative evaluation of the number of qualitatively distinct scenarios that were tested during the case study;
- **Task Range**: A qualitative evaluation of degree to which two tasks that were attempted during the case study differ from one another;
- **Task Number**: An evaluation of the number of qualitatively distinct tasks that were tested during the case study;
- **Method Range**: A qualitative evaluation of the degree to which requirements engineering methods (notations, techniques, strategies) that were applied during the case study differ from one another;
- **Method Number**: An evaluation of the number of qualitatively distinct methods that were tested during the case study.

Range measures are important because they provide an indication of the extent to which two different domains that are tested during a case study actually provide an indication of generality. Recall that generality is defined by the range of *distinct* scenarios to which a component can be applied. The quantitative element of the evaluation is significant because it provides an indication of the extent to which the conclusions of a case study can be generalised.

9.2.2 Systematicity

The definition of systematicity given in this thesis draws on existing definitions within the reuse literature [LMV97, Sch99, Kon96]. This thesis defines "systematicity" in the following way:

> Systematicity is the extent to which reuse is a driving force, rather than an incidental occurrence, in the software development process, and to which such reuse is supported by a repeatable set of procedures and tools.

The definition above identifies two key components of systematicity:

- Centrality to the software engineering process;
- Repeatability of the procedures which support reuse operations.

These are illustrated in Table 9.2.

Component of Systematicity	**Observable Measure**
Centrality	Percentage of manual operations versus total operations
Repeatability	Average frequency of the application of a given operation

Table 9.2: Observable Metrics for Evaluating RORE with Respect to Systematicity

The two criteria outlined above can be characterised as follows:

- **Percentage of Manual versus Total Operations**: An indication of the proportion of manual operations to reuse-driven operations from which the engineering process is composed. A high number of reuse operations versus the total number of operations performed in order to refine a model is indicative of the centraility of reuse to a model transformation framework. This is because it suggests that — in line with the design objective of RORE which is stated in section 3.3.3 — operations are driven "predominantly by reuse" rather than by manual reasoning;
- **Average Frequency of the Application of Operations**: A measure of the extent to which reuse operations recur throughout the RORE life-cycle.

9.2.3 Utility

At least three clear definitions of the term "utility" are given in the literature [Sut02, YF00, NF00]. This thesis draws these definitions together into the following definition:

> Utility measures the effort reduction achieved by reusing a reusable artefact versus achieving the same goal without the aid of reuse.

This definition points to two major components of the utility equation:

- The financial and temporal capital which an organisation must invest in a reuse approach in order to establish the program;
- The return which that approach will deliver in terms of the reduction which is delivered by the reuse approach in the total effort required to produce a software system.

At this stage, however, RORE has been tested only as a proof-of-concept prototype and not as a commercial product. This thesis cannot, therefore, fully and accurately evaluate RORE with respect to either criterion. The evaluation therefore focuses on the extent to which RORE exhibits the kinds of features which are known or commonly acknowledged to offer utility, as shown in Table 9.3.

Component of Utility	Observable Measure
Initial Investment	Time to formalise requirements method
Time Savings	Proportion of adaptation time versus total time
Time Savings	Usability of the prototype tool

Table 9.3: Observable Metrics for Evaluating RORE with Respect to Utility

This thesis clarifies each of these metrics as follows:

- **Time to Formalise**: The time spent formalising the Object and Information System Model schemas, and the associated requirements tasks and associated knowledge structures, into a long-term memory knowledge base.
- **Adaptation versus Total Time**: The time spent adapting reusable knowledge structures as a proportion of the total time spent applying the RORE requirements engineering tasks to construct new models. This provides an indication of the extent to which the RORE approach actually reduces manual reasoning in the requirements engineering process.

- **Usability of the Prototype**: A necessary condition of the claim that the RORE approach offers a high degree of utility, usability refers to the ease with which a user is able to utilise a tool to achieve a particular goal. This can be evaluated through user testing or, as in this thesis, through a qualitative self-evaluation of the application of the tool to a case study.

This thesis notes that data for each of these metrics was collated under laboratory test conditions for a small number of examples, and not in real-world conditions. Furthermore, control studies were not undertaken to evaluate these measures in the case of a manual process. These limitations undermine the rigour of the evaluation of utility presented in this thesis. However, the above metrics are sufficient indicators to enable this thesis to draw tentative conclusions about the utility of the RORE approach.

9.2.4 Practicality

Several articles published in the 1990s address the practicalities of reuse in industry [MR92, Gri94, Joo94, Bas96, LL98], and more recently several authors have attempted to identify success and failure factors for reuse projects [Kru02a, MET02, MDS03]. Practicality is a criterion which this thesis considers critical in the design of a novel reuse approach: a reuse approach should be capable of delivering utility in the organisation, not just in the laboratory. This thesis defines practicality as follows:

> Practicality is the ability of a reuse approach to satisfy the organisational, economic, legal and technical constraints imposed of a practical setting while still yielding utility.

Given the prototypical nature of RORE in its current stage of development, the evaluation of RORE with respect to practicality is predicated on the following assumptions about the application of RORE in a real-world scenario:

- Distribution within a single organisation as a standalone application to run on the workstation of each individual developer;
- Adoption by the organisation of requirements engineering methods which depend exclusively on public domain technologies, notations and techniques;
- Population of the knowledge bases with published model chunks and production scripts, as well as those abstracted by the organisation itself;

- A full-time team exists to review documentation produced by development teams within the organisation in order to abstract and formalise reusable knowledge structures from the work of the organisation;
- All intellectual property produced by the organisation is the property of that organisation; there are no contractual obligations to the contrary.

These constraints largely rule out the economic, organisational, social and legal factors which usually constrain reuse in practice and so allow the evaluation to focus on technical constraints instead. There are two potential issues which are likely to impact on practicality as illustrated in Table 9.4:

Component of Practicality	Observable Measure
Technical Feasibility	Performance of the Prototype Implementation
Technical Feasibility	Scalability of the Prototype Implementation

Table 9.4: Observable Metrics for Evaluating RORE with Respect to Practicality

These metrics are defined as follows:

- **Performance** of the RORE prototype tool with respect to spatial and temporal performance over the course of several analysis-action cycles.
- **Scalability**: An evaluation of the capacity of RORE to perform reasonably with grow in the complexity of requirements models and long-term memory knowledge bases.

9.3 Evaluation

9.3.1 With Respect to Generality

9.3.1.1 Scenario Number and Range

Table 9.5 shows a breakdown of the type and number of each domain used for each case study.

Six domain types were tested in total. This equates to two context types for each example which this research tested. In fact, every example consisted of precisely four application domains indicating that the examples were of comparable complexity, and this average is a result of the fact that some context types were reused both within and

Context Type	**Autopilot**	**File Transfer**	**Order Management**
ACS	0	2	1
CPP	1	0	0
MPP	0	0	1
SPP	0	0	1
TCS	0	2	1
TPH	0	0	1

Table 9.5: Breakdown of Context Types by Example

across application domains. The number of context types tested can be calculated as a percentage of the total number of context types which could have been evaluated within the constraints of the demonstration which this thesis adopted:

$$100(\frac{T}{D \times G \times A}) \tag{9.1}$$

where:

- $T = 6$ is the number of context types tested;
- $D = 4$ is the number of domain types defined by the restructured Domain Theory;
- $G = 7$ is the number of goal types defined by the restructured Domain Theory;
- $A = 3$ is the number of agent types defined by the restructured Domain Theory.

To summarise:

- Ten concrete domain instances were tested, spanning a total of six different context types;
- This amounted to an average of 1.7 concrete instances tested per context type;
- The statistic also represented 7.14% of the total range of OSM context types which are expressible using the refined OSM schema.

This thesis considers that this sample has not allowed this evaluation extensively to explore the capabilities and limitations of the RORE approach, in particular because the evaluation has not been sufficiently extensive to discover the domains, model types, and activities which RORE might not be able to support. In order to determine the

potential generality of RORE, therefore, this thesis will need to apply the approach for generalisation from case studies which is recommended by Gomm et al [GHF00] — to generalise by reasoning over the relevant homogeneity and heterogeneity between the observed and unobserved cases. To support this analysis, Table 9.6 shows an analysis of the reusable structures that were used through all three of the case studies against the context types to which they were applied. Context types are coded as follows:

- **ACS**: Allocation, Conceptual, Software;
- **CPP**: Control, Physical, Peripheral Device;
- **MPP**: Manipulation, Physical, Peripheral Device;
- **SPP**: Sensing, Physical, Peripheral Device;
- **TCS**: Transfer, Conceptual, Software;
- **TPH**: Transfer, Physical, Human.

The first conclusion which may be drawn with certainty from Table 9.6 is that the RORE approach can, under certain circumstances, support cross-domain reuse. This is evidenced by the fact that every production script and elicitation stimulus listed in the table was reused across every context type that this research tested in Chapter 8. However, it is clear that this is not true for all reusable structures in the table: not one model chunk was reused between any two distinct domains. Chapter 3 claimed that procedural reuse (as exemplified by Production Scripts) would prove more general than declarative reuse (as exemplified by Model Scripts) and these results appear to confirm that claim, although further data from additional case studies would be required to test the generality of this conclusion.

Two factors can be identified which would have affected the results, and the impact of these must be assessed in order that this thesis avoid drawing a false conclusion of generality:

- The degree of difference between the tested contexts;
- The respective designs of the tested reusable components.

Structure Type	Structure	ACS	CPP	MPP	SPP	TCS	TPH
Chunk	Classification	✓					
	CompleteRemoteTransfer					✓	
	DataSampleSensing				✓		
	NotifiedSampleControl		✓				
	ScriptedDeviceManipulation			✓			
Script	AddMonitorReport	✓	✓	✓	✓	✓	✓
	FunctionalResponsibilityAssignment	✓	✓	✓	✓	✓	✓
	ObjectModelGeneration	✓	✓	✓	✓	✓	✓
Stimulus	DataSourceAndSink	✓	✓	✓	✓	✓	✓
	FunctionalResourceConfiguration	✓	✓	✓	✓	✓	✓

Table 9.6: Summary of Knowledge Structures and their Application to Context Types

Degree of Difference. The results illustrated by Table 9.6 cannot truly be said to be indicative of domain generality unless there exists a substantial conceptual distance and structural distance between the domains that were tested. When this thesis refers to "domain generality" it means the range of different scenarios to which a reusable structure can be applied. The greater the range of scenarios which may be modelled, the greater the domain generality. Accordingly, the important concept here is not the difference between the representations of the scenarios themselves, but the conceptual and structural differences which exist between the domains themselves.

It is immediately clear that there is at least a small degree of variation between the application domains as pointed to by the fact that testing of RORE covered domains spanning all three agent types and two of the four domain types as shown in Table 9.7.

These differences are significant. Each Agent Type gives rise to a substantially different set of generic requirements and concerns. For instance, a Peripheral Device domain generally gives rise to functional requirements which necessitate the software system to control that device, while a Software domain gives rise to a functional requirement that the goal of a domain be enacted predominantly by some part of the software system. A Human domain, by contrast, requires no such functional requirement to be introduced at all. These represent substantially different software systems. However, as Sutcliffe observes [Sut02], the structure of the domain is the primary factor which distinguishes one domain from another. Accordingly, Table 9.8 summarises the structures of each of the tested domain goals, and lists the tested domains for each goal type.

The above table reveals that there are in fact some striking structural similarities amongst some of the goal types described in Table 9.8. However, as the pairwise comparison in Table 9.9 shows there are, in fact, some important distinctions between the goal types. Each cell states how the row's Goal Type is ditinguished from the column's Goal Type:

There are, then, clear structural distinctions between each of the domain goals which, taken together with the conceptual distinctions that arise from differences between contexts in agent and domain type, suggest that each of the domains which this research tested are indeed significantly different. There is a final piece of evidence to support this conclusion, however. Chapter 7 stated that each Goal Type is associated with an underlying deeper semantics which cannot directly be represented by Sutcliffe and Maiden's meta-schema, or the restructured form of it. Each Goal Type is associated with a key concept which defines the central idea which underpins the behaviour

		Agent Type		
		Human	Software	Peripheral Device
Domain Type	Physical	BuildMonitoringDomain	-	TCardPrintingDomain, XAngleController-Domain, YAngle-ControllerDomain, XAngleSensorDomain, YAngleSensorDomain
	Conceptual	-	OrderRetrievalDomain, OrderSchedulingDo-main, UploadAlloca-tionDomain, Down-loadAllocationDomain, UploadTransferDo-main, DownloadTrans-ferDomain	-
	Physical/Financial	-	-	-
	Electronic/Finacial	-	-	-

Table 9.7: Classification of Domains Tested by Agent and Domain Type

Goal Type	Structure	Sample Domains
Allocation	A key object resides in a second structure. An agent object creates an association between the key object and a second structure object	OrderSchedulingDomain, DownloadFileAllocation, UploadFileAllocation
Control	An agent dispatches events which are detected by a key object. The key object changes its state accordingly.	XAngleControllerDomain, YAngleControllerDomain
Manipulation	An agent interacts with a key object directly to change its secondary state. An event is triggered when the state change is complete.	TCardPrintingDomain
Sensing	A key object changes its state. The secondary state change produces an event which is detected by an agent object.	XAngleSensingDomain, YAngleSensingDomain
Transfer	An agent object interacts with a key object causing it to change its primary state. An event is produced denoting completion of the activity.	OrderRetrievalDomain, BuildMonitoringDomain, UploadFileTransferDomain, DownloadFileTransferDomain

Table 9.8: Structural Summary of Tested Domains with Examples

	Allocation	**Control**	**Manipulation**	**Sensing**	**Transfer**
Allocation	-	Allocation retains state; control changes primary or secondary state.	Allocation retains state; manipulation changes secondary state.	Allocation creates mapping; sensing detects state changes.	Allocation retains state; Transfer changes primary state.
Control	-	-	Controlled key object responds to command and transforms own state. Manipulation: Agent object performs transformation.	Control causes state change; Sensing detects change.	Controlled key object commanded to change own state; Transfer: Agent enacts primary state change. Control domain changes primary or secondary state.
Manipulation	-	-	-	Manipulation: Agent causes state change. Sensing: Agent detects state change.	Manipulation domains change secondary state; Transfer domains change primary state.
Sensing	-	-	-	-	Sensing: state change is spontaneous, Agent is notified. Transfer: Agent causes state change.

Table 9.9: Distinctions Between Goal Types

of that goal type. When this is examined, this thesis observes that the semantics of each goal type is in fact substantially different in each case:

Goal Type	Key Concept
Allocation	Location
Control	State Transitions
Manipulation	Secondary State
Sensing	A State
Transfer	Location

Table 9.10: Structural Summary of Tested Domains with Examples

These semantic concepts are human interpretations - rather than formalised interpretations - of the meaning of a Domain Goal. However, as shown in Table 9.8, these semantic interpretations differ significantly between most of the goal types. While there is some overlap between certain goals, however, Goal Types which share a common key concept are distinguished by significant structural. This thesis concludes, therefore, that there is a sufficient degree of distinction between the contexts which this research tested in the demonstration to support the conclusion that the particular set of knowledge structures which this thesis identified do indeed support domain generality, and that the reuse of procedural and elicitation knowledge supports a greater degree of domain generality than the more conventional declarative reuse.

Comparison of Designs. A second factor could also explain, and potentially falsify, the results which Table 9.6 presented is the comparative design of each knowledge structure that was used during the case study. More precisely, generality is related to both granularity and abstraction [Sut02], and so if model chunks were designed at a lower level of abstraction and coarser level of granularity than either the production scripts or elicitation stimuli that were used, then this variation would be a probable explanation for the observed variation in generality between model chunks and production scripts.

There exists an inherent methodological difficulty in comparing the granularity and abstraction of production scripts and model chunks. The granularity of production scripts can be estimated in terms of logical distance which measures (crudely) the number of logical inferences that need to be drawn to produce an output from a given input. One might also consider the granularity of the output from a production script rather than the granularity of the script itself. A reasonable heuristic, however, might be that a fair test of the relative generality of a production script and a model chunk

might pitch:

- A coarse grained model chunk against a logically distant production script;
- *OR* a fine-grained model chunk against a logically "close" production script.

A more concrete heuristic constrains the two types of knowledge structure as follows:

- Model chunks should comprise no more than two or three nodes, as per Sutcliffe's prescription [Sut02];
- Production scripts should be geared towards producing facts of no more than two or three fact types.

One could apply either measure to constrain an elicitation stimulus:

- *Either* the model chunk which a user provides as a response to a stimulus should comprise no more than two or three nodes;
- *OR* the elicitation stimulus should be geared towards eliciting facts of no more than two or three fact types.

We adopt the second of these two measures as it is applicable to all three elicitation stimulus types.

A brief survey of the reusable knowledge structures which this research applied throughout the demonstration in Chapters 7 and 8 reveals that these constraints were indeed satisfied. For instance, the transformation procedure which is described in section 7.4 identifies elicitation stimuli to request from the requirements engineer facts of three different types.

- Object properties;
- Resources;
- Data sources and sinks.

Separate elicitation stimuli are defined for each fact type, and so these stimuli satisfy the criterion that a stimulis should elicit "facts of no more than two or three fact types". The granularity and logical distance of all components was, therefore, largely comparable suggesting a fair test.

There is, however, a second factor which impacts on the generality of individual components: abstraction. The more abstract a component, the more general it will be. Conversely, the less abstract a component is, the less general [Sut02]. Comparing the abstraction of production scripts and model chunks, however, presents a significant challenge because the concept of abstraction has different emphases when applied to processes as opposed to models. In either case, abstraction refers to information hiding, but when applied to models the term abstraction refers to the amount of information that is hidden about the nature of the concepts and activities which that model describes. By contrast, what is hidden when a process is abstracted is the detailed sequence of steps by which the goal of that process is realised: the "implementation". This is of little value in comparing the abstraction of a production script to that of a model chunk. What is more significant is the specificity of the facts over which a script operates and produces.

The model chunks which Chapter 7 defined may be considered to reside at about the same level of abstraction as the generic domain abstractions described by Sutcliffe and Maiden [SM98]. The facts which comprise the model chunk in each case give an indication of the general type of concept to which the fact refers, and of the role of the fact within the scenario described by the chunk. However, neither set of chunks identifies the specific attributes of the concepts to which they refer, leaving flexibility in the range of interpretations which the model may be given.

The production scripts which were defined generally do not refer explicitly to specific facts. Instead, they use labels and references to refer to facts which are retrieved through queries and provided as input when the production script is initially fired. This makes a general evaluation of the level of abstraction of the production scripts which were described in Chapter 7 difficult since this property is dependent on the specificity of the input facts provided and of those retrieved through any queries. However, this thesis does conclude that in the context of the specific examples which were tested (the Autopilot, File Transfer and Order Management examples), each fact over which these production scripts operated described a specific concept in a specific domain. Furthermore, the facts which each production script produced were context-specific and needed no further adaptation to reify them. Were it true that each production script referred explicitly to specific facts within an input and output model, this would suggest that these production scripts are, in fact, *less* abstract than the model chunks. This is not the case, however, because like model chunks the production scripts use placeholders (variable names in the case of production scripts, abstract names in the case

of model chunks) to refer to facts in specific domains. This thesis concludes, therefore, that the production scripts and the model chunks which this research produced represent an equatable level of abstraction.

9.3.1.2 Task and Method Number and Range

Task Number and Range. The case study which Chapters 7 and 8 present is a small scale study which ranges over just one task: that of transforming Object System Models into Information System Models. The central purpose of this task is the transformation of a source model of one type (an Object System Model) into a target model of a second type (an Information System Model). However, there is good reason to believe that the approach can support at least one other requirements engineering task: the from-scratch production of an initial Object System Model. Six of the basic components that are necessary to support a basic, non-analogical implementation of such a task were fully tested through the existing case study:

- The retrieval of model chunks and elicitation stimuli;
- The adaptation of model chunks;
- The use of chunk-based stimuli to produce new model facts;
- The use of fact editing stimuli to adapt existing model facts manually;
- The integration of knowledge produced into a target model.

A basic implementation of a process to construct an OSM from scratch based on these components might comprise the following steps (following the general process described by Sutcliffe and Maiden [SM98]):

- **Chunk-Based Stimulus**: Define domains, but do not provide details at this point;
- **Chunk-Based Stimulus**: Define key, structure and agent objects;
- **Chunk-Based Stimulus**: Define primary states, state transitions and goal state;
- **Chunk-Based Inference**: Match goal type to each goal state to add additional information (events, secondary states);
- **Fact-Editing Stimulus**: Associate goal states with domains.

We can be confident that this implementation of the "GenerateNewOSM" task would be effective because it uses features of RORE which were thoroughly tested during the case study presented in Chapter 7, and is based on fact and model types, as well as knowledge structures, which are already defined within the long-term memory for the case study. However, it is desirable that RORE be able to support a wider range of tasks than the GenerateOSM and GenerateISMSpecification tasks this thesis has already discussed.

Gomm et al have argued, within the context of social research (although they consider their discussion to be more generally applicable), that it is possible to utilise case studies in generalising from observed to observed cases [GHF00]. Wieringa [Wie12] concurs, and suggests that one method for achieving this generalisation is to use architectural properties of a case study to determine relevancy of an observed case to an unobserved case. Gomm et al consider more broadly how researchers might avoid error in generalising from studied to unstudied cases, emphasising the importance of taking into account the "relevant heterogeneity" within the population under discussion:

> Researchers can [select] cases for study [which are] typical in relevant respects. Whether this is possible, even in principle, will depend on the level of relevant heterogeneity in the population, and on the availability of information about this...[Alternatively, one can] study a small sample of cases that have been selected to cover the extremes of expected relevant heterogeneity within the population. It is worth noting that here cases do not all have to be studied in the same depth: one or two may be investigated in detail, with others examined more superficially to check the likely generalisability of findings from the main case study [GHF00].

Drawing on this discussion, one can consider whether or not the properties are typical or atypical that made the formalisation of the OSM-ISM transformation procedure in Chapter 7 — and its application in Chapter 8 to the Autopilot, File Transfer and Order Management examples — possible using the RORE approach. The key properties that are needed in order to enable a requirements engineering activity to be formalised using RORE are:

- That the activity produces, or transforms, models which are represented in a notation that comprises at base an acyclic graph, comprising at least one type of node and at least one type of arc. Requirements notations which have previously been formalised using the Telos formalism (see [MBJK90]);

- That the activity can be formalised by a sequence of additions or deletions of, or label changes to, the facts within such a model.

Given this analysis, one might also consider other concrete, reuse-driven requirements tasks which share these properties, and which it might therefore be possible to implement using the RORE approach. One such task is the reuse-driven approach to constructing goal hierarchies described by Massonet and Van Lamsweerde [MVL97]. KAOS is a goal-oriented approach to requirements engineering which seeks to decompose high-level organisational goals into low-level, operationalisable system goals. To construct a goal hierarchy, high-level goals are specified and then each goal is reified by identifying what must be done in order to realise that goal. This is an iterative process which stops at the point that all goals have been decomposed to the operationalisable level. Goals are represented as nodes in a goal-hierarchy with arcs between goals representing various kinds of dependency [DFvL91, DVLF93].

Implementing this process in RORE would be straightforward in comparison to the implementation of the GenerateNewOSM and GenerateISMSpecification tasks: the KAOS modeling language comprises a comparatively small number of fact types - predominantly goals and their relationships - which can readily be modelled. Indeed, the refined OSM meta-schema comprised a notion of "goal" which is significantly more complex than that which is defined in KAOS: whereas KAOS represents goals as individual labelled nodes in an acyclic graph, the OSM meta-schema represents goals as aggregate concepts consisting of state transitions over key objects. Thus, this thesis concludes that there is no significant construct required to represent a GORE goal hierarchy which cannot be handled by some demonstrably efficacious component of RORE. Furthermore, being iterative in nature, the process by which a GORE goal hierarchy is constructed could be implemented using just three actions:

- **Chunk-based Stimulus**: Specify initial high-level goals;
- **Chunk-based Inference**: Select, adapt and apply requirements pattern;
- **Fact-Editing Stimulus**: Reify high-level goal manually by adding a new lower-level goal.

The goal-hierarchy example does raise one potential problem. RORE determines whether or not a model is complete by evaluating the state of a source and target model. Because these checks are expressed as SPARQL-DL ASK queries, certain kinds of

check are possible depending on whether the information required to perform the check is contained within either the source, or the target, or an imported model:

- The existence of facts of a specific type;
- The existence of specific individuals;
- The existence of facts with specific property values;
- The existence of relationships between specific facts, or facts satisfying certain relational criteria.

However, the primary condition which needs to be met by a goal hierarchy in order to be considered "sufficient" or "complete" is the "operationalisability" of all goals. It is not immediately clear how this fuzzy concept might be stated explicitly and precisely in terms of checks of these kinds. This presents a significant challenge to the ability of RORE fully to implement the KAOS procedure. Nonetheless, this would limit only the ability of the RORE approach to determine that a goal hierarchy was complete, and would not significantly impact on the ability of RORE incrementally to construct such a hierarchy. As such, the thought experiment has been a useful one in that it has helped to clarify the properties which the task shares in common with those which this thesis has already established could satisfactorily be implemented as RORE tasks. This thesis has shown, therefore, that RORE is sufficiently general to support the formalisation of requirements tasks which satisfy the following criteria:

- The task must either produce or transform a requirements model of one kind, possibly into a model of another kind;
- The model types over which the task operates must be expressible in terms of a graph comprising nodes and arcs between those nodes;
- The task must be expressible in terms of a sequence of steps such that each step consists either of the reuse of a model chunk, or of a production script, or of an elicitation stimulus;
- The task has a set of well-defined stopping criteria which can be expressed in terms of the existence of facts of a particular type, and the attributes of and relationships between those fact.

It is clear that a number of other requirements tasks do indeed satisfy these criteria: the Jacksonian decomposition of problem models into problem domains [Jac01b] and the matching of such domains to problem frames, and the identification and correct of certain kinds of inconsistencies within a model, are two such examples. This thesis concludes, therefore, that RORE does indeed exhibit a high degree of Task Generality.

Method Number and Range. Although this research only utilised a single task of a single method in the case study, this thesis has implicitly addressed in the argument with respect to Task Generality the question of method generality. RORE defines a method as being a set of model types and tasks which collectively serve incrementally to improve the quality of models, and to transform those models into new models of a different type. Thus far, this thesis has demonstrated or argued the following:

- That RORE supports the expression of at least two model types (Object and Information System Models);
- That RORE is capable of expressing another significant requirements model type - KAOS [MVL97];
- That RORE can support a range of requirements tasks which satisfy certain conditions.

Given that the specification of a method requires only that RORE can represent a range of different model types, as well as tasks over those model types, the above conclusions already justify the claim that RORE is capable of supporting a range of requirements engineering methods, and so exhibits Method Generality.

9.3.1.3 Expressivity of the RORE Knowledge Model

In general this thesis found little difficulty in applying the knowledge structures which Chapter 6 outlined in order to formalise the OSM-ISM transformation requirements task. Two distinct requirements models - the Object System Model and the Information System Model - were successfully formalised using the RORE metamodeling framework. Furthermore, RORE was broadly adequate to formalise the transformation task which Chapter 7 outlined. There were, however, limits to this expressivity. In its original form (that of the Chunk-based Stimulus), the Elicitation Stimulus provided only the means for users to respond by specifying an entirely new set of facts. However, this research found in implementing the OSM-ISM transformation procedure that

rather than requesting users to specify new facts, it was actually necessary to request users to reify existing sets of facts. The existing structure of the Elicitation Stimulus was not well-equipped to support this objective. This is because the Elicitation Stimulus was abstracted from the fact acquisition dialogues which underpin the Domain Theory's AIR tool. However, the Domain Theory tool is geared towards the earliest phases of requirements engineering, rather than supporting the latter phases. The earliest stages of requirements engineering should emphasise the acquisition of new information, whereas a dialectical shift can be expected to occur towards information transformation and away from information acquisition as requirements engineering progressed. This hypothesis is confirmed by the fact that the OSM-ISM transformation process depended almost exclusively on production scripts and elicitation stimuli which transformed existing facts, rather than those stimuli which introduced novel facts.

The RORE framework also inherits from its underlying technologies a certain lack of expressive power. In particular the SPARQL-DL query syntax which this research adopted in order to represent model queries and conditions is limited in at one important ways. While these limitations naturally affect any and all query- and condition-based constructs within the RORE framework, this research found through the case study that it most significantly impacted in practice on the specification and representation of analysis rules and rule-based matching conditions. The query syntax does not support queries over multiple, non-integrated, ontologies. This thesis can illustrate the significance of this by reference to the limited implementation of the "HasMonitorReports" analysis rule. A correct implementation of this rule would check that a report exists in the ISM for each key object in the OSM which belongs to a domain whose monitor tag is true. This, however, patently involves a check over both the source and the target model. While this is ostensibly a limitation of the SPARQL-DL query syntax, it is also a product of limitations in the design assumptions which underpin RORE. This research had assumed in the design of RORE that conditions would need to be tested against either the source, or the target model, but never against both concurrently. This is a problem with multiple possible solutions: either the ability to fire conditions over both models concurrently should be introduced; or source information which needs to be referenced when querying the target model should be transferred to the target model as traceability information in which case the problem is a limitation of the implementation of the OSM-ISM transformation procedure. However, this thesis cannot comment further on precisely which solution would be most appropriate as

further case studies would be required to clarify the question.

In order to address these limitations this research introduced a number of refinements to the RORE specification, as outlined in Section 9.3.5. The result of these refinements was that the limitations which prevented key production scripts and elicitation stimuli from being implemented fully were resolved, and so a full and effective implementation of the OSM-ISM transformation procedure was possible. However, because this research did not attempt to restructure the SPARQL-DL query syntax, the example refinements did not impact on this particular limitation which, therefore, remains an issue. While this research did not find reason to believe that this would significantly limit the generality of the RORE approach, as the knowledge base grows it may well significantly limit the utility of the approach since it restricts the power to design effective filters over the reusable structures that are displayed to a user at a particular point in time.

9.3.2 With Respect to Systematicity

9.3.2.1 Results of the Case Study

The assessment of the systematicity of RORE is based on a quantitative evaluation of the levels of reuse achieved during the case studies. In order to gather the quantitative data needed to perform this quantitative evaluation, this research first performed a qualitative evaluation which aimed to classify and count the operations that were performed during the case study. This evaluation focused in particular on the productive activities of RORE, as the analytical and integration operations are fully automated by the RORE task assistant prototype. This thesis considered, therefore, that including these would bias the results to show a more favourable outcome.

We classified operations as either “Manual” or “Reuse-Driven” as indicated in Table 9.11:

Operation Class	Operation Type
Manual	Chunk Adaptation, Elicitation
Reuse	Chunk-Based Inference, Rule-Based Inference

Table 9.11: Classification of RORE Operations for Evaluation of Systematicity

Metric	Autopilot	File Transfer	Order Management	Summary
Total Ops	7	7	8	22
Total Manual	2	2	2	6
Total Reuse	5	5	6	16
% Manual	28.57	28.57	25	27.27
% Reuse	71.43	71.43	75	72.73
Manual Freq PO	0.29	0.29	0.25	0.28
Reuse Freq PO	0.71	0.71	0.75	0.72

Table 9.12: Statistics for the Evaluation of Systematicity of the RORE Process with Respect to Reuse

This thesis then reviewed all logs produced by the RORE requirements task assistant during the thing, and highlighted each instance of the operation types identified in Table 9.11 (a simple task since this is highlighted in the logs already). This research then produced one spreadsheet for each example showing the number of manual operations, the number of reuse operations, and the total number of operations for that example. This research then calculated statistical values for each example, before aggregating these values to produce a summary for the case study as a whole. This summary is presented in Table 9.12.

We also counted the number of occurrences of each individual requirements tasks across all three case studies with the results shown in Table 9.13.

We finally summarised these results, as shown in Table 9.14 in terms of the higher-level categorisations.

9.3.2.2 Discussion

Section 9.2 defined two metrics against which the systematicity of the RORE approach could be measured:

- Centrality, estimated by the ratio of reuse to manual operations;
- Repeatability, estimated by the recurrence of requirements tasks.

The statistics presented in Section 9.3.2.1 clearly indicate that RORE satisfies both criteria. The findings of this evaluation indicate that the RORE approach interweaves manual and reuse-driven operations via a set of highly repeatable procedures which depend on reuse to operate. The main findings as follows can be summarised as follows:

Task	**Autopilot**	**File Transfer**	**Order Management**	**Totals**	**Frequency**
Completeness Analysis	7	7	8	22	0.33
Quality Analysis	7	7	8	22	0.33
Chunk-Based Inference	2	2	3	7	0.11
Rule-Based Inference	3	3	3	9	0.14
Elicitation	2	2	2	6	0.09
Totals	21	21	24	66	
Mean Frequency:	0.2				
Standard Deviation:	0.11				

Table 9.13: Statistics for the Evaluation of Systematicity of the RORE Process with Respect to Requirements Tasks

Task	Frequency
Analysis	0.67
Reuse	0.24
Manual	0.09
Total	1

Table 9.14: Statistics for the Evaluation of Systematicity of the RORE Process with Respect to Reuse

- The average ratio of reuse-manual productive operations (those excluding analysis) was 72%:28% (approximately 3:1);
- When analysis was additionally taken into account, the ratio across all three categories of operation was 67%:24%:9% (analysis:reuse:manaul — approximately 6:2:1);
- On average, a given requirements task recurred once for every five operations;
- Predictably, analysis occurred more frequently (0.33 times for each time of analysis, or 0.67 times for all kinds of analysis collectively);
- Elicitation occurred slightly fewer times than other reuse operations (0.09) times;
- The spread of frequency across all productive operations was much more even than this thesis had anticipated (0.11 for chunk-based inference, 0.14 for rule-based inference, 0.09 for elicitation).

Within the requirements engineering literature, the literature survey presented in this thesis (see Chapter 2) identified no research which has previously attempted to quantify levels of requirements reuse. However, a recent study by Lucredio, de Almeida and Fortes [LAF12] across three application domains found that model-driven engineering (MDE) applied to the task of code generation can deliver between 80-90% reuse, of which 40-50% will be generated by model generators. The authors note that their own work is consistent with the findings of other authors [LAF12] investigating similar questions. The figures presented in this chapter indicate that RORE yields a lower (by approximately 10%) degree of reuse than observed by Lucredio et al. However, this is partially explicable by a difference in metrics (we use the frequency of reuse operations, whereas Lucredio et al depend on Lines of Code metrics which are not relevant to the RORE approach). It is also explicable by the design goals which underpin the RORE approach: Lucredio et al investigated model-driven

approaches which depended on domain-specific languages to support transformation [LAF12], whereas RORE has intentionally sought to move away from the domain-specific paradigm. In order to achieve this generality, however, RORE has traded a degree of automation by introducing systematic support for user intervention in the generative process. This naturally reduces the degree of reuse, but does so by design.

By means of comparison, Selby has investigated levels of reuse in library-based development projects, and observed a 30% rate of reuse. Clearly the 72% level of reuse yielded by RORE on the case studies this research has tested represents an improvement over library based approaches to reuse. This thesis concludes, therefore, that while the generality of RORE slightly undermines the level of reuse achieved when compared to other model-driven approaches, RORE is still able to achieve a significant improvement over more traditional approaches to reuse. Quantitative studies of reuse have, to date, been confined to analysing code-level reuse and so, by necessity, the results against which this research has compared the RORE approach focus on code (rather than requirements) -level reuse. However, this thesis anticipates that were such statistics available specifically for the requirements engineering literature that they would be largely consistent with those produced for code-level reuse. This research anticipates such a similar level of reuse can be achieved by RORE in practice, as this thesis sees few fundamental distinctions between requirements engineering and software programming — both being essentially model-driven activities which involve the creation of new artefacts based on logical reasoning over existing artefacts. There is one significant exception: that requirements engineering depends more heavily on interaction with stakeholders than does programming, and this necessarily reduces the degree of automated reuse that is possible; but this thesis anticipates this factor impacting to a similar degree across all reuse technologies, and thus not significantly impacting on the comparative merits of different technologies.

9.3.3 With Respect to Utility

9.3.3.1 Time to Formalise Method

This thesis interprets the phrase “formalisation of a method” as referring specifically to the activity of modeling the different components of a method (fact types, model types, phase, activities and reusable structures) using the RORE knowledge types. This definition includes both the initial formulation of the formal specification, and subsequent debugging on a number of case studies. This research did not formally measure the

period of time which it took me to formalise the OSM-ISM transformation method using the RORE approach. Instead, the timings are based on clock readings that were taken informally while working on this aspect of the case study. However, these timings should be sufficient to give an indication of the potential ratio of investment to return in the context of this case study. Certainly the two activities were completed in a single evening. The timings are as follows:

Activity	Estimate of Time Taken
Initial Modeling	4.5-6 hours
Subsequent Debugging	2-3 hours

Table 9.15: Estimate of Timings for Formalisation Activities

It can clearly be seen, even from these rough timings, that the effort required to formalise a modeling approach using the RORE requirements task assistant is not substantial so long as the modeling language is already well-specified. The validity of these timings and any conclusions one might draw from them is naturally limited by a number of differences between the laboratory conditions under which this research tested the RORE approach, and the realities of commercial requirements engineering. In particular, this thesis formalised a single task spanning just two model types. Furthermore, the thesis required the formalisation of a small number of abstractions. This is partly a result of the generality of the components involved, and consequently the simplicity of the knowledge bases could be replicated in practice, this thesis suggests. However, it is also partly due to the comparatively small scale of the case studies versus a real-world software engineering case study.

In order to determine that RORE can offer similar advantages in real-world practice as have been observed under the laboratory conditions of this evaluation, a case study could be conducted by taking the following steps:

1. Work with a team of requirements engineers in practice in order to formalise, using RORE, the requirements modeling notations that are used within that team;

2. Perform domain analysis over historical artefacts which have been produced by that team in order to identify abstractions and transformations which can be formalised into a RORE long-term memory knowledgebase;

3. Request that the team then use the RORE tool in a small number of future requirements engineering projects.

The team and experimenter should keep detailed timesheets about the time spent on each activity within the case study. This would allow the timings for knowledge engineering activities to be carefully assessed. The case study could also be more readily applied within teams who already specialise within particular application domains, or adopt a product-line approach, because abstractions for formalisation would more likely be readily available within such contexts.

However, that the timings which are presented in this evaluation have not been acquired through a rigorous commercial case study does not undermine their value in providing an indication of the utility of the RORE approach. The critical factor in evaluating the extent to which the time taken to formalise a model impacts on the utility of the RORE approach is the ratio of formalisation time to reductions in time taken to engineering requirements on each project where the approach is applied. If the application of the RORE approach were to result in an overall increase in the time taken to engineer models due to the time initially taken to formalise those modeling languages, then the RORE approach would be a lame duck solution. If, however, a significant reduction in development time is realised over several projects then the initial time taken is mitigated by the running effort reduction.

While this thesis cannot present accurate comparisons of the time taken to transform OSMs into ISMs manually, it is possible to reason from personal experience that the task would take approximately two hours for an experienced requirements engineer to produce a high quality model which accurately reflected the requirements of the application domain. By contrast, as the timings discussed by Section 9.3.3.2 show, the RORE requirements task assistant is capable of producing a high quality model in significantly less than two hours. The statistics which were discussed in relation to systematicity also point to a reduction in effort achieved by applying the RORE approach: 70% reuse levels, versus 30% reuse levels with more traditional reuse approaches indicates that the reduction in effort realised by RORE will be significant if, indeed, reuse does reduce effort as is commonly assumed [Kru92].

If my estimates of the time taken manually to transform an OSM into an ISM are accurate, then the effort reduction achieved by applying the RORE approach to the OSM-ISM transformation task would not pay for, or would just break even with, the capital investment in formalising the transformation procedure described in Chapter 7. However, adopting a reuse approach offers increasing returns: the return is greater the more projects it is applied to, because effort reduction is cumulative over several

projects. This is particularly true for RORE, versus domain-specific approaches, because the RORE approach can be applied to a wider range of projects than can an individual domain-specific library. Since most real-world organisations would be unlikely to invest in a technology which would be used on just three projects, as has been the case in this thesis, one can expect real-world organisations who might invest in a hypothetical future commercial RORE tool to reap significantly greater rewards, therefore.

9.3.3.2 Adaptation Time versus Total Time

This thesis also measured the relationship between adaptation time and the total time taken to generate an ISM for each case study. This is an important metric by which to judge utility given the strong relationship between utility and effort reduction [Sut02]. Effort is reduced by reuse and by automated procedures, and increased where work must be done manually. For each case study, therefore, this research carefully measured the time taken to generate a model from start to finish and the time spent on the manual adaptation of results. These results are presented in Table 9.16:

Example	Adaptation Time (mins.secs)	Total Time (mins.secs)
Autopilot	1.13	6.49
File Transfer	2.29	7.43
Order Management	2.34	8.57
Total	6.32	22.49
Adaptation / Total Time:	43.90%	

Table 9.16: Estimate of Timings for Formalisation Activities

The results show that more than 40% (43.90%) of the time taken to generate an ISM from an OSM was spent on adapting reusable components. Two comments must be made about these results before analysing them further. Firstly, these results are slightly skewed by the performance of the RORE tool, which Section 9.3.4.1 discusses. Much of the time spent on adaptation was not spent actually doing work, but instead was spent waiting for the RORE tool to complete automated processes. This performance issue arose from the noted performance limitations of OWL-related technologies, and disproportionately impacted on manual adaptation. The result of this is that the actual effort required by a human actor to apply the RORE tool would be slightly lower than the above timings indicate. Nonetheless, the results are sufficient to draw some conclusions about the utility of the RORE tool. Indeed, an overestimate of the

effort required to perform the manual aspects of the RORE procedure can only lead to an underestimate of the utility of the RORE tool. If the RORE tool actually offers greater-than-expected utility then any conclusion about the feasibility of applying the tool in practice will not be affected.

A second observation arises in comparing the relationship between the manual and total work performed during a RORE session as measured by number of operations (see Section 9.3.2.1) and as measured by time. This thesis observes that whereas manual work accounts for slightly less than 10% (9%) of the total number of operations performed during a RORE session, it accounts for slightly more than 40% (43.90%) of the time taken to generate a model. This indicates that the time spent on each non-manual (reuse-driven and automatic) operation in RORE is significantly less on average than the time spent on manual operations, as shown in Table 9.17.

Operation Type	**Time Taken (secs)**	**Number of Operations**	**Average Time / Op (secs)**
Non-Manual	893	60	14.88
Manual	449	6	74.83

Table 9.17: Timings Per Type of Operation in RORE

The observation that automated and reuse-driven operations are faster and more efficient than human-implemented operations is to be expected: these are the fundamental reasons that motivate the application of software to any domain in the first place. However, it is interesting to observe that the average time for each non-manual operation (including all kinds of analysis, inference and integration assuming no adaptation) is significantly higher than one might expect or desire for a computerised operation. This too is an effect which is caused by a performance issue associated with the OWL technologies on which the prototype tool was constructed (see Section 9.3.4.1) and which specifically impacts on the execution of production scripts. Although this research did not acquire timings to support this observation, this thesis concludes that the time to apply a production script was significantly higher than the time to apply an analysis rule or a model chunk without adaptation. The reason for this is clear: the execution of production scripts requires on average many more calls to the OWL API and Pellet reasoner [SPG+07] than does the execution of a single analysis rule (one call to retrieve the rule, one to fire the rule) or the application of a model chunk. Since the OWL architecture on which the RORE prototype was built was found to have important performance limitations, it is logical that production scripts would take longer to execute than other reuse operations. This naturally raises the overall average time

per operation, although reuse operations still took a fifth of the time taken by manual operations on average.

These results present a challenge for the current version of the RORE prototype tool - waiting for 14 seconds for a single operation to complete is practically problematic - the conclusion is less clear cut when one considers the bigger picture. Clearly future versions of RORE, and in particularly any commercial product which may be developed as a product of this research, would need to address the performance issue. However, this thesis considers that the absolute time taken to generate each ISM model (7.6 minutes on average) still represents a significant improvement over a manual alternative to the RORE approach.

9.3.3.3 Usability of Prototype Tool

Component Explanation. This research did not perform a formal usability evaluation of the prototype, but instead report the experiences of this thesis of applying the prototype tool to the case study. While this naturally biases this evaluation to a certain extent, this research makes a considerable effort to overcome this bias and to report as honestly as possible the issues which were encountered when utilising the tool. This research identifies limitations in four areas with respect to the usability of the prototype RORE requirements task assistant. Firstly, this thesis considers an issue which is raised by Maiden [Mai92] in relation to the design of tools to support reuse: the need for explanatory mechanisms which help users to understand the purpose, function and structure of reusable components.

Maiden's Intelligent Reuse Advisor [Mai92] (IRA, subsequently AIR [SM98]) provides dialogues which clearly explain in natural language the structure and purpose of each individual component. These dialogues are supported by a range of intelligent algorithms for generating explanations [Mai92]. The prototype RORE tool attempts to provide support to requirements engineers for comprehending the function of reusable knowledge structures. However, this support took the form of a set of Viewer components which allowed requirements engineers to explore the contents of reusable knowledge structures and models. No clear summary is available within the prototype RORE tool to support this task. This research found that this was not an effective means of supporting the choice between two potentially relevant components, because it left the author to reason from scratch about the relevance of a structure to a particular context. This thesis considers, however, that there are circumstances in which such a tool might

be preferable to a high-level summary, in particular situations in which the choice between two components rests on low-level nuances which require an exploration of the low-level facts which two structures comprise. A tool to summarise a reusable knowledge structure would, therefore, complement the current Viewer-based approach.

Familiarity with Components. Because of this limitation in the power of the prototype tool to explain reusable structures at a high-level this research tended, when enacting the case studies, to avoid utilising the explanatory tools where necessary. In particular, throughout the testing phase of the development of RORE the user became sufficiently familiar with a handful of the reusable structures that they no longer needed to refer to any form of explanation to make relevant decisions. While, therefore, the RORE prototype may lack techniques and tools for adequately explaining reusable structures to users in the first instance, the support which RORE provides for utilising generalised and flexible reusable knowledge structures means that requirements engineers can use significantly smaller libraries than might otherwise be possible, and so can familiarise themselves rapidly with a small set of frequently recurring knowledge structures. This improves usability in the long-term because if users instinctively understand from past experience which knowledge structures to apply in a given situation then this is preferable to a situation in which users must learn about a new aspect of a framework each time they apply it.

Clarity of Notations. Another factor which adversely impacted on the utility of the RORE approach was the lack of a clear notation for representing reusable knowledge structures, or indeed the models which were being generated. While the user did develop a technique for generating graphical representations of the models that were produced during a RORE session, the technique was a prototype for the purposes of producing graphics for this thesis, rather than a quality long-term solution. Graphical output is useful because it provides an abstract view of reusable structures and models which can be more rapidly digested than can detailed textual descriptions (recall the common observation that "a picture describes a thousand words"). As such, the lack of a clear graphical representation of models and reusable structures further undermines the ability of RORE to summarise knowledge structures to requirements engineers.

Rigidity of the Cycle Structure. Aside from the lack of explanatory tools provided by RORE, another issue which this research identified as affecting the usability of the

prototype tool is the rigidity of the RORE cycle structure. In specifying the cycle structure this thesis did not consider fully the possible exceptional cases that may arise during the course of a RORE session. This thesis identified the possibility that an Information Requirement might be generated which cannot be satisfied by *any* reusable knowledge structure in the currently LTM. During the case study, a number of other such exceptional cases emerged. One concrete example of such a case can be found in the case of the File Transfer example for which an information requirement was generated stipulating the need to produce a monitor report, despite the fact that all domains within the File Transfer example had a false monitor tag. In this case the tool compels users to proceed with the remainder of the cycle, to satisfy an information requirement that did not make sense given the context. No facts were ultimately generated, but the process wasted time which might have been spent more productively. This is partly a limitation in the specification of the analysis rule concerned which did not check the monitor tags associated with domains, but it is also indicative of a limitation of the implementation of the prototype tool. This implementation forces users to follow the specified procedure precisely and offers them little control over the sequence of events within the process and so can create uncertainty in some situations. The implementation can easily be resolved, but the precise extent to and manner in which a RORE tool should control the sequence of events within a RORE session remains an open question.

Lack of Rollback Operations. The rigidity of the RORE cycle structure is further compounded by the lack of any support for rollback operations. This proved problematic in two scenarios. Firstly, during the initial testing phase of the RORE prototype this research encountered a number of scenarios in which the execution of production scripts, or the application of a model chunk, failed for one reason or another (typically due to a bug in the prototype implementation). It would have been useful in this circumstance to have a means of rolling back the changes which had been made by that operation thus far so that the bug could be fixed and the cycle recommenced. As it was, such a feature was not available and thus the example had to be commenced again from scratch. While this was primarily a problem in the test phase of development, this thesis anticipates that similar issues may well occur in a number of real-world scenarios (for example, where a bug in a production script exists).

A second case in which some form of rollback feature would be desirable is in the case of user error. There were a number of points when running the case studies during

which the user made an error, typically applying the wrong model chunk, and was then compelled by the tool to integrate the resultant facts. User error is a fact of life in software engineering, and so strategies for coping with this when it occurs must be provided in order for any tool to be truly usable. As such, the lack of a proper "undo" function impacted to an extent on the ease with which the user was able to generate the models for each example.

9.3.4 With Respect to Practicality

9.3.4.1 Performance of the Prototype Implementation

To support the aims of this thesis it is desirable that the RORE prototype be not only theoretically significant, but an approach which, with some further development, could be exploited in a practical setting. This evaluation has discussed to a certain extent the performance of the RORE tool but to support any conclusion that the approach can be theoretically practical it is desirable to undertake a more rigorous analysis of the performance of the RORE prototype. A particular concern was in relation to the performance of some of the underlying technologies: specifically the Pellet reasoner. Performance issues have been discussed within the OWL community and some benchmarking work has been done (see, for instance, [Pan05], [GHT06], [LS08]) and this work has generally shown that Pellet suffers from performance limitations when reasoning over certain kinds of axioms, and over large-scale ontologies. No reasoner is without its limitations, and Pellet was chosen to implement RORE because it is one of the most comprehensive and well-established reasoners for OWL.

While attempting to test the RORE prototype this research encountered performance issues in several areas - not just constrained to Pellet, but also in relation to other OWL libraries - which significantly impacted on the performance of the RORE prototype. Initially this research had attempted to execute the RORE prototype on a machine with an Intel Pentium Core 2 Duo (2.8 GHz) CPU, and 2048 MB of DDR memory. However, this attempt failed completely. The RORE prototype was unable successfully to complete a single full example while running on this machine. After three RORE cycles, both the memory and CPU were operating at full capacity, and after a further one or two cycles the effort invariably failed completely resulting in a Java memory exception.

To continue testing the RORE prototype, therefore, this research opted to continue the experiment on a significantly more powerful machine (Intel Core i5-2400 4 Core

(3.1 GHz); 8192 MB memory). This research attempted to run the same examples on this second machine, but the error persisted. Setting the Java maximum heap size to their maximum possible capacity under a 32 bit architecture transpired to be insufficient, and the out-of-memory error continued to plague the effort. This research also adopted a strategy for executing the RORE tool whereby the SWRL rule for implementing inheritance (see Section 9.3.5), which was introduced during the testing phase, was removed from long-term memory. The rule was switched back into LTM only when it was actually needed. With these changes, this research was able to achieve acceptable performance (from the user perspective) by running RORE under the most recent JVM (7.2) and allowing it to utilise up to 6.5 GB of memory.

However, when switching in the SWRL rule on the final cycle of each example (it is needed to support the retrieval of all facts of type "Resource" regardless of the specific kind of resource the fact represents) RORE still experienced a significant slow-down. Figure 9.1 illustrates the problem. It shows the CPU and Memory usage data that this research gathered as part of a profiling exercise to explore the performance issue further. The profiling exercise used the Autopilot example to test the RORE prototype.

The figure is labelled to indicate the start of each cycle. What the figure shows is that memory usage grows rapidly in the first cycle but then remains relatively stable throughout the remaining cycles of the example. Finally, in the seventh cycle memory usage roughly doubles from 2 GB (immediately before a round of garbage collection) to a peak of ¿ 4.5 GB of memory usage and an apparent plateau of 4 GB. In comparison to the sixth cycle - which was also a round of elicitation, and which averaged approximately 2 GB of memory consumption - the only significant change was the introduction of the SWRL rule. To test whether or not the dramatic effect truly was the impact of a single SWRL rule this research adopted an alternative implementation strategy which scrapped the rule and introduced instead a programmatic implementation of sub-typing and inheritance. The sixth cycle still offered a slight performance reduction versus previous cycles with this alternative solution, but the performance was significantly improved.

It is the Pellet reasoner specifically, as opposed to any other part of the OWL architecture, which is predominantly to blame for the poor memory performance on the tests because it is the Pellet reasoner that actually reasons over a RORE model and LTM in the current version of the prototype. However, it is not the only source of potential memory leaks. This research encountered, throughout the testing phase, bugs in the RORE prototype which appeared to stem from the Protg Code Generation library

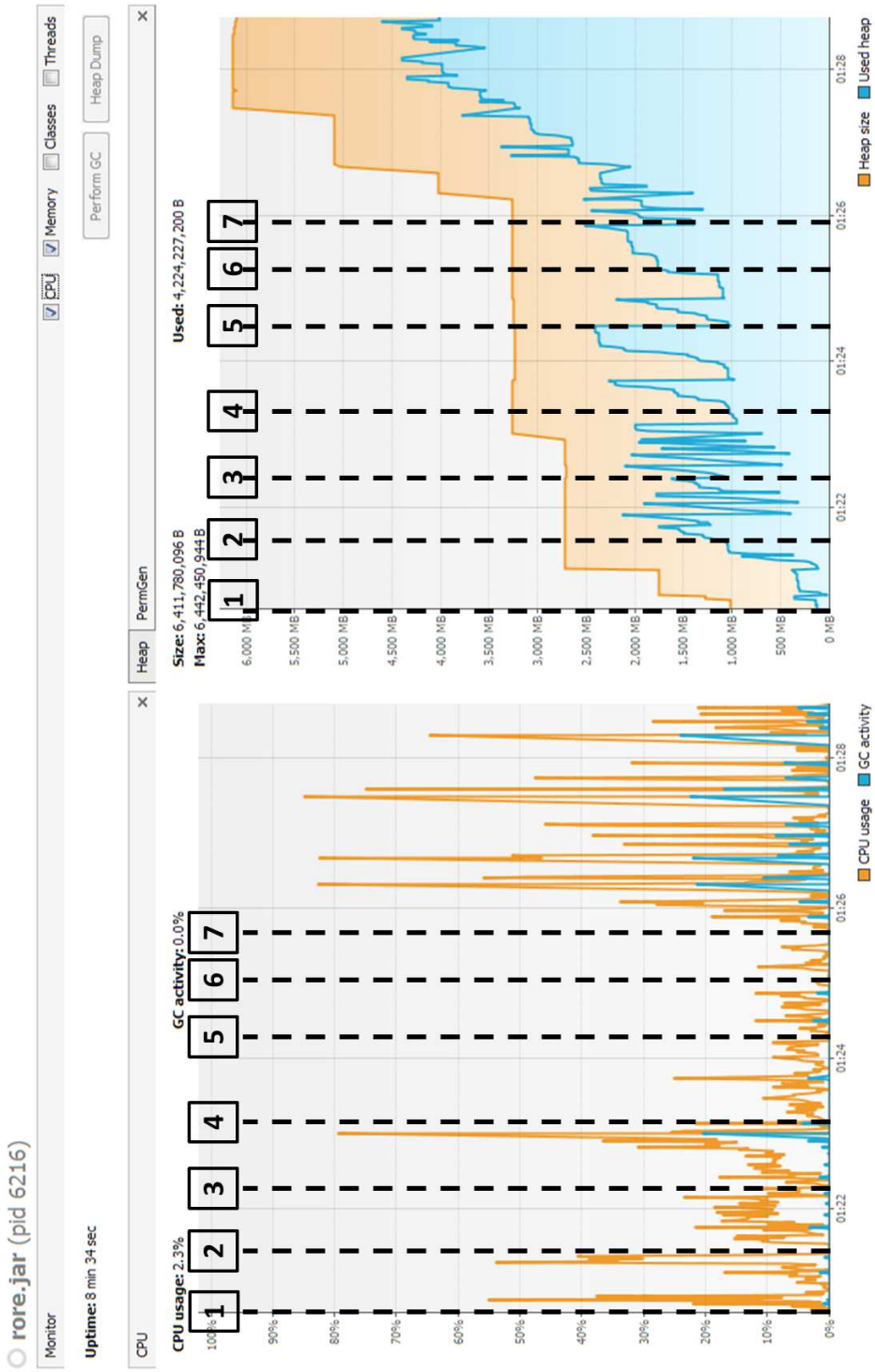

Figure 9.1: Performance data collected from the profiling exercise

on which the prototype is built. Specifically, this research discovered when testing the production script interpreter which is built into RORE that under certain conditions - the boundaries of which have been narrowed but not yet completely clarified - the Pellet reasoner was not picking up modifications made to a model by the code generation library. Because the source of this issue still remains unclear, no elegant solution has been possible at this stage. Instead, a number of workarounds have been identified.

These workarounds are effective in the sense that they enable a complete RORE session to be executed. However, they involve repeatedly committing an ontology, dropping instances of the OWL API and then reloading the ontology from scratch. The profiling data revealed that in excess of 5000 invocations of the commit and load methods over an ontology were made throughout a single RORE session. This clearly has a substantial impact on the efficiency of any software system (and on the safety of the hard disk!). However, these hacks were written disproportionately into the code for executing production scripts and thus impacted only the first three RORE cycles in each example. They are not sufficient, therefore, to explain the 7th-cycle memory issue shown in Figure 9.1.

9.3.4.2 Scalability of the Prototype Implementation

In light of these performance issues, and discussions in the literature about the potential impact of ontology size on reasoner performance, it is necessary to consider the scalability of the RORE prototype in a practical scenario. Scalability is important because the case study which this thesis presents is not representative in scale of real-world requirements engineering projects. While the specific examples are in fact drawn primarily from real-world software engineering scenarios, these examples tested just one kind of requirements engineering task. In practice, a RORE installation would need to perform well over LTMs comprising the knowledge required to support a range of requirements engineering tasks. This is where the concern in relation to the scalability of RORE is most pointed. This research did not, in fact, gather sufficient evidence to draw firm conclusions about the scalability of RORE in either direction, although the results presented in this thesis do not show any evidence of reduced performance in line with growth in model complexity. Instead, two other factors appeared to have a much more significant impact on performance than did scale.

- Type of axioms in a model/LTM;
- Memory leaks in some part of the prototype code.

In any event, the structure of RORE is such that should any problem of scale arise in future iterations of this work, it can readily be managed through a modular approach. The RORE tool already supports an approach to live-switching of LTMs. As such, there is no reason why a different LTM could not be defined for each different task that a requirements engineering might need to perform. This would reduce the scale of each individual ontology, and this is what matters - if at all - in relation to the performance of the OWL reasoning architecture. Scalability, therefore, does not present a significant threat to the RORE approach.

9.3.5 Refinements Made to the RORE Architecture

During and throughout the testing performed over the case studies, and this evaluative procedure, a number of refinements were made to the RORE framework and its prototype implementation as follows:

- **Introducing of Subtyping with Inheritance**: In the initial design of the RORE framework, as presented in Chapter 3, this thesis took the decision not to introduce sub-typing of fact types. However, in formalising the FunctionalResponsibilitiesAssignment production script it became clear that it would be necessary formally to model Agents, KeyObjects and Structures as subtypes of Objects. This research therefore modified the immutable layer of the RORE knowledge model to introduce the "subTypeOf" property of ComplexTypes. This research also introduced two SWRL rules to the model: the first which enforces subtyping (all facts of type, a, which is a subtype of a fact type, b, are also facts of type b), and a rule which enforces inheritance (if a is a fact type which is a subtype of the fact type b, and b has the property p, then a also has the property p);

- **Elicitation Stimulus Subtypes**: The initial design of the RORE framework specified just one type of elicitation stimulus: that which required the requirements engineer to specify model chunks as responses. This proved inadequate as a solution when formalising the OSM-ISM transformation procedure because more common than requiring users to respond with novel information, was the case in which users were required to select from, or adapt, existing information. This research therefore introduced two new kinds of elicitation stimulus: a fact editing stimulus, which selected facts from either the source or target model to be

adapted; and multiple choice stimuli, which selects facts from either the source or target model from which a response may be selected;

- **ASK/SELECT Queries**: The initial concept behind the design of production scripts was that each script would transform a single set of facts to produce a single set of output facts. For this reason, ASK/SELECT queries were not included in the initial specification of the production expression syntax which were presented in Section 6.2.3. However, in formalising the OSM-ISM transformation procedure it became clear that this model limited the expressive power of production scripts. This research therefore introduced into the production expression syntax ASK and SELECT queries as new types of value. Each query may be fired over either the source or the target model;

- **Template Placeholders**: The initial specification of production expressions provided no construct for string concatenation. This proved a severe limitation in implementing production scripts as it meant that only literal values could be accepted as the names and values of variables, and in queries. However, it was often desirable (in particular in constructing fact names) to use information from the input facts to adapt string values to the particular context of the source model. This research therefore introduced template place-holders which allow the values of vars to be substituted into a string literal prior to that literal being evaluated. This allowed, for instance, the names of classes generated by the "GenerateObjectModel" production script to be customised according to the name of objects in the input source facts, further advancing the generality of the approach.

These refinements strengthen the RORE framework in light of some of the limitations which were uncovered throughout the evaluative process. Accordingly, the concerns which Section 9.3.1.3 expressed about the expressivity of the RORE knowledge model were largely resolved. RORE supports a greater degree of generality as a result.

9.4 Comparison with Related Work

This chapter makes a concerted effort objectively to evaluate the RORE framework against the criteria which this thesis laid out and defined in Section 9.2. However, it is also necessary in evaluating RORE to consider the extent to which the attributes that have been highlighted by this evaluation represent an improvement over existing

approaches within the literature. This section therefore seeks to undertake precisely this task: the comparison of the RORE approach, as this thesis has evaluated it thus far, against the three main alternative approaches to reuse at the requirements level:

- The Domain Theory;
- Problem Frames/Problem-Oriented Software Engineering;
- Requirements/Requirements Engineering Patterns.

This section is concerned less with the pragmatics of the RORE approach as the prototype requirements assistant has proven - subject to some future optimisation and refinement - to be a basically feasible approach to real-world requirements engineering. Instead, therefore, this section focuses specifically on the questions of generality and systematicity. The introduction identified these properties of a reuse approach as being those RORE should be designed to balance.

9.4.1 Comparison with the Domain Theory

9.4.1.1 With Respect to Generality

Method Generality. Perhaps the area in which RORE can most readily be compared to the Domain Theory is in the area of method generality. Recall that method generality as defined in Section 9.2 is the extent to which an approach supports or facilitates its own tailoring or adaptation to support the method of a requirements engineer's choosing. Section 9.3.1.2 argued that RORE does indeed support method generality since by virtue of the fact that its knowledge structures sit at the level of the meta-model rather than at the level of the model. New requirements model types can be defined within RORE and the reusable knowledge structures which will be used to transform instances of those model types can be specified in the same terms as the model types themselves. This arrangement provides a layer of indirection between the underlying RORE activities and the specific knowledge representation languages through which concrete knowledge instances will be expressed and so fully supports the adaptation of the RORE approach to fit a particular method.

The primary constraint on this is that the method must express knowledge in terms of an acyclic graph. Within the field of requirements engineering, this constraint is not a significant restriction, particularly because many requirements notations (including the Domain Theory [SM98], KAOS [MVL97] and i* [Yu93]) have been formalised

using the Telos formalism [MBJK90] — a formalism for meta-modeling which represents the semantics of modeling notations as an acylcic graph. Furthermore, the context and problem modeling notations described by Jackson [Jac01b] can be conceived of as acyclic graphs, as this thesis illustrated in Chapter 3. As such, the restriction that methods which are formalised in RORE should represent knowledge as an acyclic graph is not a major restriction. However, there are some notations (e.g. natural language and textual use cases) which cannot be readily represented as graphs, and so the constraint does impose a small restriction on the method generality of the RORE approach.

By contrast, the Domain Theory strictly specifies a particular meta-schema for expressing requirements knowledge. This meta-schema evolved from its inception in Maiden's thesis [Mai92] to the initial formulation of a "Domain Theory" [SM98]. Sutcliffe has also significantly elaborated on the kinds of knowledge which may be expressed within the domain theory, introducing the notion of generalised and generic tasks to support the modeling of conceptual processes [Sut02] and information system models to model information processes within a domain [SM98]. Furthermore, Papamargaritis has extended the domain theory to support application generation, and thus post-requirements aspects of software development [Pap06]. He refines, but does not significantly extend, the knowledge schema on which the domain theory is based, and he does not depart from the basic principle which underpins all incarnations of the Domain Theory, namely that the particular schema by which knowledge is to be expressed can be assumed. Finally, Sutcliffe has proposed knowledge claims as a mode of extending the domain theory with open-ended knowledge by providing support for the generalisation and integration of new reusable knowledge [SC99, Sut02]. Inevitably this enhances the richness of the domain theory's library for reuse, but it does not significantly enhance the method generality of the approach. This is so for two reasons: firstly, the claims must be expressed in a rigidly defined template structure which constrains the kind of knowledge which may be expressed in this way; secondly, claims are used, according to Sutcliffe [Sut02], to tag domain models with additional knowledge. While this may increase *what* is expressed about a domain, it fundamentally does not allow knowledge to be expressed in different ways, and so does not support method generality. This thesis concludes, therefore, that RORE represents an improvement on the Domain Theory with respect to method generality.

Task Generality. RORE also represents an improvement over the Domain Theory with respect to task generality. Although this thesis only evaluated the RORE approach

with respect to a single task (the transformation of OSMs into ISMs), Section 9.3.1.2 argued that one can extrapolate from this task to identify other kinds of task which could also be formalised using the RORE approach. This research identified at least two other prominent requirements engineering tasks which RORE could support, and extrapolated from these arguments to a more general class of requirements task which might be supported by RORE. The domain theory is designed from its inception to support two distinct requirements tasks [Mai92, MS94, SM98]:

- Requirements Critiquing;
- Requirements Elaboration.

The process of Requirements Elaboration within the Domain Theory amounts to the process of generating an OSM for an application domain because OSMs are constructed by providing an initial set of facts which are then reified by incrementally refining the match of those to a low-level generic domain model [SM98, Sut02]. The process by which the Domain Theory supports Requirements Critiquing is inherent in this process of elaboration: as requirements engineers and domain experts refine an OSM, they are presented with alternative generic domain models and explanations of each model. This forces them to think hard about the precise nature of the particular problem they are attempting to model and thus compels the requirements engineer to adopt a critical attitude to the requirements they are modeling. Section 9.3.1.2 argued that RORE could readily be used to implement the OSM generation process which is also supported by the Domain Theory since RORE comprises many of the same building blocks from which that process is composed (support for analogical and rule-based matching, support for reuse of model chunks, support for manual fact specification). RORE can therefore be seen as a superset of the Domain Theory as specified by Sutcliffe and Maiden [SM98] in terms of the tasks it supports.

However, Papamargaritis has extended the Domain Theory to support application generation [Pap06]. While this work does indeed introduce new tasks into the realms of the Domain Theory, there are two reasons why this cannot be considered significantly to improve the task generality of the Domain Theory, aside from the fact that generality is explicitly not an aim of Papamargaritis' work: he intentionally scopes his solution to support the generation of applications within a single OSM family. Firstly, Papamargaritis adopts a "great leap forwards"-style approach. As such, the tasks from which his generative extension is composed are coarse-grained and tighlty-coupled. As such, while Papamargaritis' extension represents a significant functional extension,

it represents a single coarse-grained task from which the decomposition of decoupled components would require significant work. Secondly, the extension which Papamargaritis presents contains no measures which allow users to extend the approach further than does Papamargaritis himself; that is, no additional tasks can be introduced beyond those which the extension itself directly comprises. This thesis concludes, therefore, that while the Domain Theory has been extended over time to support tasks beyond those for which it was original formulated, this has not been made possible through built-in extensibility, but instead has required adaptation of the theory, its schema and its processes themselves. This is in stark contrast to the RORE approach which supports the integration of new tasks by means of parameterisation in the form of model-type and task type specifications.

Domain Generality. A comparison of the Domain Theory with the RORE approach in respect of domain generality is a comparatively tricky task because it depends on what grounds the comparison is made. For instance, one approach would be count the range of domains which can be represented by the restructured version of the Domain Theory presented in Chapter 7, and compare this to the range of domains represented by Sutcliffe and Maiden's OSM library [SM98]. The results in this case come out as follows:

- **RORE** = 7 Goal Types x 4 Domain Types x 3 Agent Types = (7*4*3) = 84 abstract contexts;
- **Domain Theory (as in [Sut02], Appendix A)** = 28 abstract domain models spanning 3 levels of abstraction.

From this perspective the RORE approach clearly trumps the domain theory by a ratio of 3:1. However, the refined domain theory as this research applied it during this evaluation is not a fundamental part of the RORE approach but is, rather, one particular configuration of the RORE approach with a particular set of reusable knowledge structures. It would be perfectly possible (albeit pragmatically nonsensical) to configure the RORE tool with a set of reusable knowledge structures that were significantly *less* general than those provided by the Domain Theory. Similarly, while the *tasks* which the Domain Theory supports cannot easily be extended, the knowledge structures which it provides can be since abstract domain models in the Domain Theory are described as data in Prolog [Mai92]. As such, one could easily enough adapt the Domain Theory's knowledge base, at least to include abstract domain models to make

up the difference between the refined domain theory and the formulation by Sutcliffe and Maiden [SM98].

However, this thesis has already argued that the RORE approach represents a superset of the Domain Theory in the sense that it is generalised from it. To demonstrate that this is indeed true, as opposed to, for instance, RORE being an equivalent set to the Domain Theory, this thesis must demonstrate both that there exist no domains which can be addressed by the Domain Theory but which cannot be addressed by RORE, and secondly that there *are* domains which RORE can cover which the Domain Theory cannot. This thesis has already argued that each of the tasks which is supported by the Domain Theory can be implemented in RORE, and has also argued that those domains which are supported by the Domain Theory can also be supported by the restructured Domain Theory. That being the case, this thesis concludes that in the latest incarnation of the OSM library (as given in [Sut02]), there exist no domains which cannot also be supported by the restructured domain theory. The work provided by Papamargaritis does not extend the domain generality but instead extends task generality along the project timeline.

It is possible, however, to identify a number of domains which cannot readily be supported by the Domain Theory because of its commitment to a pre-assumed meta-schema for expressing requirements knowledge, and its expressed goal of supporting "transactional" problems specifically. One such domain is the lexical parsing domain. The Domain Theory represents problems in terms of a single kind of binary relationship between key objects and structure objects, as well as in terms of state transitions over these static structures. One could find a means of expressing some aspects of a lexical parser using the Domain Theory representation and library. For instance:

- The reading of a program which is to be parsed by the parser could be represented as an Object Sensing problem, in which the parser "senses" a file containing that program;

- The tokenisation of the program could be represented as an Object Decomposition problem, where the program is the whole object and the individual tokens within that program are the part objects which the whole comprises;

- The semantic interpretation of each token could be represented as an Allocation problem, where each token is the key object and different semantic constructs within the language being parsed are represented as structure objects;

- The construction of an abstract syntax tree to represent the program could be represented as an Object Composition problem in which the Abstract Syntax tree is the whole key object and the individual tokens are the part key objects.

However, the cut of the world which is offered by the Domain Theory abstractions, and the attempt to represent the parsing process as a sequence of transitions over containment relationships between key and structure objects, does not offer either an efficient — or, necessarily, intuitive — representation of the specification for a parser. Furthermore, there are critical parts of the parsing process which the metaschema of the Domain Theory cannot easily express: lexical and semantic checking of a program could not easily be specified using this representation because the Domain Theory supports conditions which are expressed in terms of relationships between key and structure objects, or in terms of the chronology of events.

By contrast, the greater level of abstraction provided by the RORE meta-modeling framework can more readily be adapted to support the modeling of lexical parsing domains, in part because the EBNF specification of a language could be specified as a RORE Model Type in its own right (each construct in the EBNF specification would become a Fact Type within the RORE Model Type, with Property Types representing the relationships between constructs). Thus RORE can support representations which are purpose-built for dealing with lexical parsing domains and so can provide a clear and concise representation of requirements within such domains. This thesis concludes, therefore, that there are domains which can be supported by RORE but which cannot be supported by the domain theory, and that the converse is true. As such the RORE approach really does offer a greater degree of domain generality than do the essential components of the Domain Theory.

9.4.1.2 With Respect to Systematicity

This thesis interprets systematicity in terms of two basic questions:

- Is reuse supported by a repeatable system for selecting and applying reusable structures?
- Is reuse central to an approach's support for a requirements engineering task?

It should be noted that this definition considers only whether an approach is systematic with respect to those tasks it supports. It does not, therefore, duplicate the "task generality test".

When these two questions are applied to compare RORE against the Domain Theory, this thesis finds that the two approaches are broadly similar in terms of systematicity. On the first question, this thesis finds that both RORE and the Domain Theory do indeed provide repeatable procedures to support each reuse procedure: matching, adaptation and integration. This is to be expected, given that the basic RORE processes and mechanisms have essentially been abstracted from the Domain Theory. Both the Domain Theory and RORE provide both rule-based and analogical matching procedures which support the retrieval of context-relevant components. Both the Domain Theory and RORE support adaptation through user dialogues which allow specific facts to be manipulated. Finally, both RORE and the Domain Theory support integration based on matches between abstract facts and target facts.

Similarly, on the second question no significant difference may be found between the two approaches. Bearing in mind that the Domain Theory is designed to support two requirements tasks - elaboration and critiquing [Mai92, SM98, Sut02] - it does so predominantly through reuse. Every procedure within the Domain Theory's implementation - the Advisor for Intelligent Reuse (AIR) [Sut02] - is geared towards supporting or guided by reuse. The approach consists of four broad phases: Fact Capture and Matching; Explanation; Model Selection and Explanation; Critiquing. Fact Capture exists only to support Matching; Matching is the process by which an abstract Domain Model is retrieved; Explanation exists to assist a user in understanding the structure and function of an abstraction; Model Selection is the process by which a user manually makes the final choice as to which abstraction is most appropriate; and Critiquing forces the user to reconcile any inconsistencies but does so by determining whether or not a concrete model neatly fits an abstract model. Although the RORE procedure is a little different in structure, it is no less reuse-oriented. Indeed, every requirements task in RORE is supported by a reusable knowledge structure, whereas every lower-level task is directly designed to support these reuse-driven tasks. This thesis concludes, therefore, that both RORE and the Domain Theory represent approximately equal degrees of systematicity.

9.4.2 Comparison with Problem Frames/POSE

9.4.2.1 With Respect to Generality

Method Generality. Unlike RORE and the Domain Theory, POSE does not stipulate any particular schema or language through which problems, their requirements or their

solutions should be expressed. Indeed, as Hall, Rapanotti and Jackson state:

> POSE is a formal system for working with non-formal and formal descriptions. Moreover, formality may sometimes be appropriate when strict stakeholders such as regulatory bodies governing the development of the most safety-critical of software are involved. However, as we know from the real world, only when it is focused is formality appropriate.

Transformations in POSE describe relationships that hold between three sets of conditions: a premise, a conclusion, and a justification. Each transformation is an abstract pattern which describes the attributes which must be expressed by an artefact in order for the conditions of the transformation to be satisfied [HRJ08]. Furthermore, while Hall and Rapanotti have developed a tool - POELog - to support the POSE approach, this is an implementation, and not a fundamental part, of the POSE approach. As such, POSE makes no inherent commitments as to the form in which an artefact should be expressed.

By contrast, the RORE approach does constrain the range of forms through which can be expressed the knowledge over which it operates. Recall that the knowledge which is to be manipulated by RORE must be expressible in graph form: that is, as a set of nodes which are interconnected by a set of arcs over those nodes. There are, therefore, methods which could not readily be formalised in RORE which POSE can handle readily. In particular, while in theory it may be possible to formalise natural language in graph form (nodes representing letters; properties representing relations, for example), in practice natural language has never successfully and fully been formalised. POSE, however, does not depend on a formalised expression of the language through which knowledge is expressed and so can handle methods depending on natural language quite readily. Specifically, POSE may lend itself neatly to early-phase requirements engineering where requirements are typically informal and poorly defined.

Domain Generality. One way of viewing POSE is as an instance of a more general framework, Problem-Oriented Engineering (POE), which supports engineering and design in the general case [HR09]. POE, and POSE by extension, is itself rooted in Jackson's Problem-Frames Approach (PFA, described in [Jac01b]), and accordingly draws on the basic structure which Jackson outlines for describing problems, their solutions, and their requirements [HRJ08]. Although PFA outlines five generic problem classes (the "Problem Frames" or "PFs"), Jackson explicitly states that these are

simply the frames that he has observed in his own experience and thus that it may be possible to identify additional frames [Jac01b]. In any event, the method which Jackson advocates for problem decomposition, solving and solution recomposition is neither implicitly or explicitly dependent on any specific PF. PFs are important within PFA because they provide, in effect, a stopping condition for problem decomposition: problem decomposition continues until sub-problems can be mapped onto a known PF [Jac05]. It is not, however, a necessary condition that these PFs be the five PFs which Jackson himself describes. As such, PFA can readily be extended in the event that a particular sub-problem is identified which cannot be mapped onto a currently-known PF by specifying the new PF and the relevant frame concerns. This will require, as Sutcliffe notes, a process of abstraction and design of the PF [Sut02].

The generality of POSE is naturally spoken for by its relationship to the more general POE framework: that the basic framework which POSE instantiates is claimed to be applicable to a range of other engineering disciplines attests (in principle) to the generality of the approach. This generality is achieved by two related strategies. The central procedural component of POSE - the transformation - is described by a ternary relationship between three sets of conditions: the premise; the conclusion; and the justification. Firstly, each is a set of conditions which holds over either two sets of artefacts, or two states of a single set of artefacts. These conditions are abstract in the extreme in that they are expressed simply in terms of a small set of symbols and the relationships between them, each representing general constructs such as “Domain”, “Problem”, “Solution” or “Requirement” (see, for instance, the “Domain Description Interpretation” transformation in [HRJ08]). The consequence of this is that, as Section 9.4.2.1 discussed, a transformation can be applied to any artefact for which a reasoner exists (human or otherwise) that can infer the transformation conditions from the specific knowledge expressed within the artefact. Secondly, the POSE method has been defined in two ways: in a general sense, as discussed in [HRJ08], such that transformations are incrementally applied by matching problem descriptions to transformation justifications until a solution to each sub-problem has been elaborated; or by the POE process pattern, which is not specified in terms of any specific transformation, but rather must be composed from collections of transformations as discussed in [HR08]. Given these two properties of POSE, it is extremely difficult to identify any application domain to which the POSE approach could not, at least in principle, be applied assuming a manual application of the approach.

RORE itself shares with both PFA and POSE these two critical properties:

- The ability to extend the tool to previously unsupported domains by introducing new knowledge structures;
- Indirection between the method and the specific knowledge about how the method should be enacted.

As such, RORE should in principle exhibit an approximately comparative degree of generality. However, there is a good reason why this is not the case: whereas POSE does not constrain the language through which problem descriptions are expressed [HRJ08], RORE does Section 9.4.2.1 discussed. There is at least one example of a domain type which RORE could not easily model, but which POSE would not struggle with: the natural language processing domain (NLP). Section 9.4.2.1 argued that the graph-based representation on which RORE depends is inefficient at best for expressing the rules and constraints which underpin NLP. By constrast, POSE makes no such assumption about representation, and indeed the PFA's ontological foundation expressely includes the notion of a lexical domain. While not all such domains necessarily assume natural language (formal languages are also instances of a lexicon), they are certainly included in the range of lexical domains. This thesis, therefore, that RORE is also *less* domain general than POSE.

Task Generality. Consideration of the task generality of RORE as compared with that of POSE depends significantly on how precisely this thesis defines the concept itself. "Task Generality" may be defined in at least two directions: vertically, that is in the dimension running from requirements through to design; or horizontally, within the requirements phase itself. POSE explicitly provides support for post-requirements software engineering (solution elaboration, for instance, is a task which is generally held to be situated with a "design" or "architecture" phase, as distinct from requirements engineering: indeed, the STRAW workshops in '01 and '03 investigated techniques for moving from the requirements phase to the architecture phase; and Hall et al. have investigated the application of PFs to this end [HJL$^+$02]). RORE, by contrast, is intentionally focused within the requirements engineering phase. Although it may well be generalisable to subsequent phases, and this is one potential avenue for future research, this thesis has neither demonstrated this to be so, nor has the RORE framework been designed with this goal in mind. For now, therefore, the null hypothesis is that POSE is more task general along the vertical dimension than is RORE.

However, there is a crucial difference between the two frameworks which ensures that, within requirements engineering RORE is actually capable of supporting

a broader range of tasks than is POSE. RORE supports two basic classes of task - analysis and action. The key distinguishing feature between the two classes is whether or not the task changes the state of the target model: analysis does not, whereas actions do. The function of analysis is evaluative; the function of actions is productive. The fundamental structure which is used to represent procedural knowledge in POSE, however, is the transformation [HRJ08]. Transformations comprise both a precondition and a postcondition and, as such, are structured fundamentally to represent state changes over an artefact or set of artefacts. While transformations could be used, therefore, to represent non-transformative inferences (conclusions about the state of an artefact which do not actually change that state), there is no precedence in the literature on POSE for such an interpretation of a POSE sequent, and such an interpretation would represent an important departure from the stated POSE method. Certain POSE transformations *imply* an analytical element: consider the "Domain Interpretation" schema, for instance, which according to Hall et al [HRJ08] re-expresses a transformation "in some way to make it more suitable for problem solving". However, this analytical element is not explicitly modelled within the POSE approach, and as such cannot be considered explicitly to be supported by the approach. This thesis concludes, therefore, that RORE is *more* task general than POSE.

9.4.2.2 With Respect to Systematicity

RORE may be said to be *more* systematic than POSE with respect to reuse in terms of both centrality and repeatability. In POSE requirements engineering, and subsequently software specification, is driven by reuse in the sense that each POSE transformation instantiates an abstract transformation schema [HRJ08] which is specialised for a particular problem specification. These transformations are then mapped onto a concrete problem description by substituting the terms in that description for terms in the conclusion of the abstract transformation. The transformation is completed by restructuring the original state of the problem description to satisfy the structure describe by the premise of the transformation. While the preceding stages are entirely reuse-oriented, however, this final stage is not, in the sense that POSE offers little methodological guidance on precisely *how* that transformation should be realised. When compared with the production scripts in RORE, which provide step-by-step guidance linking the precondition to the postcondition, POSE transformations offer very little concrete guidance: this is not, in fact, their purpose which is, rather, to provide *justification* for a design solution. This thesis argues, therefore, that reuse plays a more central role in

the RORE framework than it does in the POSE framework which leaves significantly more reasoning up to the discretion of the software engineer than does RORE. For this same reason, this thesis also argues that RORE lacks repeatability: manual reasoning is rarely precisely repeatable.

There is one scenario in which the claim that POSE leaves the hard work up to a software engineer is not so true: the case in which the engineer is applying the procedure without support of an automated tool such as POELog [HR08]. As Hall et al. note, it may be desirable - particularly where an organisation frequently produces systems of a similar kind - to provide an automated implementation of POSE [HR08], and POELog is designed to support this case [HR08]. POELog is a Prolog encoding of POE which provides predicates (step, problem, rationale, concern, domain) through which POELog "applications" can be defined. Each such application comprises concrete instantiations of a POE transformation, specified in terms of the POELog predicates, which collectively form a program for producing a POELog development tree from a null (unsolved) problem. The application can then be run repeatedly to generate development trees from the same POELog specification. POELog applications can thus be seen as a form of product line engineering [CN01] in which a generator is built to support the rapid generation of artefacts within a particular application domain. When utilised in this scenario, POSE does indeed support a similar degree of systematicity to RORE because each step in the realisation of a development tree by POELog is achieved by *reusing* knowledge which is encoded in the POELog application. Nonetheless, that the degree of systematicity which POSE offers is dependent on the particular implementation of POSE which a developer chooses is indicative of the fact that any extensive systematicity which POSE offers beyond a well-defined procedure for selecting, retrieving and applying transformations in the general case must be accidental, and not essential, to the POSE approach.

9.4.3 Comparison with Requirements Patterns

9.4.3.1 With Respect to Generality

When compared with respect to generality, RORE - and indeed any other systematic framework - is unable to compete with pattern-based requirements in general engineering due to the sheer lack of constraints on the specification of patterns. While some authors have proposed pattern templates in the requirements literature [SHC+10, SL11], these simply stipulate the components which must be present within a requirements

pattern and do not constrain how knowledge can be expressed or the domains to which it may apply. Chapter 2 surveyed requirements patterns which conveyed a vast array of different kinds of requirements-level knowledge:

- Content reuse for elicitation [Fin88, MVL97, CDA93, RM93, RPR98, KSN10];
- Non-functional requirements realisation [Yu11];
- Support for identifying security requirements [SFO03];
- Support for requirements elaboration [Mai92, DVL96];
- Patterns for enforcing specific NFRs or architectures [KC02, LG06];
- Patterns providing knowledge about the requirements process itself [HL04];
- Patterns addressing organisational aspects of requirements engineering [KGM03].

I also surveyed requirements patterns which were domain-independent as well as concern- or domain-specific, and in the latter case which covered domains as diverse as:

- Multi-agent applications ;
- Non-functional concerns and their realisation [Yu11];
- Security requirements [SFO03], robustness requirements [SF10];
- Requirements interactions [DPM05];
- Requirements for COTS Systems [MBFQ08];
- Call-for-Tender Systems [RMBFQ09a];
- Service-oriented information exchange requirements [MBLN06].

It is clear, therefore, that there exists a broad range of patterns from which requirements engineers can acquire reusable knowledge and guidance. Given this fact, pattern-based engineering, therefore, can achieve a significantly greater degree of generality in total than can RORE because the pattern paradigm imposes significantly fewer constraints on the reusable structures that it supports, and on the process by which those structures are reused. New patterns may always be prepared to support

novel tasks, and in practice requirements engineers can adopt any pattern they like because pattern-based engineering offers no agreed-upon system for selecting or applying patterns to a given scenario. By contrast, no task can be achieved in RORE unless the reusable knowledge structures are available in long-term memory to support that task and so RORE enforces systematic reuse. Furthermore, reusable knowledge structures must be specified within the RORE formalism which may not be ideally suitable for supporting the kinds of tasks for which some requirements pattern libraries exist.

9.4.3.2 With Respect to Systematicity

The comparison between pattern-based engineering and RORE can be considered from one of two angles. If one were to take the position of a purist, then all patterns would be organised into an Alexandrian-style pattern language [Ale77] such that patterns were inter-related according to dependencies between them such as "problem-solution", "alternative", "requires", "replaces", "equivalent" and "occurs-with" relationships. Indeed, a number of such pattern languages have been proposed in the requirements literature [Yu11, SFO03, RRJ03, MBLN06, Zha11]. Utilising patterns as part of a broader pattern language is naturally the most systematic form in which patterns can be utilised. The language organises patterns into a reuse-driven framework which guides the engineer through a process of problem solving, one pattern at a time. This satisfies the goal of centrality - the problem-solving process is driven by the application of patterns to transform a problem description - and the goal of repeatability as, if the pattern language is well-designed, then any user should be able to follow the process with relative ease. One of the most comprehensive pattern languages that has been proposed in the software engineering literature - the Pattern-Oriented Software Architecture series [BHS07] - comes not from the requirements engineering literature, but from the literature on software architecture; however, it illustrates the point nicely.

In practice, however, it is likely that most requirements engineers, where they reuse knowledge at all, do so opportunistically rather than as part of a coherent and systematic pattern language. That this is the likely scenario is indicated by the widespread acknowledgement of the pattern concept in the requirements literature (see [SM98, Jac01b, RPR98, MVL97, KSN10, KC02, DPM05, RMBFQ09a], for instance) but the relatively low range of pattern languages proposed. It is also increasingly likely the more patterns that are proposed in the literature, as Agerbo and Cornils have argued [AC98]. In this opportunistic scenario, the use of requirements patterns has little to offer in the way of systematicity: the approach is unsupported by effective and general

tools, and requirements engineers select and reuse patterns with which they are well acquainted, rather than choosing those which best fit the circumstances.

It is difficult to call, therefore, the extent to which RORE compares favourably with requirements patterns approaches because the state of art is defined by a very broad church of pattern-based approaches, as well as disparate patterns, and pattern languages organised to different levels of formality. However, given the likely state of practice, RORE represents a probable improvement on most contemporary and practical approaches to pattern-based requirements engineering.

9.5 Extending the Evaluation of RORE

The evaluation of RORE which has been presented in this thesis is limited in the following ways:

- The evaluation has considered the application of RORE to transfer just two types of model (the OSM and ISM);
- The evaluation has considered just one type of transformation;
- The evaluation has considered the application of RORE within just three small-scale examples;
- The evaluation was a self-study in which the author applied the tool himself, and did not include other requirements engineers.

The evaluation as it is presented in this thesis has been sufficient to validate that the RORE approach can indeed provide a powerful, reuse-driven approach to the transformation and production of requirements artefacts within the assumptions discussed in Section 3.2. However, the restricted scope of the evaluation means that certain questions about the extent to which RORE might be applicable, and in particular in commercial practice, cannot be readily answered at this stage. Further evaluation would seek to answer the following questions:

1. What other requirements models can RORE be applied in order to produce and/or transform?
2. In particular, could RORE be applied to provide tool support for the POSE approach (as a major alternative to RORE, e.g., by encoding the POE Process Pattern within RORE)?

3. What requirements engineering activities (aside from the transformation of OSMs into ISMs, and more generally the production of software specifications from requirements models) can, and cannot, be supported by the RORE approach?

4. Are there any application domains which RORE fundamentally cannot support?

5. What problems and limitations arise when the RORE prototype tool is appplied in the context of commercial practice?

6. How "usable" is the tool when utilised by other requirements engineers (i.e., aside from this author)?

The evaluation presented in this thesis has begun to propose answers to some of these questions. For instance, in Section 9.3.1.2 this evaluation argued that RORE could be applied to support the refinement of KAOS goal models by reuse of requirements fragments, and Section 9.4.1.1 argued that RORE could be applied to generate the specifications for lexical parsers. These suggestions have been proposed in line with the guidance of Gomm et al [GHF00] — that one can generalise from observed to unobserved cases based where the salient properties of the observed case are typical of the unobserved cases to which the generalisation is made. However, the arguments and comparisons made in this evaluation have been superficial, and so further confirmation of the proposed answers in this thesis would be desirable.

Furthermore, other questions remain unanswered — even superficially — by this evaluation. This thesis therefore recommends that the evaluation could be extended, quickly and efficiently, in the following ways as part of any further work:

- To answer Question 6 above, conduct a usability study with requirements engineers using the OSM-ISM transformation task — which has already been formalised using RORE as part of this research — as the basis for the study;

- To address Question 1 above, show that RORE can be applied to formalise the KAOS model type and the reuse-driven Goal Refinement method described by Massonet and Van Lamsweerde [MVL97]. This would be efficient because the KAOS language, and the goal refinement patterns, have each been formalised using Telos [MBJK90] and so the existing formalisations need only be adapted to RORE;

- To address Questions 1 and 3 above, extend the RORE-formalisation of Jackson's Problem Frames approach to include Context, Problem and Solution diagrams and explore the extent to which RORE can be used to support the transformation and refinement of models across all three stages of Jackson's method;
- To answer Question 5 above, provide the RORE tool (and some basic training) to requirements engineers for application on real-world, small-scale requirements engineering projects. Collect and evaluate the logs which the RORE tool produces as a result of these case studies to determine how effectively the RORE approach supported industrial practice.

9.6 Summary

This chapter has validated the RORE approach by discussing the results of the demonstration of RORE that was presented in Chapters 7 and 8. A set of evaluation criteria were presented in order to evaluate RORE in an objective fashion. These evaluation criteria were generality, systematicity, utility and practicality. For each criterion, a definition was given and specific metrics were identified against which RORE would be validated. These criteria were then assessed against data which was gathered through the demonstration of RORE in Chapters 7 and 8. This evaluation suggests that RORE offers an acceptable level of generality and without significantly affecting utility. However, the evaluation also highlighted some practical issues relating to the usability and spatio-temporal performance of the prototype Requirements Task Assistant.

In Section 9.4 the results of this evaluation were used to compare the RORE approach to three other requirements reuse approaches: The Domain Theory, POSE and requirements patterns as a general approach. This discussion highlighted that RORE generally satisfies the main aims of this thesis by offering an improvement over existing approaches with respect to the balance between generality and utility. While RORE did not necessarily perform better against any one other approach on all metrics, where RORE performed less well on some generality metrics than other approaches, RORE was found to perform better on utility metrics. In particular, this discussion found that RORE offers domain and task generality approaching, but not quite equalling, that offered by POSE while offering significantly better utility than POSE. Furthermore, RORE was found to offer utility rivalling that of the Domain Theory while offering significantly better task and method generality. The discussion concluded that no systematic approach to requirements reuse could beat the generality offered by the

opportunistic reuse of patterns, but that RORE offers significant improvements over opportunistic pattern reuse with respect to utility. This thesis concludes, therefore, that RORE has successfully achieved an improved trade-off between generality and utility when compared with existing requirements reuse approaches.

Chapter 10

Conclusions

10.1 Achievements

10.1.1 Improved Requirements-Level Reuse wrt. Generality and Utility

The most significant achievement of this thesis is the proposal of a novel approach to requirements-level reuse which offers an incremental improvement over existing approaches with respect to the fundamental trade-off that exists between three conflicting properties — generality, systematicity and utility — while ensuring that a practically feasible prototype tool can be produced. Chapter 9 evaluated this approach and pointed to two major conclusions:

- When compared to the existing approaches to requirements-level reuse which offer the greatest generality, RORE is slightly less general but offers a significantly greater degree of utility;
- When compared to the existing approaches to requiremenets-level reuse which offer the greatest utility and systematicity, RORE retains a comparable level of utility while offerring a favourable degree of generality;
- Despite some practical limitations of the current version of the prototype tool, it is feasible that a practically viable version of the tool could be developed, as the limitations are accidental and not essential.

The central achievement of this thesis has, therefore, been to offer an approach to requirements-level reuse which improves on contemporary approaches by offering a

blend of generality, utility and systematicity which ensures that:

- Systematic reuse is supported across a wide range of application domains;
- Detailed guidance is given for engineering-by-reuse using the approach;
- Reducing, significantly, the effort required to produce requirements artefacts versus the production, without reuse, of those artefacts;
- A high degree of practicability is feasible with respect to the memory and temporal performance of the prototype tool.

The approach represents an important contribution to the requirements reuse literature. As Chapter 1 discussed, an effective approach to requirements-level reuse would be highly beneficial both to requirements engineering and to the reuse literature in general. Generality is desirable because it allows requirements engineers to utilise a single framework to support a wide range of tasks on a range of different projects and therefore avoids the need for requirements engineers to familiarise themselves with a large number of frameworks: a time-consuming task. This research is particularly timely at present because there has been an upturn of interest in requirements reuse research in recent years (as indicated by the emergence of reuse-oriented workshops at the Requirements Engineering conference, such as the Requirements Patterns (RePa) workshop, and as illustrated by the emphasis on the topic in key survey papers [NE00, CA07]).

10.1.2 Reusable Design Heuristics and Rationale

The design of RORE is based on five design heuristics. These are presented in Section 3.3.4. Chief among these heuristics is the injunction to support both procedural and declarative reuse as mutually complementary approaches. This thesis has not rigorously evaluated all of these design heuristics individually, although they can be afforded a degree of credibility by virtue of the success of the RORE approach as a whole in realising its design goals. The thesis has, however, provided strong evidence (in Chapter 9) to support the notion that procedural reuse is inherently more general than declarative reuse, and so appears to be an important contributing factor in the properties of the RORE approach.

Although these heuristics are embodied in this thesis by the RORE approach specifically, they also represent a novel contribution in their own right. This is because they

can be treated as guidance or principles to support design-for-reuse generally, and so can be reused by other researchers in the requirements reuse community to develop their own approaches to generalised requirements reuse; approaches which may further improve on the gains which have been made by RORE.

10.1.3 Prototype Tool Support for Generalised Requirements-Level Reuse

Chapter 4 argued that in order to be effective in practice the RORE approach would need to be supported by effective software tools. Maiden has also previously argued that reuse should be supported by intelligent software tools [Mai92]. Chapter 4 therefore proposes a prototype tool to support the RORE approach. This thesis has identified performance and usability limitations which would inhibit the practicality of the approach in the context of real-world requirements engineering. Nonetheless, Chapter 9 argues that the prototype tool offers a respectable degree of utility in addition to the generality benefits which the RORE approach offers. With some further development, therefore, this prototype tool is one candidate for a commercial tool to support reuse-driven requirements engineering.

10.1.4 Extensions to the Domain Theory

Finally, this thesis makes modest, but nonetheless interesting, contributions to the literature on the Domain Theory. The case study presented in Chapter 7 introduces a refined version of the Domain Theory with a view to achieving a greater degree of generality than does the version presented by Sutcliffe and Maiden [SM98]. This refined version introduces both a generalised meta-schema for expressing Object System Models, and a novel schema for expressing Information System Models. This represents a contribution to the literature not only because of the apparent improvement with respect to the generality of the OSM library, but also because the Domain Theory currently lacks a clear notation for expressing ISMs. Furthermore, the case study introduces a novel process for transforming OSMs into ISMs by reuse. This represents a contribution in itself because whereas Papamargaritis' approach to application generation is constrained to a single OSM family, the transformation presented in this thesis is domain general, but is only capable of generating software specifications. The approach thus represents an alternative, more incremental approach to requirements progression.

10.2 Lessons Learnt

Through the research presented in this thesis, and through the evaluation of that research in Chapter 9, the following general lessons have been learnt.

- Addressing the trade-off between generality and utility is a non-trivial problem, and no single solution can provide both extreme generality and extreme utility. In order to provide reuse solutions which improve on those that are available at present, therefore, it is desirable to develop tools which integrate different solutions into a coherent framework (such as a refined instantiation of POSE) so as to take advantage of the particular benefits which each of those solutions provides;

- Providing support for procedural reuse within systematic requirements reuse frameworks - as a complement, not an alternative, to declarative reuse - can significantly increase the generality of the requirements reuse approach, without significantly undermining the utility of that approach.

- The evaluation in Chapter 9 identifies objective criteria for each of the major properties against which RORE was evaluated. These criteria could be used as the basis for an evaluative framework to support the assessment of other requirements reuse approaches;

- The three generic requirements engineering tasks presented in this thesis - Analysis, Inference and Elicitation - and their specialisations have been shown to be sufficient process patterns to guide the production and refinement of requirements models. They provide a template, therefore, on which other requirements-reuse approaches and methods can be based;

- This thesis has introduced, and provided evidence to support, the idea that Clancey's heuristic classification provides an effective pattern for implementing tools to support requirements-level reuse because it concisely captures the three major activities which underpin a reuse-driven approach to requirements model construction and refinement;

- Retrieval mechanisms are a critically important part of the reuse equation. An effective retrieval mechanism can have a significant impact on both generality and utility, so retrieval mechanisms should be carefully evaluated before being integrated into reuse tools;

- This thesis appears to have demonstrated a limitation in the Pellet OWL reasoner which appears to support existing work within the web ontology space that indicates that while Pellet performs well over small ontologies and for certain kinds of query, the SWRL reasoner in Pellet is not yet practical in all scenarios. This thesis has not, however, sought to delineate the precise boundaries of this limitation.

10.3 Future Work

Future work on the RORE approach would aim to:

1. **Publish Research** which disseminate knowledge of the RORE approach and of the general lessons learnt in this thesis;

2. **Conduct Real-World Case Studies** in order to further validate the arguments made in this thesis as to the generality, utility and practicality of the RORE approach;

3. **Conduct Usability Test with Requirements Engineers** to further validate the arguments made in this thesis about the utility of the RORE approach;

4. **Extend the Validation with Additional Model Types** in order to further test the limitations of RORE with respect to the kinds of model which the approach can generate and transform, and to add evidence to further support the conclusion of this thesis that RORE is general across different requirements engineering methodologies. In particular, whereas this thesis has generalised the Domain Theory to support a broader range of requirements engineering tasks and domains, future work would investigate the possibility of applying RORE to formalise the POSE Process Pattern;

5. **Investigate Support for Stakeholder Validation** of requirements models and the inferences over those models which are produced by RORE.

Bibliography

[ABB+04] J.R. Anderson, D. Bothell, M.D. Byrne, S. Douglass, C. Lebiere, and Y. Qin. An integrated theory of the mind. *Psychological review*, 111(4):1036, 2004.

[ABBC99] D.L. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: A domain-specific language for form-based services. *Software Engineering, IEEE Transactions on*, 25(3):334–346, 1999.

[AC98] E. Agerbo and A. Cornils. How to preserve the benefits of design patterns. *ACM SIGPLAN Notices*, 33(10):134–143, 1998.

[Ale77] Christopher Alexander. *A pattern language: towns, buildings, construction*, volume 2. Oxford University Press, USA, 1977.

[And04] J.R. Anderson. *Cognitive psychology and its implications*. Worth Pub, 2004.

[AT05] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734–749, 2005.

[Baa03] Franz Baader. *The description logic handbook: theory, implementation, and applications*. Cambridge: Cambridge University Press, 2003.

[Bas96] Paul G Bassett. *Framing software reuse: lessons from the real world*. Prentice-Hall, Inc., 1996.

[BBGM00] L. Baum, M. Becker, L. Geyer, and G. Molter. Mapping requirements to reusable components using design spaces. In *ICRE*, page 159. Published by the IEEE Computer Society, 2000.

[BBP+05] M. Bittner, A. Botorabi, A. Poth, M.O. Reiser, and M. Weber. Managing variability and reuse of features and requirements for large and complex organizational structures. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 469–470. IEEE, 2005.

[BDJ10] R.K. Bhatia, M. Dave, and RC Joshi. Ant colony based rule generation for reusable software component retrieval. *ACM SIGSOFT Software Engineering Notes*, 35(2):1–5, 2010.

[Bek03] J.D. Bekenstein. Information in the holographic universe. *Scientific American*, 289(2):58–65, 2003.

[Bek04] J.D. Bekenstein. Black holes and information theory. *Contemporary Physics*, 45(1):31–43, 2004.

[BHS07] F. Buschmann, K. Henney, and D.C. Schmidt. *Pattern-oriented software architecture: On patterns and pattern languages*, volume 5. John Wiley & Sons Inc, 2007.

[BJJSS04] L. Bass, B.E. John, N. Juristo, and M.I. Sanchez-Segura. Usability-supporting architectural patterns. In *Proceedings of the 26th International Conference on Software Engineering*, pages 716–717. IEEE Computer Society, 2004.

[BLP04] Stan Bühne, Kim Lauenroth, and Klaus Pohl. Why is it not sufficient to model requirements variability with feature models. In *Proceedings of the Workshop: Automotive Requirements Engineering (AURE04), co-located at RE04, Nagoya, Japan*. Citeseer, 2004.

[BM07] A. Bragança and R.J. Machado. Automating mappings between use case diagrams and feature models for software product lines. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 3–12. IEEE, 2007.

[BMU75] Barry W Boehm, Robert K McClean, and DE Urfrig. Some experience with automated aids to the design of large-scale reliable software. *Software Engineering, IEEE Transactions on*, 1(1):125–133, 1975.

[BNT02] R. Biddle, J. Noble, and E. Tempero. Supporting reusable use cases. *Software Reuse: Methods, Techniques, and Tools*, pages 135–138, 2002.

[Boo05] Grady Booch. Handbook of software architecture. *The Handbook of Software Architecture*, 2005.

[BVHH+04] S. Bechhofer, F. Van Harmelen, J. Hendler, I. Horrocks, D.L. McGuinness, P.F. Patel-Schneider, L.A. Stein, et al. Owl web ontology language reference. *W3C recommendation*, 10:2006–01, 2004.

[CA07] B.H.C. Cheng and J.M. Atlee. Research directions in requirements engineering. In *2007 Future of Software Engineering*, pages 285–303. IEEE Computer Society, 2007.

[CDA93] S. Castano and V. De Antonellis. Reuse of conceptual requirement specifications. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 121–124. IEEE, 1993.

[CDAFP98] S. Castano, V. De Antonellis, M.G. Fugini, and B. Pernici. Conceptual schema analysis: techniques and applications. *ACM Transactions on Database Systems (TODS)*, 23(3):286–333, 1998.

[CDM+11] C. Cappiello, F. Daniel, M. Matera, M. Picozzi, and M. Weiss. Enabling end user development through mashups: Requirements, abstractions and innovation toolkits. *End-User Development*, pages 9–24, 2011.

[Chu91] Lawrence Chung. Representation and utilization of non-functional requirements for information system design. In *Advanced Information Systems Engineering*, pages 5–30. Springer, 1991.

[CJ97] B.H.C. Cheng and J.J. Jeng. Reusing analogous components. *Knowledge and Data Engineering, IEEE Transactions on*, 9(2):341–349, 1997.

[CKM02] Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: the tropos project. *Information systems*, 27(6):365–389, 2002.

[Cla85] W.J. Clancey. Heuristic classification. *Artificial intelligence*, 27(3):289–350, 1985.

[CN01] P. Clements and L. Northrop. *Software product lines*. Addison-Wesley, 2001.

[CNM95] P. Coad, D. North, and M. Mayfield. *Object models: strategies, patterns and applications*. Prentice Hall, Englewood Cliffs, 1995.

[Cor12] Intel Corporation. Intel 64 and ia-32 architectures software developers manual (combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c). `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, Accessed 2012.

[CS06] L. Chung and S. Supakkul. Capturing and reusing functional and non-functional requirements knowledge: A goal-object pattern approach. In *Information Reuse and Integration, 2006 IEEE International Conference on*, pages 539–544. IEEE, 2006.

[CSdPLDW13] H Cunha, JC Sampaio do Prado Leite, L Duboc, and V Werneck. The challenges of representing transparency as patterns. In *Proc. of the 3rd International Workshop on Requirements Patterns (RePa'13)*. IEEE, 2013.

[CWK05] L.M. Cysneiros, V.M. Werneck, and A. Kushniruk. Reusable knowledge for satisficing usability requirements. In *Proceedings of the 13th International Conference on Requirements Engineering*. IEEE Computer Society, 2005.

[Cza05] K. Czarnecki. Overview of generative software development. *Unconventional Programming Paradigms*, pages 326–341, 2005.

[CZW+11] S. Cai, Y. Zou, L. Wang, B. Xie, and W. Shao. A semi-supervised approach for component recommendation based on citations. *Top Productivity through Software Reuse*, pages 78–86, 2011.

[CZZM05] K. Chen, W. Zhang, H. Zhao, and H. Mei. An approach to constructing feature models based on requirements clustering. In *Requirements*

Engineering, 2005. Proceedings. 13th IEEE International Conference on, pages 31–40. IEEE, 2005.

[DFvL91] A. Dardenne, S. Fickas, and A. van Lamsweerde. Goal-directed concept acquisition in requirements elicitation. In *Proceedings of the 6th international workshop on Software specification and design*, pages 14–21. IEEE Computer Society Press, 1991.

[DK11] T. Dao and K. Kang. Mapping features to reusable components: a problem frames-based approach. *Software Product Lines: Going Beyond*, pages 377–392, 2011.

[DPM05] I. Diaz, O. Pastor, and A. Matteo. Modeling interactions using role-driven patterns. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 209–218. IEEE, 2005.

[DVL96] R. Darimont and A. Van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *ACM SIGSOFT Software Engineering Notes*, 21(6):179–190, 1996.

[DVLF93] A. Dardenne, A. Van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1):3–50, 1993.

[EAM06] M. El-Attar and J. Miller. Matching antipatterns to improve the quality of use case models. In *Proceedings of the 14th international conference on Requirements Engineering*. IEEE Computer Society, 2006.

[EMM04] I.A.M. El-Maddah and T.S.E. Maibaum. Requirements-reuse using GOPCSD: Component-based development of process control systems. *Software Reuse: Methods, Techniques and Tools*, pages 318–328, 2004.

[Eri97] SK Eric. Why agent-oriented requirements engineering. In *Proceedings of 3rd Workshop on Requirements Engineering For Software Quality*, 1997.

[Fin88] A. Finkelstein. Re-use of formatted requirements specifications. *Software Engineering Journal*, 3(5):186–197, 1988.

[FK05] W.B. Frakes and K. Kang. Software reuse research: Status and future. *Software Engineering, IEEE Transactions on*, 31(7):529–536, 2005.

[FNP97] R.E. Faith, L.S. Nyland, and J.F. Prins. Khepera: A system for rapid implementation of domain specific languages. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 19–19. USENIX Association, 1997.

[Fow97] M. Fowler. *Analysis patterns: reusable object models*. Addison-Wesley, 1997.

[Gam95] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GB97] L. Goldin and D.M. Berry. Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation. *Automated Software Engineering*, 4(4):375–412, 1997.

[Gen83] D. Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive science*, 7(2):155–170, 1983.

[GHF00] Roger Gomm, Martyn Hammersley, and Peter Foster. *Case study method: Key issues, key texts*. Sage, 2000.

[GHT06] Tom Gardiner, Ian Horrocks, and Dmitry Tsarkov. Automated benchmarking of description logic reasoners. In *Proceedings of the International Workshop on Description Logics (06) CEUR*, volume 189, pages 167–174, 2006.

[Gio03] P. Giorgini. Organizational patterns for early requirements analysis. In *15TH INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING (CAISE'03)*, 2003.

[Gla98] Robert L Glass. Software runaways: lessons learned from massive software project failures. *Prentice-Hall Co. and Yourdon Press. Communications of the ACM*, 41(7):14–17, 1998.

[GLC+01] F. Gobet, P.C.R. Lane, S. Croker, P.C.H. Cheng, G. Jones, I. Oliver, and J.M. Pine. Chunking mechanisms in human learning. *Trends in cognitive sciences*, 5(6):236–243, 2001.

[GM97] D. Gentner and A.B. Markman. Structure mapping in analogy and similarity. *American psychologist*, 52(1):45, 1997.

[Gom95] H. Gomaa. Domain modeling methods and environments. *ACM SIGSOFT Software Engineering Notes*, 20(SI):256–258, 1995.

[Gri94] Martin L Griss. Software reuse experience at hewlett-packard. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, page 270. IEEE, 1994.

[Gro09] Object Management Group. Omg unified modeling language (omg uml), superstructure: Version 2.2. `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/`, Accessed 2009.

[Gro11] Object Management Group. Omg meta object facility (mof) core specification. `http://www.omg.org/spec/MOF/2.4.1/PDF/`, 2011.

[Gru08] L. Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th international conference on Software engineering*, pages 31–40. ACM, 2008.

[GT04] J.Y. Gil and Y. Tsoglin. Jamoos-a domain-specific language for language processing. *Journal of Computing and Information Technology*, 9(4):305–321, 2004.

[Gui90] R. Guindon. Designing the design process: Exploiting opportunistic thoughts. *Human-Computer Interaction*, 5(2):305–344, 1990.

[GW06] Tony Gorschek and Claes Wohlin. Requirements abstraction model. *Requirements Engineering*, 11(1):79–101, 2006.

[GY01] D. Gross and E. Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, 2001.

[HA06] O. Hummel and C. Atkinson. Using the web as a reuse repository. *Reuse of Off-the-Shelf Components*, pages 298–311, 2006.

[Hap13] Solve Me Happy. Poe wiki. `http://problemoriented.wikispaces.com/1.+Basic+building+blocks`, Accessed 2013.

[HB11] M. Horridge and S. Bechhofer. The owl api: A java api for owl ontologies. *Semantic Web*, 2(1):11–21, 2011.

[HC11] Sargon Hasso and RC Carlson. Software patterns catalog based on a new classification scheme. *Software: Practice and Experience*, 2011.

[HH03] N. Heumesser and F. Houdek. Towards systematic recycling of systems requirements. In *Proceedings of the 25th International Conference on Software Engineering*, pages 512–519. IEEE Computer Society, 2003.

[Hin08] Chris Hinds. The case against a positivist philosophy of requirements engineering. *Requirements engineering*, 13(4):315–328, 2008.

[HJA08] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE software*, pages 45–52, 2008.

[HJL+02] J.G. Hall, M. Jackson, R.C. Laney, B. Nuseibeh, and L. Rapanotti. Relating software requirements and architectures using problem frames. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 137–144. IEEE, 2002.

[HJN11] Klaus Marius Hansen, Kristjan Jonasson, and Helmut Neukirchen. An empirical study of software architectures effect on product quality. *Journal of Systems and Software*, 84(7):1233–1243, 2011.

[HK97] F. Houdek and H. Kempter. Quality patterns: an approach to packaging software engineering experience. *ACM SIGSOFT Software Engineering Notes*, 22(3):81–88, 1997.

[HL04] L. Hagge and K. Lappe. Patterns for the RE process. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 90–99. IEEE, 2004.

[HL05] L. Hagge and K. Lappe. Sharing requirements engineering experience using patterns. *Software, IEEE*, 22(1):24–31, 2005.

[HM04] K. Hammond and G. Michaelson. The design of Hume: a high-level language for the real-time embedded systems domain. *Domain-Specific Program Generation*, pages 5–35, 2004.

[Hol06] R. Holmes. Unanticipated reuse of large-scale software features. In *Proceedings of the 28th international conference on Software engineering*, pages 961–964. ACM, 2006.

[HR08] J.G. Hall and L. Rapanotti. Poelog: a prologbased engine for problem oriented engineering. Technical report, Technical Report TR2008/07, The Open University, 2008.

[HR09] J.G. Hall and L. Rapanotti. Assurance-driven design in problem oriented engineering. *International Journal on Advances in Systems and Measurements*, 2(1):119–130, 2009.

[HRJ05] J.G. Hall, L. Rapanotti, and M. Jackson. Problem frame semantics for software development. *Software and Systems Modeling*, 4(2):189–198, 2005.

[HRJ07] J.G. Hall, L. Rapanotti, and M. Jackson. Problem oriented software engineering: A design-theoretic framework for software engineering. In *Fifth IEEE International Conference on Software Engineering and Formal Methods, 2007*. IEEE Computer Society, 2007.

[HRJ08] J.G. Hall, L. Rapanotti, and M.A. Jackson. Problem oriented software engineering: Solving the package router control problem. *IEEE Transactions on Software Engineering*, pages 226–241, 2008.

[Jac92] Ivar Jacobson. *Object-oriented software engineering: a use case driven approach*. Pearson Education India, 1992.

[Jac01a] M. Jackson. Problem analysis and structure. *Nato Science Series Sub Series III Computer and Systems Sciences*, 180:3–20, 2001.

[Jac01b] M. Jackson. *Problem frames: analysing and structuring software development problems*. Addison-Wesley, 2001.

[Jac05] M. Jackson. Problem frames and software engineering. *Information and Software Technology*, 47(14):903–912, 2005.

[JBR+92] M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, and Y. Vassilou. Theories underlying requirements engineering: An overview of nature at genesis. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 19–31. IEEE, 1992.

[JL06] Z. Jin and L. Liu. Towards automatic problem decomposition: an ontology-based approach. In *Proceedings of the 2006 international workshop on Advances and applications of problem frames*, pages 41–48. ACM, 2006.

[Joo94] Rebecca Joos. Software reuse at Motorola. *Software, IEEE*, 11(5):42–47, 1994.

[Jur95] I. Jurisica. A similarity-based retrieval tool for software repositories. In *In The Third Workshop on AI and Software Engineering: Breaking the Mold. IJCAI-95*. Citeseer, 1995.

[Kan90] K.C. Kang. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.

[KC02] S. Konrad and B.H.C. Cheng. Requirements patterns for embedded systems. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 127–136. IEEE, 2002.

[KC05a] S. Konrad and B.H.C. Cheng. Facilitating the construction of specification pattern-based properties. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 329–338. IEEE, 2005.

[KC05b] S. Konrad and B.H.C. Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381. ACM, 2005.

[KF07] M. Kolp and S. Faulkner. Patterns for organizational modeling. *International Journal of Enterprise Information Systems*, 3(3):1–22, 2007.

[KGM03] M. Kolp, P. Giorgini, and J. Mylopoulos. Organizational patterns for early requirements analysis. In *Advanced Information Systems Engineering*, page 1030. Springer, 2003.

[KHG11] M. Kamalrudin, J. Hosking, and J. Grundy. Improving requirements quality using essential use case interaction patterns. In *Proceeding of the 33rd international conference on Software engineering*, pages 531–540. ACM, 2011.

[KKL+98] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.

[KLD02] K.C. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *Software, IEEE*, 19(4):58–65, 2002.

[Kon96] Jyrki Kontio. A case study in applying a systematic method for cots selection. In *Software Engineering, 1996., Proceedings of the 18th International Conference on*, pages 201–209. IEEE, 1996.

[Kru92] C.W. Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[Kru02a] C. Krueger. Eliminating the adoption barrier. *IEEE Software*, 19(4):29–31, 2002.

[Kru02b] C.W. Krueger. Easing the transition to software mass customization. *Software Product-Family Engineering*, pages 178–184, 2002.

[KSN10] H. Kaindl, M. Smiałek, and W. Nowakowski. Case-based reuse with partial requirements specifications. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 399–400. IEEE, 2010.

[KWK11] S. Kiv, Y. Wautelet, and M. Kolp. A multi-agent architectural pattern for wrapping off-the-shelf components. *Agent and Multi-Agent Systems: Technologies and Applications*, pages 321–331, 2011.

[LAF12] Daniel Lucredio, Eduardo Santana de Almeida, and Renata PM Fortes. An investigation on the impact of MDE on software reuse. In *Software Components Architectures and Reuse (SBCARS), 2012 Sixth Brazilian Symposium on*, pages 101–110. IEEE, 2012.

[Lan61] R. Landauer. Irreversibility and heat generation in the computing process. *IBM journal of research and development*, 5(3):183–191, 1961.

[Lap05] Alexei Lapouchnian. Goal-oriented requirements engineering: An overview of the current research. Technical report, University of Toronto, 2005.

[Lar04] C. Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. Prentice Hall PTR, 2004.

[LBN+09] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.

[LCPCdB10] L. Lavazza, A. Coen-Porisini, P. Colombo, and V. del Bianco. A meta-model supporting the decomposition of problem descriptions. In *2010 Fifth International Conference on Software Engineering Advances*, pages 50–57. IEEE, 2010.

[LDB06] L. Lavazza and V. Del Bianco. Combining problem frames and UML in the description of software requirements. *Fundamental Approaches to Software Engineering*, pages 199–213, 2006.

[LG06] A. Lindoso and R. Girardi. The SRAMO technique for analysis and reuse of requirements in multi-agent application engineering. In *IX workshop on Requirements Engineering, Cadernos do IME, UERJ Press*, volume 20, pages 41–50, 2006.

[LJ06] L. Liu and Z. Jin. Integrating goals and problem frames in requirements analysis. In *Requirements Engineering, 14th IEEE International Conference*, pages 349–350. IEEE, 2006.

[LL98] Andy Lynex and Paul J Layzell. Organisational considerations for software reuse. *Annals of Software Engineering*, 5(1):105–124, 1998.

[LLG02] Oscar López, Miguel A Laguna, and Francisco J García. Metamodeling for requirements reuse. *Anais do WER*, 2002.

[LM03] D. Liu and H. Mei. Mapping requirements to software architecture by feature-orientation. In *STRAW03 Second International SofTware Requirements to Architectures Workshop*, page 69. Citeseer, 2003.

[LMV97] W. Lam, J.A. McDermid, and AJ Vickers. Ten steps towards systematic requirements reuse. *Requirements Engineering*, 2(2):102–113, 1997.

[LNI+03] L. Lin, B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett. Introducing abuse frames for analysing security requirements. In *Proceedings. 11th IEEE International on Requirements Engineering Conference, 2003*. IEEE Computer Society, 2003.

[LS08] Harris T Lin and Evren Sirin. Pellinta performance lint tool for pellet. In *Proc. of the OWL Experiences and Directions Workshop at the ISWC*, volume 8, 2008.

[LW05] K.K. Lau and Z. Wang. A taxonomy of software component models. In *Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on*, pages 88–95. IEEE, 2005.

[Mai92] N. Maiden. *Analogical specification reuse during requirements analysis*. PhD thesis, London: Department of Business Computing, City University, School of Informatics, 1992.

[Man10] D. Mannering. A POSE process with Alloy and Perfect Developer. In *Proceedings of the 1st international workshop on Applications and advances of problem orientation*, 2010.

[MBFQ08] O. Mendez-Bonilla, X. Franch, and C. Quer. Requirements patterns for COTS systems. In *Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on*, pages 232–234. IEEE, 2008.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems (TOIS)*, 8(4):325–362, 1990.

[MBLN06] Ayman Mahfouz, Leonor Barroca, Robin Laney, and Bashar Nuseibeh. Patterns for service-oriented information exchange requirements. In *Proceedings of the 2006 conference on Pattern languages of programs*, page 22. ACM, 2006.

[MCK05] F. Mccarey, M.Ó. Cinnéide, and N. Kushmerick. Rascal: A recommender agent for agile reuse. *Artificial Intelligence Review*, 24(3):253–276, 2005.

[MCN92] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *Software Engineering, IEEE Transactions on*, 18(6):483–497, 1992.

[MCY99] John Mylopoulos, Lawrence Chung, and Eric Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, 1999.

[MDS03] Tim Menzies and Justin S Di Stefano. More success and failure factors in software reuse. *Software Engineering, IEEE Transactions on*, 29(5):474–477, 2003.

[MET02] M. Morisio, M. Ezran, and C. Tully. Success and failure factors in software reuse. *IEEE Transactions on software engineering*, pages 340–357, 2002.

[Mey88] B. Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.

[MHR08] D. Mannering, J.G. Hall, and L. Rapanotti. Safety process improvement with POSE and Alloy. *Improvements in System Safety*, pages 25–41, 2008.

[MHS05] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[MI83] James Martin and Savant Institute. *Managing the data-base environment*. Prentice-Hall Englewood Cliffs (NJ), 1983.

[MMJG05] N.A.M. Maiden, S. Manning, S. Jones, and J. Greenwood. Generating requirements from systems models using patterns: a case study. *Requirements Engineering*, 10(4):276–288, 2005.

[MMS95] NAM Maiden, P. Mistry, and AG Sutcliffe. How people categorise requirements for reuse: a natural approach. *RE*, page 148, 1995.

[MR92] Johan Margono and Thomas E Rhoads. Software reuse economics: cost-benefit analysis on a large-scale ada project. In *Proceedings of the 14th international conference on Software engineering*, pages 338–348. ACM, 1992.

[MS94] NAM Maiden and AG Sutcliffe. Requirements critiquing using domain abstractions. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 184–193. IEEE, 1994.

[MS96] NAM Maiden and AG Sutcliffe. A computational mechanism for parallel problem decomposition during requirements engineering. In *Proceedings of the 8th International Workshop on Software Specification and Design*, page 159. IEEE Computer Society, 1996.

[MVL97] P. Massonet and A. Van Lamsweerde. Analogical reuse of requirements frameworks. *RE*, page 26, 1997.

[NE00] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46. ACM, 2000.

[Nei84] J.M. Neighbors. The Draco approach to constructing software from reusable components. *Software Engineering, IEEE Transactions on*, 10(5):564–574, 1984.

[NF00] Markus Nick and Raimund Feldmann. Guidelines for evaluation and improvement of reuse and experience repository systems through measurement programs. In *Proc. 3rd European Software Measurement Conference (FESMA-AEMES 2000), Madrid, Spain*, 2000.

[NoCBDoIS80] J.M. Neighbors, University of California (Berkeley). Department of Information, and Computer Science. *Software construction using components*. Citeseer, 1980.

[NZ11] J. Naish and L. Zhao. Towards a generalised framework for classifying and retrieving requirements patterns. In *Requirements Patterns (RePa), 2011 First International Workshop on*, pages 42–51. IEEE, 2011.

[Pan05] Zhengxiang Pan. Benchmarking DL reasoners using realistic ontologies. In *Proc. of the First OWL Experiences and Directions Workshop*, 2005.

[Pap06] G. Papamargaritis. *Extending the domain theory to support application generation*. PhD thesis, University of Manchester, UK, 2006.

[PD87] R. Prieto-Diaz. Domain analysis for reusability. In *Proceedings of COMPSAC*, volume 87, pages 23–29, 1987.

[PD90] R. Prieto-Díaz. Domain analysis: an introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.

[PD91] R. Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, 1991.

[Pen11] B. Penzenstadler. Exactly the information your subcontractor needs: Desyre?decomposing system requirements. In *Requirements Patterns (RePa), 2011 First International Workshop on*, pages 1–10. IEEE, 2011.

[PLC+12] J. Pimentel, M. Lucena, J. Castro, C. Silva, E. Santos, and F. Alencar. Deriving software architectural models from requirements models for adaptive systems: the stream-a approach. *Requirements Engineering*, 17(4):1–23, 2012.

[PMCS11] B. Paech, N. Maiden, L. Chung, and S. Supakkul. 2011 first international workshop on requirements patterns. In *Requirements Patterns (RePa), 2011 First International Workshop on*, pages ii–iii. IEEE, 2011.

[Pou95] J.S. Poulin. Populating software repositories: Incentives and domain-specific software. *Journal of Systems and Software*, 30(3):187–199, 1995.

[PS04] G. Papamargaritis and A. Sutcliffe. Applying the domain theory to design for reuse. *BT technology journal*, 22(2):104–115, 2004.

[PSP09] B. Penzenstadler, E. Sikora, and K. Pohl. A requirements reference model for model-based requirements engineering in the automotive domain. *Requirements Engineering: Foundation for Software Quality*, pages 212–217, 2009.

[PTRC07] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007.

[Red12] Timothy Redmond. Protege-OWL code generator. `http://protegewiki.stanford.edu/wiki/Protege-OWL_Code_Generator`, Accessed 2012.

[RHJN04] L. Rapanotti, J.G. Hall, M. Jackson, and B. Nuseibeh. Architecture-driven problem decomposition. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 80–89. IEEE, 2004.

[Ris10] L. Rising. The benefit of patterns. *Software, IEEE*, 27(5):15–17, 2010.

[RM93] K. Ryan and B. Mathews. Matching conceptual graphs as an aid to requirements re-use. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 112–120. IEEE, 1993.

[RMBFQ09a] S. Renault, O. Mendez-Bonilla, X. Franch, and C. Quer. Pabre: Pattern-based requirements elicitation. In *Research Challenges in Information Science, 2009. RCIS 2009. Third International Conference on*, pages 81–92. IEEE, 2009.

[RMBFQ09b] S. Renault, O. Mendez-Bonilla, X. Franch, and C. Quer. A pattern-based method for building requirements documents in call-for-tender processes. *International Journal of Computer Science & Applications*, 6, 2009.

[RMG+76] E. Rosch, C.B. Mervis, W.D. Gray, D.M. Johnson, and P. Boyes-Braem. Basic objects in natural categories. *Cognitive psychology*, 8(3):382–439, 1976.

[RPR98] C. Rolland, V. Plihon, and J. Ralyté. Specifying the reuse context of scenario method chunks. In *Advanced Information Systems Engineering*, pages 191–218. Springer, 1998.

[RRJ03] MS Rajasree, P.J.K. Reddy, and D. Janakiram. Pattern oriented software development: Moving seamlessly from requirements to architecture. In *STRAW?03 Second International SofTware Requirements to Architectures Workshop*, page 54. Citeseer, 2003.

[RW91] H.B. Reubenstein and R.C. Waters. The requirements apprentice: Automated assistance for requirements acquisition. *Software Engineering, IEEE Transactions on*, 17(3):226–240, 1991.

[SABP07] E. Sadraei, A. Aurum, G. Beydoun, and B. Paech. A field study of the requirements engineering practice in australian software industry. *Requirements Engineering*, 12(3):145–162, 2007.

[Sae00] M. Saeki. Patterns and aspects for use cases: reuse techniques for use case descriptions. In *Requirements Engineering, 2000. Proceedings. 4th International Conference on*. IEEE, 2000.

[SC99] A.G. Sutcliffe and J.M. Carroll. Designing claims for reuse in interactive systems design. *International Journal of Human Computer Studies*, 50(3):213–242, 1999.

[Sch82] R.C. Schank. *Dynamic memory: A theory of reminding and learning in computers and people*, volume 240. Cambridge University Press Cambridge, 1982.

[Sch99] Douglas C Schmidt. Why software reuse has failed and how to make it work for you. *C++ Report*, 11(1), 1999.

[SF10] A. Shahrokni and R. Feldt. Towards a framework for specifying software robustness requirements based on patterns. *Requirements Engineering: Foundation for Software Quality*, pages 79–84, 2010.

[SFO03] G. Sindre, D.G. Firesmith, and A.L. Opdahl. A reuse-based approach to determining security requirements. In *Proc. 9th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'03)*, pages 127–136, 2003.

[SGL08] Farida Semmak, Christophe Gnaho, and Regine Laleau. Extended kaos to support variability for goal oriented requirements reuse. *Proceedings of MoDISE-EUS*, page 23, 2008.

[Sha01] C.E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[SHC+10] S. Supakkul, T. Hill, L. Chung, T.T. Tun, and JC Sampaio do Prado Leite. An NFR pattern approach to dealing with NFRs. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 179–188. IEEE, 2010.

[SHOC09a] S. Supakkul, T. Hill, E. Oladimeji, and L. Chung. Security threat and vulnerability mitigation patterns: A case of credit card theft mitigation. In *Proc. of the 16th Patterns Languages of Programs (PLoP 2009)*. IEEE, 2009.

[SHOC09b] Sam Supakkul, Tom Hill, Ebenezer Akin Oladimeji, and Lawrence Chung. Capturing, organizing, and reusing knowledge of NFRs: An NFR pattern approach. In *Managing Requirements Knowledge (MARK), 2009 Second International Workshop on*, pages 75–84. IEEE, 2009.

[Sim74] H.A. Simon. The structure of ill structured problems. *Artificial intelligence*, 4(3):181–201, 1974.

[SJ06a] R. Seater and D. Jackson. Problem frame transformations: deriving specifications from requirements. In *Proceedings of the 2006 international workshop on Advances and applications of problem frames*, pages 71–80. ACM, 2006.

[SJ06b] R. Seater and D. Jackson. Requirement progression in problem frames applied to a proton therapy system. In *14th IEEE International Conference Requirements Engineering*. IEEE Computer Society, 2006.

[SJG07] R. Seater, D. Jackson, and R. Gheyi. Requirement progression in problem frames: deriving specifications from requirements. *Requirements Engineering*, 12(2):77–102, 2007.

[ŚKK+10] M. Śmiałek, A. Kalnins, E. Kalnina, A. Ambroziewicz, T. Straszak, and K. Wolter. Comprehensive system for systematic case-driven software reuse. *SOFSEM 2010: Theory and Practice of Computer Science*, pages 697–708, 2010.

[SL11] M. Serrano and J.C.S.P. Leite. Capturing transparency-related requirements patterns through argumentation. In *Requirements Patterns*

(RePa), 2011 First International Workshop on, pages 32–41. IEEE, 2011.

[SLM+07] E. Souza, M. Lencastre, R. Melo, L. Ramires, and K. Alves. Analyzing problem frames together with solution patterns. In *Proceedings of Workshop em Engenharia de Requisitos*, 2007.

[SM98] A. Sutcliffe and N. Maiden. The domain theory for requirements engineering. *Software Engineering, IEEE Transactions on*, 24(3):174–196, 1998.

[SP07] E. Sirin and B. Parsia. Sparql-dl: Sparql query for owl-dl. In *Proceedings of OWLED 2007*. OWLED, 2007.

[SPG+07] E. Sirin, B. Parsia, B.C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[SPZ06] A. Sutcliffe, G. Papamargaritis, and L. Zhao. Comparing requirements analysis methods for developing reusable component libraries. *Journal of Systems and Software*, 79(2):273–289, 2006.

[Sup13] Sam Supakkul. NFRs modeling patterns and anti-patterns. `http://www.utdallas.edu/~supakkul/NFR-modeling/patterns/dev-knowledge/NFRs%20goal%20patterns.html`, Accessed 2013.

[Sut00] A. Sutcliffe. Domain analysis for software reuse. *Journal of Systems and Software*, 50(3):175–199, 2000.

[Sut02] A. Sutcliffe. *The domain theory: Patterns for knowledge and software reuse*. L. Erlbaum Associates, 2002.

[SW62] C.E. Shannon and W. Weaver. The mathematical theory of communication. 19(9), 1962.

[SW90] Mildred LG Shaw and J Brian Woodward. Modeling expert knowledge. *Knowledge Acquisition*, 2(3):179–206, 1990.

[SW11] L. Scheinholtz and I. Wilmont. Interview patterns for requirements elicitation. *Requirements Engineering: Foundation for Software Quality*, pages 72–77, 2011.

[Sys12] Derivo Semantic Systems. Sparql-dl api. `http://www.derivo.de/en/resources/sparql-dl-api.html`, 2012.

[TJL+09] T.T. Tun, M. Jackson, R. Laney, B. Nuseibeh, and Y. Yu. Are your lights off? Using Problem Frames to diagnose system failures. In *2009 17th IEEE International Requirements Engineering Conference*, pages 343–348. IEEE, 2009.

[TMC97] S. Thibault, R. Marlet, and C. Consel. A domain specific language for video device drivers: from design to implementation. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 2–2. USENIX Association, 1997.

[TNMG02] A. Toval, J. Nicolás, B. Moros, and F. García. Requirements reuse for improving information systems security: a practitioner's approach. *Requirements Engineering*, 6(4):205–219, 2002.

[TSU+10] S. Toyabe, T. Sagawa, M. Ueda, E. Muneyuki, and M. Sano. Experimental demonstration of information-to-energy conversion and validation of the generalized jarzynski equality. *Nature Physics*, 6(12):988–992, 2010.

[Uni12] Stanford University. welcome to protg. `http://protege.stanford.edu/`, 2012.

[vdBvdS11] J. van den Bos and T. van der Storm. Bringing domain-specific languages to digital forensics. In *Proceedings of ICSE*, volume 11, pages 671–680, 2011.

[VDKV00] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[vKPKH02] A. von Knethen, B. Paech, F. Kiedaisch, and F. Houdek. Systematic

requirements recycling through abstraction and traceability. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 273–281. IEEE, 2002.

[VL04] Axel Van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. In *Proceedings of the 26th International Conference on Software Engineering*, pages 148–157. IEEE Computer Society, 2004.

[WDC06] CHU Wang, Q. Depei, and LIU Chuda. Architecture-based problem frames constructing for software reuse. In *Proceedings of the 2006 international workshop on Advances and applications of problem frames*, pages 19–24. ACM, 2006.

[Wei11] M. Weiss. Pay to play or requirements prioritization in collectives. In *Requirements Patterns (RePa), 2011 First International Workshop on*, pages 28–31. IEEE, 2011.

[WGvE04] R. Wieringa, J. Gordijn, and P. van Eck. Value framing: A prelude to software problem framing. In *1st International Workshop on Advances and Applications of Problem Frames (IWAAPF)*. Citeseer, 2004.

[Wie05] R. Wieringa. Requirements researchers: are we really doing research? *Requirements Engineering*, 10(4):304–306, 2005.

[Wie12] Roel Wieringa. Designing technical action research and generalizing from real-world cases. In *Advanced Information Systems Engineering*, pages 697–698. Springer, 2012.

[WJ09] B. Waldmann and P. Jones. Feature-oriented requirements satisfy needs for reuse and systems view. In *2009 17th IEEE International Requirements Engineering Conference*, pages 329–334. IEEE, 2009.

[WM08] M. Weiss and H. Mouratidis. Selecting security patterns that fulfill security requirements. In *16th IEEE International Requirements Engineering Conference*, pages 169–172. IEEE, 2008.

[WZL11] Y. Wen, H. Zhao, and L. Liu. Analysing security requirements patterns based on problems decomposition and composition. In *Requirements Patterns (RePa), 2011 First International Workshop on*, pages 11–20. IEEE, 2011.

[WZZ+09] B. Wang, W. Zhang, H. Zhao, Z. Jin, and H. Mei. A use case based approach to feature models' construction. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 121–130. IEEE, 2009.

[XZAR06] L. Xu, H. Ziv, T.A. Alspaugh, and D.J. Richardson. An architectural pattern for non-functional dependability requirements. *Journal of Systems and Software*, 79(10):1370–1378, 2006.

[YC] E. Yourdon and L. Constantine. Structured design. 1979.

[Ye02] Y. Ye. An empirical user study of an active reuse repository system. *Software Reuse: Methods, Techniques, and Tools*, pages 429–436, 2002.

[YF00] Y. Ye and G. Fischer. Promoting reuse with active reuse repository systems. *Software Reuse: Advances in Software Reusability*, pages 9–46, 2000.

[Yu93] Eric SK Yu. Modeling organizations for information systems requirements engineering. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 34–41. IEEE, 1993.

[Yu97] E.S.K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. *RE*, page 226, 1997.

[Yu11] L.L.E. Yu. From requirements to architectural design–using goals and scenarios. In *First International Workshop From Software Requirements to Architectures-STRAW'01*, page 22, 2011.

[Zha11] L. Zhao. A pattern language for evolving scenarios into multi-perspective models for requirements specification. In *Requirements Patterns (RePa), 2011 First International Workshop on*, pages 21–27. IEEE, 2011.

Appendix A

Immutable Layer: Formalisation

```
<?xml version="1.0"?>

<!DOCTYPE Ontology [
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<Ontology xmlns="http://www.w3.org/2002/07/owl#"
      xml:base="http://jamesnaish.wordpress.com/ROREKnowledgeModel.
          owl"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:xml="http://www.w3.org/XML/1998/namespace"
      ontologyIRI="http://jamesnaish.wordpress.com/ROREKnowledgeModel
          .owl">
    <Prefix name="" IRI="http://www.w3.org/2002/07/owl#"/>
    <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
    <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-
        ns#"/>
    <Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#"/>
    <Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#"/
        >
    <Declaration>
        <Class IRI="#Activity"/>
    </Declaration>
    <Declaration>
```

```
    <Class IRI="#AdaptationScript"/>
</Declaration>
<Declaration>
    <Class IRI="#AnalysisRule"/>
</Declaration>
<Declaration>
    <Class IRI="#BehaviouralUnit"/>
</Declaration>
<Declaration>
    <Class IRI="#BooleanFact"/>
</Declaration>
<Declaration>
    <Class IRI="#BooleanType"/>
</Declaration>
<Declaration>
    <Class IRI="#ChunkBasedElicitationStimulus"/>
</Declaration>
<Declaration>
    <Class IRI="#ChunkCondition"/>
</Declaration>
<Declaration>
    <Class IRI="#ChunkFactTypeAggregation"/>
</Declaration>
<Declaration>
    <Class IRI="#ComplexFact"/>
</Declaration>
<Declaration>
    <Class IRI="#ComplexType"/>
</Declaration>
<Declaration>
    <Class IRI="#ElicitationStimulus"/>
</Declaration>
<Declaration>
    <Class IRI="#Fact"/>
</Declaration>
<Declaration>
    <Class IRI="#FactAggregation"/>
</Declaration>
<Declaration>
    <Class IRI="#FactEditingElicitationStimulus"/>
</Declaration>
<Declaration>
```

```
    <Class IRI="#FactType"/>
</Declaration>
<Declaration>
    <Class IRI="#FactTypeAggregation"/>
</Declaration>
<Declaration>
    <Class IRI="#IndexDescription"/>
</Declaration>
<Declaration>
    <Class IRI="#InformationRequirement"/>
</Declaration>
<Declaration>
    <Class IRI="#MatchingCondition"/>
</Declaration>
<Declaration>
    <Class IRI="#Model"/>
</Declaration>
<Declaration>
    <Class IRI="#ModelChunk"/>
</Declaration>
<Declaration>
    <Class IRI="#ModelChunkType"/>
</Declaration>
<Declaration>
    <Class IRI="#ModelType"/>
</Declaration>
<Declaration>
    <Class IRI="#ModellingContext"/>
</Declaration>
<Declaration>
    <Class IRI="#MultipleChoiceElicitationStimulus"/>
</Declaration>
<Declaration>
    <Class IRI="#NumericFact"/>
</Declaration>
<Declaration>
    <Class IRI="#NumericType"/>
</Declaration>
<Declaration>
    <Class IRI="#Phase"/>
</Declaration>
<Declaration>
```

```
    <Class IRI="#ProductionRule"/>
</Declaration>
<Declaration>
    <Class IRI="#ProductionScript"/>
</Declaration>
<Declaration>
    <Class IRI="#Property"/>
</Declaration>
<Declaration>
    <Class IRI="#PropertyType"/>
</Declaration>
<Declaration>
    <Class IRI="#RORERule"/>
</Declaration>
<Declaration>
    <Class IRI="#ReusableStructure"/>
</Declaration>
<Declaration>
    <Class IRI="#RuleCondition"/>
</Declaration>
<Declaration>
    <Class IRI="#SimpleFact"/>
</Declaration>
<Declaration>
    <Class IRI="#SimpleType"/>
</Declaration>
<Declaration>
    <Class IRI="#StringFact"/>
</Declaration>
<Declaration>
    <Class IRI="#StringType"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#adaptedBy"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#aggregatesChunkFacts"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#aggregatesFactTypes"/>
</Declaration>
<Declaration>
```

```
    <ObjectProperty IRI="#aggregatesFacts"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#belongsToPhase"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#enforcesGoalFactType"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasActivityConstraint"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasChunkFactTypeAggregation"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasConditionChunk"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasConditionRule"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasCurrentActivity"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasCurrentInformationRequirement"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasCurrentPhase"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasCurrentSourceModel"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasCurrentTargetModel"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasFactAggregation"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasFactTypeAggregation"/>
</Declaration>
<Declaration>
```

```
    <ObjectProperty IRI="#hasFirstAdaptationRule"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasFirstProductionRule"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasNextActivity"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasNextPhase"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasNextProductionRule"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasPhaseConstraint"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasPostconditionConstraint"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasPreconditionConstraint"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasProperty"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasPropertyOfType"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasPropertyValue"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasPropertyValueType"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasResponseOption"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasResponseStructure"/>
</Declaration>
<Declaration>
```

```
    <ObjectProperty IRI="#hasSourceFactTypeConstraint"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasTargetFactTypeConstraint"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasTriggerConditionConstraint"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#indexedBy"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#instantiatesChunkType"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#instantiatesFactType"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#instantiatesFactTypeAggregation"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#instantiatesModelType"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#instantiatesPropertyType"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isSubsetOfModelType"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#producesModelOfType"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#requiresGoalFactType"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#subTypeOf"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#checksContextElement"/>
</Declaration>
<Declaration>
```

```
    <DataProperty IRI="#hasActivityName"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasAggregationName"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasAntecedent"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasBooleanValue"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasCardinality"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasConsequent"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasCycleGoal"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasCyclePostcondition"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasElicitationStimulus"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasFactQuery"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasFactTypeName"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasInputFactQuery"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasModelTypeName"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasNumericValue"/>
</Declaration>
<Declaration>
```

```
    <DataProperty IRI="#hasPhaseName"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasPropertyName"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasResponseOptionsQuery"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasSimpleValue"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#hasStringValue"/>
</Declaration>
<Declaration>
    <Datatype IRI="#ContextElement"/>
</Declaration>
<SubClassOf>
    <Class IRI="#Activity"/>
    <Class IRI="#BehaviouralUnit"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#AnalysisRule"/>
    <Class IRI="#RORERule"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#AnalysisRule"/>
    <Class IRI="#ReusableStructure"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#BooleanFact"/>
    <Class IRI="#SimpleFact"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#BooleanType"/>
    <Class IRI="#SimpleType"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ChunkBasedElicitationStimulus"/>
    <Class IRI="#ElicitationStimulus"/>
</SubClassOf>
<SubClassOf>
```

```
    <Class IRI="#ChunkCondition"/>
    <Class IRI="#MatchingCondition"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ChunkFactTypeAggregation"/>
    <Class IRI="#FactTypeAggregation"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ComplexFact"/>
    <Class IRI="#Fact"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ComplexType"/>
    <Class IRI="#FactType"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ElicitationStimulus"/>
    <Class IRI="#ReusableStructure"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#FactAggregation"/>
    <Class abbreviatedIRI="owl:Thing"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#FactEditingElicitationStimulus"/>
    <Class IRI="#ElicitationStimulus"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ModelChunk"/>
    <Class IRI="#ReusableStructure"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#MultipleChoiceElicitationStimulus"/>
    <Class IRI="#ElicitationStimulus"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#NumericFact"/>
    <Class IRI="#SimpleFact"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#NumericType"/>
    <Class IRI="#SimpleType"/>
```

```
</SubClassOf>
<SubClassOf>
    <Class IRI="#Phase"/>
    <Class IRI="#BehaviouralUnit"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ProductionRule"/>
    <Class IRI="#RORERule"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ProductionScript"/>
    <Class IRI="#ReusableStructure"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#PropertyType"/>
    <Class abbreviatedIRI="owl:Thing"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#RuleCondition"/>
    <Class IRI="#MatchingCondition"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#SimpleFact"/>
    <Class IRI="#Fact"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#SimpleType"/>
    <Class IRI="#FactType"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#StringFact"/>
    <Class IRI="#SimpleFact"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#StringType"/>
    <Class IRI="#SimpleType"/>
</SubClassOf>
<SubObjectPropertyOf>
    <ObjectProperty IRI="#aggregatesFactTypes"/>
    <ObjectProperty abbreviatedIRI="owl:topObjectProperty"/>
</SubObjectPropertyOf>
<SubObjectPropertyOf>
```

```
    <ObjectProperty IRI="#aggregatesFacts"/>
    <ObjectProperty abbreviatedIRI="owl:topObjectProperty"/>
</SubObjectPropertyOf>
<SubObjectPropertyOf>
    <ObjectProperty IRI="#hasCurrentPhase"/>
    <ObjectProperty abbreviatedIRI="owl:topObjectProperty"/>
</SubObjectPropertyOf>
<SubObjectPropertyOf>
    <ObjectProperty IRI="#hasFactAggregation"/>
    <ObjectProperty abbreviatedIRI="owl:topObjectProperty"/>
</SubObjectPropertyOf>
<SubObjectPropertyOf>
    <ObjectProperty IRI="#hasFirstAdaptationRule"/>
    <ObjectProperty abbreviatedIRI="owl:topObjectProperty"/>
</SubObjectPropertyOf>
<SubObjectPropertyOf>
    <ObjectProperty IRI="#instantiatesFactType"/>
    <ObjectProperty abbreviatedIRI="owl:topObjectProperty"/>
</SubObjectPropertyOf>
<SubObjectPropertyOf>
    <ObjectProperty IRI="#instantiatesFactTypeAggregation"/>
    <ObjectProperty abbreviatedIRI="owl:topObjectProperty"/>
</SubObjectPropertyOf>
<SubObjectPropertyOf>
    <ObjectProperty IRI="#instantiatesPropertyType"/>
    <ObjectProperty abbreviatedIRI="owl:topObjectProperty"/>
</SubObjectPropertyOf>
<TransitiveObjectProperty>
    <ObjectProperty IRI="#subTypeOf"/>
</TransitiveObjectProperty>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#adaptedBy"/>
    <ObjectIntersectionOf>
        <Class IRI="#ReusableStructure"/>
        <ObjectIntersectionOf>
            <ObjectComplementOf>
                <Class IRI="#AnalysisRule"/>
            </ObjectComplementOf>
            <ObjectComplementOf>
                <Class IRI="#ProductionScript"/>
            </ObjectComplementOf>
        </ObjectIntersectionOf>
```

```
        </ObjectIntersectionOf>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#aggregatesChunkFacts"/>
        <Class IRI="#ModelChunk"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#aggregatesFactTypes"/>
        <Class IRI="#FactTypeAggregation"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#aggregatesFacts"/>
        <Class IRI="#FactAggregation"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#belongsToPhase"/>
        <Class IRI="#Activity"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#enforcesGoalFactType"/>
        <Class IRI="#AnalysisRule"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasActivityConstraint"/>
        <Class IRI="#IndexDescription"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasChunkFactTypeAggregation"/>
        <Class IRI="#ModelChunkType"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasConditionChunk"/>
        <Class IRI="#ChunkCondition"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasConditionRule"/>
        <Class IRI="#RuleCondition"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasCurrentActivity"/>
        <Class IRI="#ModellingContext"/>
    </ObjectPropertyDomain>
```

```
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasCurrentInformationRequirement"/>
    <Class IRI="#ModellingContext"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasCurrentPhase"/>
    <Class IRI="#ModellingContext"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasCurrentSourceModel"/>
    <Class IRI="#ModellingContext"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasCurrentTargetModel"/>
    <Class IRI="#ModellingContext"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasFactAggregation"/>
    <Class IRI="#Model"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasFactTypeAggregation"/>
    <Class IRI="#ModelType"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasFirstAdaptationRule"/>
    <Class IRI="#AdaptationScript"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasFirstProductionRule"/>
    <Class IRI="#ProductionScript"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasNextActivity"/>
    <Class IRI="#Activity"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasNextPhase"/>
    <Class IRI="#Phase"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasNextProductionRule"/>
```

```
        <Class IRI="#ProductionRule"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasPhaseConstraint"/>
        <Class IRI="#IndexDescription"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasPostconditionConstraint"/>
        <Class IRI="#IndexDescription"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasPreconditionConstraint"/>
        <Class IRI="#IndexDescription"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasProperty"/>
        <Class IRI="#ComplexFact"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasPropertyOfType"/>
        <Class IRI="#ComplexType"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasPropertyValue"/>
        <Class IRI="#Property"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasPropertyValueType"/>
        <Class IRI="#PropertyType"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasResponseOption"/>
        <Class IRI="#MultipleChoiceElicitationStimulus"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasResponseStructure"/>
        <Class IRI="#ChunkBasedElicitationStimulus"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#hasSourceFactTypeConstraint"/>
        <Class IRI="#IndexDescription"/>
    </ObjectPropertyDomain>
```

```
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasTargetFactTypeConstraint"/>
    <Class IRI="#IndexDescription"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasTriggerConditionConstraint"/>
    <Class IRI="#IndexDescription"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#indexedBy"/>
    <Class IRI="#ReusableStructure"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#instantiatesChunkType"/>
    <Class IRI="#ModelChunk"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#instantiatesFactType"/>
    <Class IRI="#Fact"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#instantiatesFactTypeAggregation"/>
    <Class IRI="#FactAggregation"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#instantiatesModelType"/>
    <Class IRI="#Model"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#instantiatesPropertyType"/>
    <Class IRI="#Property"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isSubsetOfModelType"/>
    <Class IRI="#ModelChunkType"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#producesModelOfType"/>
    <Class IRI="#Activity"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#requiresGoalFactType"/>
```

```
    <Class IRI="#InformationRequirement"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#subTypeOf"/>
    <Class IRI="#FactType"/>
</ObjectPropertyDomain>
<ObjectPropertyRange>
    <ObjectProperty IRI="#aggregatesChunkFacts"/>
    <Class IRI="#Fact"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#aggregatesFactTypes"/>
    <Class IRI="#FactType"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#aggregatesFacts"/>
    <Class IRI="#Fact"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#belongsToPhase"/>
    <Class IRI="#Phase"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#enforcesGoalFactType"/>
    <Class IRI="#FactType"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasActivityConstraint"/>
    <Class IRI="#Activity"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasChunkFactTypeAggregation"/>
    <Class IRI="#ChunkFactTypeAggregation"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasConditionChunk"/>
    <Class IRI="#ModelChunk"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasConditionRule"/>
    <Class IRI="#AnalysisRule"/>
</ObjectPropertyRange>
```

```
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasCurrentActivity"/>
    <Class IRI="#Activity"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasCurrentInformationRequirement"/>
    <Class IRI="#InformationRequirement"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasCurrentPhase"/>
    <Class IRI="#Phase"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasCurrentSourceModel"/>
    <Class IRI="#Model"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasCurrentTargetModel"/>
    <Class IRI="#Model"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasFactAggregation"/>
    <Class IRI="#FactAggregation"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasFactTypeAggregation"/>
    <Class IRI="#FactTypeAggregation"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasFirstAdaptationRule"/>
    <Class IRI="#ProductionRule"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasFirstProductionRule"/>
    <Class IRI="#ProductionRule"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasNextActivity"/>
    <Class IRI="#Activity"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasNextPhase"/>
```

```
        <Class IRI="#Phase"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasNextProductionRule"/>
        <Class IRI="#ProductionRule"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasPhaseConstraint"/>
        <Class IRI="#Phase"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasPostconditionConstraint"/>
        <Class IRI="#MatchingCondition"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasPreconditionConstraint"/>
        <Class IRI="#MatchingCondition"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasProperty"/>
        <Class IRI="#Property"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasPropertyOfType"/>
        <Class IRI="#PropertyType"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasPropertyValue"/>
        <Class IRI="#Fact"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasPropertyValueType"/>
        <Class IRI="#FactType"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasResponseOption"/>
        <Class IRI="#ModelChunk"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasResponseStructure"/>
        <Class IRI="#ModelChunkType"/>
    </ObjectPropertyRange>
```

```
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasSourceFactTypeConstraint"/>
    <Class IRI="#FactType"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasTargetFactTypeConstraint"/>
    <Class IRI="#FactType"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasTriggerConditionConstraint"/>
    <Class IRI="#MatchingCondition"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#indexedBy"/>
    <Class IRI="#IndexDescription"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#instantiatesChunkType"/>
    <Class IRI="#ModelChunkType"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#instantiatesFactType"/>
    <Class IRI="#FactType"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#instantiatesFactTypeAggregation"/>
    <Class IRI="#FactTypeAggregation"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#instantiatesModelType"/>
    <Class IRI="#ModelType"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#instantiatesPropertyType"/>
    <Class IRI="#PropertyType"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#isSubsetOfModelType"/>
    <Class IRI="#ModelType"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#producesModelOfType"/>
```

```
        <Class IRI="#ModelType"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#requiresGoalFactType"/>
        <Class IRI="#FactType"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#subTypeOf"/>
        <Class IRI="#FactType"/>
    </ObjectPropertyRange>
    <SubDataPropertyOf>
        <DataProperty IRI="#hasAntecedent"/>
        <DataProperty abbreviatedIRI="owl:topDataProperty"/>
    </SubDataPropertyOf>
    <SubDataPropertyOf>
        <DataProperty IRI="#hasBooleanValue"/>
        <DataProperty IRI="#hasSimpleValue"/>
    </SubDataPropertyOf>
    <SubDataPropertyOf>
        <DataProperty IRI="#hasInputFactQuery"/>
        <DataProperty abbreviatedIRI="owl:topDataProperty"/>
    </SubDataPropertyOf>
    <SubDataPropertyOf>
        <DataProperty IRI="#hasNumericValue"/>
        <DataProperty IRI="#hasSimpleValue"/>
    </SubDataPropertyOf>
    <SubDataPropertyOf>
        <DataProperty IRI="#hasSimpleValue"/>
        <DataProperty abbreviatedIRI="owl:topDataProperty"/>
    </SubDataPropertyOf>
    <SubDataPropertyOf>
        <DataProperty IRI="#hasStringValue"/>
        <DataProperty IRI="#hasSimpleValue"/>
    </SubDataPropertyOf>
    <DataPropertyDomain>
        <DataProperty IRI="#checksContextElement"/>
        <Class IRI="#MatchingCondition"/>
    </DataPropertyDomain>
    <DataPropertyDomain>
        <DataProperty IRI="#hasActivityName"/>
        <Class IRI="#Activity"/>
    </DataPropertyDomain>
```

```
<DataPropertyDomain>
    <DataProperty IRI="#hasAggregationName"/>
    <Class IRI="#FactTypeAggregation"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#hasAntecedent"/>
    <Class IRI="#RORERule"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#hasBooleanValue"/>
    <Class IRI="#BooleanFact"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#hasCardinality"/>
    <Class IRI="#ChunkFactTypeAggregation"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#hasConsequent"/>
    <Class IRI="#RORERule"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#hasCycleGoal"/>
    <Class IRI="#InformationRequirement"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#hasCyclePostcondition"/>
    <Class IRI="#InformationRequirement"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#hasElicitationStimulus"/>
    <Class IRI="#ElicitationStimulus"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#hasFactQuery"/>
    <Class IRI="#FactEditingElicitationStimulus"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#hasFactTypeName"/>
    <Class IRI="#FactType"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#hasInputFactQuery"/>
```

```
        <Class IRI="#ProductionScript"/>
    </DataPropertyDomain>
    <DataPropertyDomain>
        <DataProperty IRI="#hasModelTypeName"/>
        <Class IRI="#ModelType"/>
    </DataPropertyDomain>
    <DataPropertyDomain>
        <DataProperty IRI="#hasNumericValue"/>
        <Class IRI="#NumericFact"/>
    </DataPropertyDomain>
    <DataPropertyDomain>
        <DataProperty IRI="#hasPhaseName"/>
        <Class IRI="#Phase"/>
    </DataPropertyDomain>
    <DataPropertyDomain>
        <DataProperty IRI="#hasPropertyName"/>
        <Class IRI="#PropertyType"/>
    </DataPropertyDomain>
    <DataPropertyDomain>
        <DataProperty IRI="#hasResponseOptionsQuery"/>
        <Class IRI="#MultipleChoiceElicitationStimulus"/>
    </DataPropertyDomain>
    <DataPropertyDomain>
        <DataProperty IRI="#hasSimpleValue"/>
        <Class IRI="#SimpleType"/>
    </DataPropertyDomain>
    <DataPropertyDomain>
        <DataProperty IRI="#hasStringValue"/>
        <Class IRI="#StringFact"/>
    </DataPropertyDomain>
    <DataPropertyRange>
        <DataProperty IRI="#checksContextElement"/>
        <Datatype IRI="#ContextElement"/>
    </DataPropertyRange>
    <DataPropertyRange>
        <DataProperty IRI="#hasActivityName"/>
        <Datatype abbreviatedIRI="xsd:string"/>
    </DataPropertyRange>
    <DataPropertyRange>
        <DataProperty IRI="#hasAggregationName"/>
        <Datatype abbreviatedIRI="xsd:string"/>
    </DataPropertyRange>
```

```
<DataPropertyRange>
    <DataProperty IRI="#hasAntecedent"/>
    <Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
    <DataProperty IRI="#hasBooleanValue"/>
    <Datatype abbreviatedIRI="xsd:boolean"/>
</DataPropertyRange>
<DataPropertyRange>
    <DataProperty IRI="#hasCardinality"/>
    <Datatype abbreviatedIRI="xsd:int"/>
</DataPropertyRange>
<DataPropertyRange>
    <DataProperty IRI="#hasConsequent"/>
    <Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
    <DataProperty IRI="#hasCycleGoal"/>
    <Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
    <DataProperty IRI="#hasCyclePostcondition"/>
    <Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
    <DataProperty IRI="#hasElicitationStimulus"/>
    <Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
    <DataProperty IRI="#hasFactQuery"/>
    <Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
    <DataProperty IRI="#hasFactTypeName"/>
    <Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
    <DataProperty IRI="#hasInputFactQuery"/>
    <Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
    <DataProperty IRI="#hasModelTypeName"/>
```

```
        <Datatype abbreviatedIRI="xsd:string"/>
    </DataPropertyRange>
    <DataPropertyRange>
        <DataProperty IRI="#hasNumericValue"/>
        <Datatype abbreviatedIRI="xsd:integer"/>
    </DataPropertyRange>
    <DataPropertyRange>
        <DataProperty IRI="#hasPhaseName"/>
        <Datatype abbreviatedIRI="xsd:string"/>
    </DataPropertyRange>
    <DataPropertyRange>
        <DataProperty IRI="#hasPropertyName"/>
        <Datatype abbreviatedIRI="xsd:string"/>
    </DataPropertyRange>
    <DataPropertyRange>
        <DataProperty IRI="#hasResponseOptionsQuery"/>
        <Datatype abbreviatedIRI="xsd:string"/>
    </DataPropertyRange>
    <DataPropertyRange>
        <DataProperty IRI="#hasStringValue"/>
        <Datatype abbreviatedIRI="xsd:string"/>
    </DataPropertyRange>
    <DatatypeDefinition>
        <Datatype IRI="#ContextElement"/>
        <DataOneOf>
            <Literal datatypeIRI="&rdf;PlainLiteral">SourceModel</
                Literal>
            <Literal datatypeIRI="&rdf;PlainLiteral">TargetModel</
                Literal>
        </DataOneOf>
    </DatatypeDefinition>
</Ontology>

<!-- Generated by the OWL API (version 3.3.1957) http://owlapi.
    sourceforge.net -->
```

Appendix B

Long-Term Memory For OSM-ISM Transformations: Formalisation

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY swrl "http://www.w3.org/2003/11/swrl#" >
    <!ENTITY swrlb "http://www.w3.org/2003/11/swrlb#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <!ENTITY LongTermMemory "http://jamesnaish.wordpress.com/
        LongTermMemory/" >
    <!ENTITY ROREKnowledgeModel "http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#" >
]>
<rdf:RDF xmlns="&LongTermMemory;phd.owl#"
      xml:base="&LongTermMemory;phd.owl"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:swrl="http://www.w3.org/2003/11/swrl#"
      xmlns:ROREKnowledgeModel="http://jamesnaish.wordpress.com/
          ROREKnowledgeModel.owl#"
      xmlns:owl="http://www.w3.org/2002/07/owl#"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
      xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:LongTermMemory="http://jamesnaish.wordpress.com/
          LongTermMemory/">
    <owl:Ontology rdf:about="&LongTermMemory;phd.owl">
```

```
    <owl:imports rdf:resource="http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl"/>
</owl:Ontology>

<!--

//
// Datatypes
//
///////////////////////////////////////////////////////////////////////////////////

 -->

<!--
///////////////////////////////////////////////////////////////////////////////////

//
// Object Properties
//
///////////////////////////////////////////////////////////////////////////////////

 -->

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    aggregatesChunkFacts -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    aggregatesChunkFacts"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    aggregatesFactTypes -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    aggregatesFactTypes"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    belongsToPhase -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    belongsToPhase"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    enforcesGoalFactType -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    enforcesGoalFactType"/>
```

```
<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasActivityConstraint -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasActivityConstraint"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasChunkFactTypeAggregation -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasChunkFactTypeAggregation"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasConditionRule -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasConditionRule"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasFactTypeAggregation -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasFactTypeAggregation"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasFirstProductionRule -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasFirstProductionRule"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasNextProductionRule -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasNextProductionRule"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasPhaseConstraint -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasPhaseConstraint"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasPreconditionConstraint -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasPreconditionConstraint"/>
```

```
<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasProperty -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;hasProperty"/
    >

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasPropertyOfType -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasPropertyOfType"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasPropertyValue -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasPropertyValue"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasPropertyValueType -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasPropertyValueType"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasSourceFactTypeConstraint -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasSourceFactTypeConstraint"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasTargetFactTypeConstraint -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    hasTargetFactTypeConstraint"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    indexedBy -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;indexedBy"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    instantiatesChunkType -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    instantiatesChunkType"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    instantiatesFactType -->
```

```
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    instantiatesFactType"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    instantiatesPropertyType -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    instantiatesPropertyType"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    producesModelOfType -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;
    producesModelOfType"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    subTypeOf -->
<owl:ObjectProperty rdf:about="&ROREKnowledgeModel;subTypeOf"/>

<!--
///////////////////////////////////////////////////////////////////////////////

//
// Data properties
//
///////////////////////////////////////////////////////////////////////////////

 -->

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasAntecedent -->
<owl:DatatypeProperty rdf:about="&ROREKnowledgeModel;
    hasAntecedent"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasCardinality -->
<owl:DatatypeProperty rdf:about="&ROREKnowledgeModel;
    hasCardinality"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasConsequent -->
<owl:DatatypeProperty rdf:about="&ROREKnowledgeModel;
    hasConsequent"/>
```

```
<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasElicitationStimulus -->
<owl:DatatypeProperty rdf:about="&ROREKnowledgeModel;
    hasElicitationStimulus"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasFactQuery -->
<owl:DatatypeProperty rdf:about="&ROREKnowledgeModel;
    hasFactQuery"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasInputFactQuery -->
<owl:DatatypeProperty rdf:about="&ROREKnowledgeModel;
    hasInputFactQuery"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    hasStringValue -->
<owl:DatatypeProperty rdf:about="&ROREKnowledgeModel;
    hasStringValue"/>

<!--
///////////////////////////////////////////////////////////////////////////////

//
// Classes
//
///////////////////////////////////////////////////////////////////////////////

 -->

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    Activity -->
<owl:Class rdf:about="&ROREKnowledgeModel;Activity"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    AnalysisRule -->
<owl:Class rdf:about="&ROREKnowledgeModel;AnalysisRule"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    BooleanType -->
<owl:Class rdf:about="&ROREKnowledgeModel;BooleanType"/>
```

```
<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    ChunkFactTypeAggregation -->
<owl:Class rdf:about="&ROREKnowledgeModel;
    ChunkFactTypeAggregation"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    ComplexFact -->
<owl:Class rdf:about="&ROREKnowledgeModel;ComplexFact"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    ComplexType -->
<owl:Class rdf:about="&ROREKnowledgeModel;ComplexType"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    FactEditingElicitationStimulus -->
<owl:Class rdf:about="&ROREKnowledgeModel;
    FactEditingElicitationStimulus"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    FactTypeAggregation -->
<owl:Class rdf:about="&ROREKnowledgeModel;FactTypeAggregation"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    IndexDescription -->
<owl:Class rdf:about="&ROREKnowledgeModel;IndexDescription"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    ModelChunk -->
<owl:Class rdf:about="&ROREKnowledgeModel;ModelChunk"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    ModelChunkType -->
<owl:Class rdf:about="&ROREKnowledgeModel;ModelChunkType"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    ModelType -->
<owl:Class rdf:about="&ROREKnowledgeModel;ModelType"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    NumericType -->
<owl:Class rdf:about="&ROREKnowledgeModel;NumericType"/>
```

```
<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    Phase -->
<owl:Class rdf:about="&ROREKnowledgeModel;Phase"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    ProductionRule -->
<owl:Class rdf:about="&ROREKnowledgeModel;ProductionRule"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    ProductionScript -->
<owl:Class rdf:about="&ROREKnowledgeModel;ProductionScript"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    Property -->
<owl:Class rdf:about="&ROREKnowledgeModel;Property"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    PropertyType -->
<owl:Class rdf:about="&ROREKnowledgeModel;PropertyType"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    RuleCondition -->
<owl:Class rdf:about="&ROREKnowledgeModel;RuleCondition"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    StringFact -->
<owl:Class rdf:about="&ROREKnowledgeModel;StringFact"/>

<!-- http://jamesnaish.wordpress.com/ROREKnowledgeModel.owl#
    StringType -->
<owl:Class rdf:about="&ROREKnowledgeModel;StringType"/>

<!--
///////////////////////////////////////////////////////////////////////////////////////

//
// Individuals
//
///////////////////////////////////////////////////////////////////////////////////////

 -->
```

```
    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        AddMonitorReportRule -->
    <owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
        AddMonitorReportRule">
        <rdf:type rdf:resource="&ROREKnowledgeModel;ProductionRule"/
            >
        <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
            >PREFIX im: &lt;http://jamesnaish.wordpress.com/
            ROREKnowledgeModel.owl#&gt;
PREFIX ltm: &lt;http://jamesnaish.wordpress.com/LongTermMemory/phd.
    owl#&gt;
SELECT ?o WHERE { Individual(?o), Type(?o, im:ComplexFact),
    PropertyValue(?o, im:instantiatesFactType, ltm:KeyObject) }</
    ROREKnowledgeModel:hasAntecedent>
        <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
            >Reports &lt; (VAR {Reports, LIST:Report}; VAR {
            CListReport, Report}; VAR {CReport, Report}; VAR {
            CKeyObject, KeyObject}; VAR {CProperty, ISMProperty}) &lt
            ; ([CKeyObject &lt;- RESULTSET*: ASSIGN {CListReport,
            CREATE {$CKeyObject.name$ListReport, Report}}; ASSIGN {
            CListReport.hasField, CREATE{has$CKeyObject.name$s,
            String}}; ASSIGN {Reports, CListReport}; ASSIGN {CReport,
             CREATE {$CKeyObject.name$Report, Report}}; ASSIGN {
            CReport.hasField, CREATE {$CKeyObject.name$Name, String
            }}; [CProperty &lt;- CKeyObject.ALL:hasProperty*: ASSIGN
            {CReport.hasField, CREATE {$CKeyObject.name$$CProperty.
            name$, String}}]; ASSIGN {Reports, CReport}])</
            ROREKnowledgeModel:hasConsequent>
    </owl:NamedIndividual>

    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        AddMonitorReportScript -->
    <owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
        AddMonitorReportScript">
        <rdf:type rdf:resource="&ROREKnowledgeModel;ProductionScript
            "/>
        <ROREKnowledgeModel:hasInputFactQuery rdf:datatype="&xsd;
            string">PREFIX im: &lt;http://jamesnaish.wordpress.com/
            ROREKnowledgeModel.owl#&gt;
PREFIX ltm: &lt;http://jamesnaish.wordpress.com/LongTermMemory/phd.
    owl#&gt;
```

```
SELECT ?d WHERE { Individual(?d), Type(?d, im:ComplexFact),
    PropertyValue(?d, im:instantiatesFactType , ltm:Domain),
    PropertyValue(?d,  im:hasProperty , ?m), Type(?m, im:Property),
    PropertyValue(?m, im:hasPropertyValue , ?v), Type(?v,
    im:BooleanFact), PropertyValue(?v, im:hasBooleanValue , &quot;true
    &quot;)  }</ROREKnowledgeModel:hasInputFactQuery>
        <ROREKnowledgeModel:hasFirstProductionRule rdf:resource="&
            LongTermMemory;phd.owl#AddMonitorReportRule"/>
        <ROREKnowledgeModel:indexedBy rdf:resource="&LongTermMemory;
            phd.owl#NoMonitorReports"/>
    </owl:NamedIndividual>

    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        Agent -->
    <owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Agent">
        <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
        <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
            phd.owl#Object"/>
        <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
            LongTermMemory;phd.owl#hasAgentType"/>
        <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
            LongTermMemory;phd.owl#hasSecondaryState"/>
    </owl:NamedIndividual>

    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        AgentObjectResponsibilityAssignmentScript -->
    <owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
        AgentObjectResponsibilityAssignmentScript">
        <rdf:type rdf:resource="&ROREKnowledgeModel;ProductionRule"/
            >
        <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
            >AgentISMObjects &lt;
(VAR {CDomain, Domain}; VAR {KeyOSMObject, KeyObject}; VAR {
    AgentOSMObject, Agent}; VAR {AgentISMObjects, LIST:Agent}; VAR {
    AgentISMObject, ISMObject}; VAR {COperation, Operation}) &lt;
 ([CDomain &lt;- RESULTSET*:
                                ASSIGN {KeyOSMObject, CDomain.
                                    hasDomainGoal.hasGoalState.
                                    realisedBy.transformsStateOf};
                                ASSIGN {AgentOSMObject, CDomain.
                                    hasDomainGoal.hasGoalState.
                                    realisedBy.enactedBy};
```

```
ASSIGN {AgentISMObject, SELECT {
    target, &quot;SELECT ?o WHERE {
    Individual(?o), SameAs(?o, tm:$
    AgentOSMObject.name$Class) }&quot
    ;}};
IF {{CDomain.hasDomainGoal.hasName
    == &quot;Allocation&quot; -&gt;
                ASSIGN {COperation,
                    CREATE {allocate$
                    KeyOSMObject.name
                    $Objects,
                    Operation}};
                ASSIGN {
                    AgentISMObject.
                    hasOperation,
                    COperation}},
        {CDomain.hasDomainGoal.
            hasName == &quot;Control&
            quot; -&gt;
                ASSIGN {COperation,
                    CREATE {command$
                    KeyOSMObject.name
                    $Objects,
                    Operation}};
                ASSIGN {
                    AgentISMObject.
                    hasOperation,
                    COperation}},
        {CDomain.hasDomainGoal.
            hasName == &quot;
            Composition&quot; -&gt;
                ASSIGN {COperation,
                    CREATE {compose$
                    KeyOSMObject.name
                    $Components,
                    Operation}};
                ASSIGN {
                    AgentISMObject.
                    hasOperation,
                    COperation}},
```

```
{CDomain.hasDomainGoal.
    hasName == &quot;
    Decomposition&quot; -&gt;
        ASSIGN {COperation,
            CREATE {decompose
            $KeyOSMObject.
            name$Components,
            Operation}};
        ASSIGN {
            AgentISMObject.
            hasOperation,
            COperation}},
{CDomain.hasDomainGoal.
    hasName == &quot;
    Manipulation&quot; -&gt;
        ASSIGN {COperation,
            CREATE {
            manipulate$
            KeyOSMObject.name
            $Objects,
            Operation}};
        ASSIGN {
            AgentISMObject.
            hasOperation,
            COperation}},
{CDomain.hasDomainGoal.
    hasName == &quot;Sensing&
    quot; -&gt;
        ASSIGN {COperation,
            CREATE {sense$
            KeyOSMObject.name
            $Objects,
            Operation}};
        ASSIGN {
            AgentISMObject.
            hasOperation,
            COperation}},
{CDomain.hasDomainGoal.
    hasName == &quot;Transfer
    &quot; -&gt;
```

```
                                                    ASSIGN {COperation,
                                                        CREATE {transfer$
                                                        KeyOSMObject.name
                                                        $Objects,
                                                        Operation}};
                                                    ASSIGN {
                                                        AgentISMObject.
                                                        hasOperation,
                                                        COperation}}};
                                ASSIGN {AgentISMObjects,
                                    AgentISMObject}])</
                                    ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX ltm: &lt;http://jamesnaish.wordpress.com/
        LongTermMemory/phd.owl#&gt; PREFIX im: &lt;http://
        jamesnaish.wordpress.com/ROREKnowledgeModel.owl#&gt;
        SELECT ?x WHERE {  Type(?x, im:Fact), PropertyValue(?x,
        im:instantiatesFactType, ltm:Domain)  }</
        ROREKnowledgeModel:hasAntecedent>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    AgentType -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    AgentType">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    AlgorithmObjects -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    AlgorithmObjects">
    <rdf:type rdf:resource="&ROREKnowledgeModel;
        ChunkFactTypeAggregation"/>
    <ROREKnowledgeModel:hasCardinality rdf:datatype="&xsd;
        integer">0</ROREKnowledgeModel:hasCardinality>
    <ROREKnowledgeModel:aggregatesFactTypes rdf:resource="&
        LongTermMemory;phd.owl#ISMObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    AlgorithmOperations -->
```

```
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    AlgorithmOperations">
    <rdf:type rdf:resource="&ROREKnowledgeModel;
        ChunkFactTypeAggregation"/>
    <ROREKnowledgeModel:hasCardinality rdf:datatype="&xsd;
        integer">0</ROREKnowledgeModel:hasCardinality>
    <ROREKnowledgeModel:aggregatesFactTypes rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    AlgorithmResources -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    AlgorithmResources">
    <rdf:type rdf:resource="&ROREKnowledgeModel;
        ChunkFactTypeAggregation"/>
    <ROREKnowledgeModel:hasCardinality rdf:datatype="&xsd;
        integer">0</ROREKnowledgeModel:hasCardinality>
    <ROREKnowledgeModel:aggregatesFactTypes rdf:resource="&
        LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    AllocateAllObjectsAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    AllocateAllObjectsAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;StringFact"/>
    <ROREKnowledgeModel:hasStringValue rdf:datatype="&xsd;string
        ">DataStoreResource ruleSource = Context.
        currentAllocationRuleSource; Collection&lt;AllocationRule
        &gt; rules = ruleSource.allRules(); foreach (Object o:
        AllKeyObjectClasses) {          o.allocateObject(this.
        allocationRules); }</ROREKnowledgeModel:hasStringValue>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    AllocateAllObjectsByClassificationAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    AllocateAllObjectsByClassificationAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;StringFact"/>
```

```
    <ROREKnowledgeModel:hasStringValue rdf:datatype="&xsd;string
        ">DataStoreResource ruleSource = Context.
        currentAllocationRuleSource; Collection&lt;AllocationRule
        &gt; rules = ruleSource.allRules(); foreach (Object o:
        AllKeyObjectClasses) {          o.allocateObject(this.
        allocationRules); }</ROREKnowledgeModel:hasStringValue>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    AllocateObjectByClassificationAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    AllocateObjectByClassificationAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;StringFact"/>
    <ROREKnowledgeModel:hasStringValue rdf:datatype="&xsd;string
        ">foreach (AllocationRule r: allocationRules) {     if (r
        .satisfiedBy(this))     {          this.allocatedLocation
        = r.targetStructure;     } }</
        ROREKnowledgeModel:hasStringValue>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    AllocationRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    AllocationRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:hasProperty rdf:resource="&
        LongTermMemory;phd.owl#
        AllocationRulehasOperationsatisfiedBy"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#ISMObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    AllocationRulehasOperationsatisfiedBy -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    AllocationRulehasOperationsatisfiedBy">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
```

```
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasOperation"/>
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#satisfiedBy"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Boolean -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Boolean"
    >
    <rdf:type rdf:resource="&ROREKnowledgeModel;BooleanType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ClassificationAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ClassificationAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ModelChunk"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#AllocationRule"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#ClassificationRuleSource"/>
    <ROREKnowledgeModel:instantiatesChunkType rdf:resource="&
        LongTermMemory;phd.owl#GenericAlgorithm"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#allocateAllObjects"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#allocateObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ClassificationRuleSource -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ClassificationRuleSource">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    CompleteRemoteTransferAlgorithm -->
```

```
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    CompleteRemoteTransferAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ModelChunk"/>
    <ROREKnowledgeModel:instantiatesChunkType rdf:resource="&
        LongTermMemory;phd.owl#GenericAlgorithm"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#RemoteDataTransferProtocol"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#transferAllCompleteObjects"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#transferCompleteObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ControlInterface -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ControlInterface">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    DataSampleSensingAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    DataSampleSensingAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ModelChunk"/>
    <ROREKnowledgeModel:instantiatesChunkType rdf:resource="&
        LongTermMemory;phd.owl#GenericAlgorithm"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#SensorInterface"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#senseAllProperties"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#senseProperty"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    DataSinksSpecifiedRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    DataSinksSpecifiedRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;AnalysisRule"/>
```

```
        <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
            >PREFIX im: &lt;http://jamesnaish.wordpress.com/
            ROREKnowledgeModel.owl#&gt; PREFIX ltm: &lt;http://
            jamesnaish.wordpress.com/LongTermMemory/phd.owl#&gt; ASK
            { Individual(?p), Type(?p, im:Property), PropertyValue(?
            ismprop, im:instantiatesFactType, ltm:ISMProperty),
            PropertyValue(?resource, im:instantiatesFactType,
            ltm:Resource), PropertyValue(?ismprop, im:hasProperty, ?p
            ), PropertyValue(?p, im:hasPropertyValue, ?resource),
            PropertyValue(?p, im:instantiatesPropertyType,
            ltm:hasDataSink) }</ROREKnowledgeModel:hasAntecedent>
        <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
            >true</ROREKnowledgeModel:hasConsequent>
        <ROREKnowledgeModel:enforcesGoalFactType rdf:resource="&
            LongTermMemory;phd.owl#Resource"/>
    </owl:NamedIndividual>

    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        DataSourceAndSinkStimulus -->
    <owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
        DataSourceAndSinkStimulus">
        <rdf:type rdf:resource="&ROREKnowledgeModel;
            FactEditingElicitationStimulus"/>
        <ROREKnowledgeModel:hasElicitationStimulus rdf:datatype="&
            xsd;string">Please ensure that the data sources and data
            sinks for each property in the Object Model are
            adequately specified.</
            ROREKnowledgeModel:hasElicitationStimulus>
        <ROREKnowledgeModel:hasFactQuery rdf:datatype="&xsd;string">
            Target [PREFIX im: &lt;http://jamesnaish.wordpress.com/
            ROREKnowledgeModel.owl#&gt;
PREFIX ltm: &lt;http://jamesnaish.wordpress.com/LongTermMemory/phd.
    owl#&gt;
SELECT ?r
WHERE { Type(?r, im:Fact), PropertyValue (?r,
    im:instantiatesFactType, ltm:ISMObject) }]</
    ROREKnowledgeModel:hasFactQuery>
        <ROREKnowledgeModel:indexedBy rdf:resource="&LongTermMemory;
            phd.owl#NoDataSourceOrSinkIndex"/>
    </owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    DataSourcesSpecifiedRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    DataSourcesSpecifiedRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;AnalysisRule"/>
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt; PREFIX ltm: &lt;http://
        jamesnaish.wordpress.com/LongTermMemory/phd.owl#&gt; ASK
        { Individual(?p), Type(?p, im:Property), PropertyValue(?
        ismprop, im:instantiatesFactType, ltm:ISMProperty),
        PropertyValue(?resource, im:instantiatesFactType,
        ltm:Resource), PropertyValue(?ismprop, im:hasProperty, ?p
        ), PropertyValue(?p, im:hasPropertyValue, ?resource),
        PropertyValue(?p, im:instantiatesPropertyType,
        ltm:hasDataSource) }</ROREKnowledgeModel:hasAntecedent>
    <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
        >true</ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:enforcesGoalFactType rdf:resource="&
        LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    DataStore -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    DataStore">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Database -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Database
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#Database"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasTable"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    DatabaseAttribute -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    DatabaseAttribute">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasValueType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    DatabaseTable -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    DatabaseTable">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasAttribute"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    DispatchNotifiedSampleCommandAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    DispatchNotifiedSampleCommandAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;StringFact"/>
    <ROREKnowledgeModel:hasStringValue rdf:datatype="&xsd;string
        ">commandObject(Command c, Object o, Interface sensor) {
            sensor.dispatchCommand(c, o); } </
        ROREKnowledgeModel:hasStringValue>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Domain -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Domain">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasAgentType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasDomainGoal"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasDomainType"/>
```

```
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#isMonitored"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    DomainAlias -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    DomainAlias">
    <rdf:type rdf:resource="&ROREKnowledgeModel;
        ChunkFactTypeAggregation"/>
    <ROREKnowledgeModel:hasCardinality rdf:datatype="&xsd;
        integer">3</ROREKnowledgeModel:hasCardinality>
    <ROREKnowledgeModel:aggregatesFactTypes rdf:resource="&
        LongTermMemory;phd.owl#Domain"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    DomainChunk -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    DomainChunk">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ModelChunkType"/
        >
    <ROREKnowledgeModel:hasChunkFactTypeAggregation rdf:resource
        ="&LongTermMemory;phd.owl#DomainAlias"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    DomainGoal -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    DomainGoal">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasGoalState"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasName"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    DomainType -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    DomainType">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
```

```
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasGenericProperty"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasName"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Event -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Event">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#detectedBy"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#dispatchedBy"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#File
    -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#File">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#DataStore"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasStructure"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#Form
    -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Form">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#UserInterface"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#modelsObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    FunctionalResourceConfigurationStimulus -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    FunctionalResourceConfigurationStimulus">
    <rdf:type rdf:resource="&ROREKnowledgeModel;
        FactEditingElicitationStimulus"/>
```

```
    <ROREKnowledgeModel:hasFactQuery rdf:datatype="&xsd;string">
        Target [PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt;
            PREFIX ltm: &lt;http://jamesnaish.wordpress.com/
                LongTermMemory/phd.owl#&gt;
            SELECT ?r WHERE { Individual(?r), Type(?r, im:Fact),
                PropertyValue(?r, im:instantiatesFactType,
                ltm:Resource) }]
            </ROREKnowledgeModel:hasFactQuery>
    <ROREKnowledgeModel:hasElicitationStimulus rdf:datatype="&
        xsd;string">Please ensure that each functional resource
        in the Resource Model has been fully specified.</
        ROREKnowledgeModel:hasElicitationStimulus>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    FunctionalResponsibilitiesNotAssignedCondition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    FunctionalResponsibilitiesNotAssignedCondition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;RuleCondition"/>
    <ROREKnowledgeModel:hasConditionRule rdf:resource="&
        LongTermMemory;phd.owl#
        FunctionalResponsibilitiesNotAssignedRule"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    FunctionalResponsibilitiesNotAssignedIndex -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    FunctionalResponsibilitiesNotAssignedIndex">
    <rdf:type rdf:resource="&ROREKnowledgeModel;IndexDescription
        "/>
    <ROREKnowledgeModel:hasSourceFactTypeConstraint rdf:resource
        ="&LongTermMemory;phd.owl#DomainGoal"/>
    <ROREKnowledgeModel:hasPreconditionConstraint rdf:resource="
        &LongTermMemory;phd.owl#
        FunctionalResponsibilitiesNotAssignedCondition"/>
    <ROREKnowledgeModel:hasActivityConstraint rdf:resource="&
        LongTermMemory;phd.owl#GenerateSoftwareSpecifications"/>
    <ROREKnowledgeModel:hasTargetFactTypeConstraint rdf:resource
        ="&LongTermMemory;phd.owl#Operation"/>
    <ROREKnowledgeModel:hasPhaseConstraint rdf:resource="&
        LongTermMemory;phd.owl#Requirements"/>
```

```
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    FunctionalResponsibilitiesNotAssignedRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    FunctionalResponsibilitiesNotAssignedRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;AnalysisRule"/>
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt; PREFIX ltm: &lt;http://
        jamesnaish.wordpress.com/LongTermMemory/phd.owl#&gt; ASK
        { Individual(?r), Type(?d, im:Fact), PropertyValue(?d,
        im:instantiatesFactType, ltm:Operation) }</
        ROREKnowledgeModel:hasAntecedent>
    <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
        >false</ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:enforcesGoalFactType rdf:resource="&
        LongTermMemory;phd.owl#Agent"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    FunctionalResponsibilityAssignmentScript -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    FunctionalResponsibilityAssignmentScript">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ProductionScript
        "/>
    <ROREKnowledgeModel:hasInputFactQuery rdf:datatype="&xsd;
        string">PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt;
            PREFIX ltm: &lt;http://jamesnaish.wordpress.com/
                LongTermMemory/phd.owl#&gt;
            SELECT ?d WHERE { Individual(?d), Type(?d, im:Fact),
                 PropertyValue(?d, im:instantiatesFactType,
                ltm:Domain) } </
                ROREKnowledgeModel:hasInputFactQuery>
    <ROREKnowledgeModel:indexedBy rdf:resource="&LongTermMemory;
        phd.owl#FunctionalResponsibilitiesNotAssignedIndex"/>
    <ROREKnowledgeModel:hasFirstProductionRule rdf:resource="&
        LongTermMemory;phd.owl#
        KeyObjectResponsibilityAssignmentScript"/>
</owl:NamedIndividual>
```

```
<!— http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    FunctionalResponsibilityEditingStimulus —>
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    FunctionalResponsibilityEditingStimulus">
    <rdf:type rdf:resource="&ROREKnowledgeModel;
        FactEditingElicitationStimulus"/>
    <ROREKnowledgeModel:hasElicitationStimulus rdf:datatype="&
        xsd;string">Please ensure the relevant operations are
        assigned to each class.</
        ROREKnowledgeModel:hasElicitationStimulus>
    <ROREKnowledgeModel:hasFactQuery rdf:datatype="&xsd;string">
        Target [PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt;
PREFIX ltm: &lt;http://jamesnaish.wordpress.com/LongTermMemory/phd.
    owl#&gt;
SELECT ?o
WHERE {  Type(?o, im:Fact), PropertyValue (?o,
    im:instantiatesFactType, ltm:ISMObject)  }]</
    ROREKnowledgeModel:hasFactQuery>
    </owl:NamedIndividual>

<!— http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    GenerateSoftwareSpecifications —>
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    GenerateSoftwareSpecifications">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Activity"/>
    <ROREKnowledgeModel:producesModelOfType rdf:resource="&
        LongTermMemory;phd.owl#InformationSystemModel"/>
    <ROREKnowledgeModel:belongsToPhase rdf:resource="&
        LongTermMemory;phd.owl#Requirements"/>
</owl:NamedIndividual>

<!— http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    GenericAlgorithm —>
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    GenericAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ModelChunkType"/
        >
    <ROREKnowledgeModel:hasChunkFactTypeAggregation rdf:resource
        ="&LongTermMemory;phd.owl#AlgorithmObjects"/>
    <ROREKnowledgeModel:hasChunkFactTypeAggregation rdf:resource
        ="&LongTermMemory;phd.owl#AlgorithmOperations"/>
```

```
    <ROREKnowledgeModel:hasChunkFactTypeAggregation rdf:resource
        ="&LongTermMemory;phd.owl#AlgorithmResources"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    GoalModel -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    GoalModel">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasAgentType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasDomain"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasDomainGoal"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    GoalState -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    GoalState">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#modelledByState"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#realisedBy"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    HasDataSinkCondition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    HasDataSinkCondition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;RuleCondition"/>
    <ROREKnowledgeModel:hasConditionRule rdf:resource="&
        LongTermMemory;phd.owl#HasDataSinkRule"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    HasDataSinkRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    HasDataSinkRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;AnalysisRule"/>
```

```
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt; PREFIX ltm: &lt;http://
        jamesnaish.wordpress.com/LongTermMemory/phd.owl#&gt; ASK
        { Individual(?p), Type(?p, im:Property), PropertyValue(?
        ismprop, im:instantiatesFactType, ltm:ISMProperty),
        PropertyValue(?resource, im:instantiatesFactType,
        ltm:Resource), PropertyValue(?ismprop, im:hasProperty, ?p
        ), PropertyValue(?p, im:hasPropertyValue, ?resource),
        PropertyValue(?p, im:instantiatesPropertyType,
        ltm:hasDataSink) }</ROREKnowledgeModel:hasAntecedent>
    <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
        >false</ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:enforcesGoalFactType rdf:resource="&
        LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    HasDataSourceOrSinkCondition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    HasDataSourceOrSinkCondition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;RuleCondition"/>
    <ROREKnowledgeModel:hasConditionRule rdf:resource="&
        LongTermMemory;phd.owl#HasDataSourceOrSinkRule"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    HasDataSourceOrSinkRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    HasDataSourceOrSinkRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;AnalysisRule"/>
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt; PREFIX ltm: &lt;http://
        jamesnaish.wordpress.com/LongTermMemory/phd.owl#&gt; ASK
        { Individual(?p), Type(?p, im:Property), PropertyValue(?
        ismprop, im:instantiatesFactType, ltm:ISMProperty),
        PropertyValue(?resource, im:instantiatesFactType,
        ltm:Resource), PropertyValue(?ismprop, im:hasProperty, ?p
        ), PropertyValue(?p, im:hasPropertyValue, ?resource),
        PropertyValue(?p, im:instantiatesPropertyType,
        ltm:hasDataSource) }</ROREKnowledgeModel:hasAntecedent>
```

```
    <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
        >false</ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:enforcesGoalFactType rdf:resource="&
        LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    HasFunctionalResponsibilitiesRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    HasFunctionalResponsibilitiesRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;AnalysisRule"/>
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt; PREFIX ltm: &lt;http://
        jamesnaish.wordpress.com/LongTermMemory/phd.owl#&gt; ASK
        { Individual(?r), Type(?d, im:Fact), PropertyValue(?d,
        im:instantiatesFactType, ltm:Operation) }</
        ROREKnowledgeModel:hasAntecedent>
    <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
        >true</ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:enforcesGoalFactType rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    HasISMObjectsRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    HasISMObjectsRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;AnalysisRule"/>
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt; PREFIX ltm: &lt;http://
        jamesnaish.wordpress.com/LongTermMemory/phd.owl#&gt; ASK
         { Individual(?o), Type(?o, im:Fact), PropertyValue(?o,
        im:instantiatesFactType, ltm:ISMObject) }</
        ROREKnowledgeModel:hasAntecedent>
    <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
        >true</ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:enforcesGoalFactType rdf:resource="&
        LongTermMemory;phd.owl#ISMObject"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    HasMonitorReportRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    HasMonitorReportRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;AnalysisRule"/>
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt; PREFIX ltm: &lt;http://
        jamesnaish.wordpress.com/LongTermMemory/phd.owl#&gt; ASK
        { Individual(?r), Type(?d, im:Fact), PropertyValue(?d,
        im:instantiatesFactType, ltm:Report) }</
        ROREKnowledgeModel:hasAntecedent>
    <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
        >true</ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:enforcesGoalFactType rdf:resource="&
        LongTermMemory;phd.owl#Report"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ISMGoalModel -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ISMGoalModel">
    <rdf:type rdf:resource="&ROREKnowledgeModel;
        FactTypeAggregation"/>
    <ROREKnowledgeModel:aggregatesFactTypes rdf:resource="&
        LongTermMemory;phd.owl#GoalModel"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ISMObject -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ISMObject">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasISMProperty"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasOperation"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasSuperclass"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ISMObjectModel -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ISMObjectModel">
    <rdf:type rdf:resource="&ROREKnowledgeModel;
        FactTypeAggregation"/>
    <ROREKnowledgeModel:aggregatesFactTypes rdf:resource="&
        LongTermMemory;phd.owl#ObjectModel"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ISMProcessModel -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ISMProcessModel">
    <rdf:type rdf:resource="&ROREKnowledgeModel;
        FactTypeAggregation"/>
    <ROREKnowledgeModel:aggregatesFactTypes rdf:resource="&
        LongTermMemory;phd.owl#ProcessModel"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ISMProperty -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ISMProperty">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasDataSink"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasDataSource"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasISMPropertyValue"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasPropertyType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ISMResourceModel -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ISMResourceModel">
    <rdf:type rdf:resource="&ROREKnowledgeModel;
        FactTypeAggregation"/>
```

```
    <ROREKnowledgeModel:aggregatesFactTypes rdf:resource="&
        LongTermMemory;phd.owl#ResourceModel"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    InformationSystemModel -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    InformationSystemModel">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ModelType"/>
    <ROREKnowledgeModel:hasFactTypeAggregation rdf:resource="&
        LongTermMemory;phd.owl#ISMGoalModel"/>
    <ROREKnowledgeModel:hasFactTypeAggregation rdf:resource="&
        LongTermMemory;phd.owl#ISMObjectModel"/>
    <ROREKnowledgeModel:hasFactTypeAggregation rdf:resource="&
        LongTermMemory;phd.owl#ISMProcessModel"/>
    <ROREKnowledgeModel:hasFactTypeAggregation rdf:resource="&
        LongTermMemory;phd.owl#ISMResourceModel"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Integer -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Integer"
    >
    <rdf:type rdf:resource="&ROREKnowledgeModel;NumericType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Interface -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    Interface">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    KeyObject -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    KeyObject">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#Object"/>
```

```
        <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
            LongTermMemory;phd.owl#hasPrimaryState"/>
        <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
            LongTermMemory;phd.owl#hasSecondaryState"/>
    </owl:NamedIndividual>

    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        KeyObjectResponsibilityAssignmentScript -->
    <owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
        KeyObjectResponsibilityAssignmentScript">
        <rdf:type rdf:resource="&ROREKnowledgeModel;ProductionRule"/
            >
        <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
            >KeyISMObjects &lt;
(VAR {CDomain, Domain}; VAR {KeyOSMObject, KeyObject}; VAR {
    KeyISMObjects, LIST:KeyObject}; VAR {KeyISMObject, KeyObject};
    VAR {COperation, Operation}; VAR {Parameters, LIST:Parameter};
    VAR {CParameter, Parameter}; VAR {OperationName, String}) &lt;
([CDomain &lt;- RESULTSET*:
                                    ASSIGN {KeyOSMObject, CDomain.
                                        hasDomainGoal.hasGoalState.
                                        realisedBy.transformsStateOf};
                                    ASSIGN {KeyISMObject, SELECT {target
                                        , &quot;SELECT ?o WHERE {
                                        Individual(?o), SameAs(?o, tm:$
                                        KeyOSMObject.name$Class) }&quot
                                        ;}};
                                    IF {{CDomain.hasDomainGoal.hasName
                                        == &quot;Allocation&quot; -&gt;
                                                        ASSIGN {
                                                            COperation
                                                            , CREATE
                                                            {
                                                            allocateTo
                                                            $
                                                            KeyOSMObject
                                                            .name$,
                                                            Operation
                                                            }};
```

```
ASSIGN {
    Parameters
    ,
    LIST:Parameter
    };
ASSIGN {
    CParameter
    , CREATE
    {$
    KeyOSMObject
    .name$
    TargetStructure
    ,
    Parameter
    }};
ASSIGN {
    CParameter
    .
    hasValueType
    , SELECT
    {source ,
    &quot;
    SELECT ?t
     WHERE {
    SameAs(?t
    ,  wm:$
    CDomain.
    hasDomainGoal
    .
    hasGoalState
    .
    realisedBy
    .
    producesState
    .
    containsInStructure
    .name$
    Class) }&
    quot;}};
```

```
                                        ASSIGN {
                                            Parameters
                                            ,
                                            CParameter
                                            };
                                        ASSIGN {
                                            COperation
                                            .
                                            hasParameter
                                            ,
                                            Parameters
                                            };
                                        ASSIGN {
                                            KeyISMObject
                                            .
                                            hasOperation
                                            ,
                                            COperation
                                            }},
                                {CDomain.hasDomainGoal.
                                    hasName == &quot;Control&
                                    quot; -&gt;
                                        ASSIGN {
                                            COperation
                                            , CREATE
                                            {
                                            issueCommandTo
                                            $
                                            KeyOSMObject
                                            .name$,
                                            Operation
                                            }};
                                        ASSIGN {
                                            Parameters
                                            ,
                                            LIST:Parameter
                                            };
```

```
                    ASSIGN {
                        CParameter
                        , CREATE
                        {$
                        KeyOSMObject
                        .name$
                        Command,
                        Parameter
                        }};
                    ASSIGN {
                        Parameters
                        ,
                        CParameter
                        }; ASSIGN
                         {
                        COperation
                        .
                        hasParameter
                        ,
                        Parameters
                        };
                    ASSIGN {
                        KeyISMObject
                        .
                        hasOperation
                        ,
                        COperation
                        }},
{CDomain.hasDomainGoal.
    hasName == &quot;
    Composition&quot; –&gt;
                    ASSIGN {
                        COperation
                        , CREATE
                        {
                        addComponentTo
                        $
                        KeyOSMObject
                        .name$,
                        Operation
                        }};
```

```
ASSIGN {
    Parameters
    ,
    LIST:Parameter
    };
ASSIGN {
    CParameter
    , CREATE
    {$
    KeyOSMObject
    .name$
    Component
    ,
    Parameter
    }};
ASSIGN {
    CParameter
    .
    hasValueType
    , SELECT
    {source ,
    &quot;
    SELECT ?t
     WHERE {
    SameAs(?t
    , wm:$
    CDomain.
    hasDomainGoal
    .
    hasGoalState
    .
    realisedBy
    .
    transformsStateOf
    .name$
    Class) }&
    quot;}};
ASSIGN {
    Parameters
    ,
    CParameter
    };
```

```
                    ASSIGN {
                        COperation
                        .
                        hasParameter
                        ,
                        Parameters
                        };
                    ASSIGN {
                        KeyISMObject
                        .
                        hasOperation
                        ,
                        COperation
                        }},
{CDomain.hasDomainGoal.
    hasName == &quot;
    Decomposition&quot; –&gt;
                    ASSIGN {
                        COperation
                        , CREATE
                        {
                        removeComponentFrom
                        $
                        KeyOSMObject
                        .name$,
                        Operation
                        }};
                    ASSIGN {
                        Parameters
                        ,
                        LIST:Parameter
                        };
                    ASSIGN {
                        CParameter
                        , CREATE
                        {$
                        KeyOSMObject
                        .name$
                        Component
                        ,
                        Parameter
                        }};
```

```
ASSIGN {
    CParameter
    .
    hasValueType
    , SELECT
    {source ,
    &quot;
    SELECT ?t
     WHERE {
    SameAs(?t
    , wm:$
    CDomain.
    hasDomainGoal
    .
    hasGoalState
    .
    realisedBy
    .
    transformsStateOf
    .name$
    Class) }&
    quot;}};
ASSIGN {
    Parameters
    ,
    CParameter
    };
ASSIGN {
    COperation
    .
    hasParameter
    ,
    Parameters
    };
ASSIGN {
    KeyISMObject
    .
    hasOperation
    ,
    COperation
    }},
```

```
{CDomain.hasDomainGoal.
    hasName == &quot;
    Manipulation&quot; –&gt;
                ASSIGN {
                    COperation
                    , CREATE
                    {
                    manipulatePropertyOf
                    $
                    KeyOSMObject
                    .name$,
                    Operation
                    }};
                ASSIGN {
                    Parameters
                    ,
                    LIST:Parameter
                    };
                ASSIGN {
                    CParameter
                    , CREATE
                    {$
                    KeyOSMObject
                    .name$
                    property ,

                    Parameter
                    }};
                ASSIGN {
                    CParameter
                    .
                    hasValueType
                    , SELECT
                    {source ,
                    &quot;
                    SELECT ?t
                     WHERE {
                    SameAs(?t
                    ,
                    ltm:ISMProperty
                    ) }&quot
                    ;}};
```

```
                                        ASSIGN {
                                            Parameters
                                            ,
                                            CParameter
                                            };
                                        ASSIGN {
                                            CParameter
                                            , CREATE
                                            {$
                                            KeyOSMObject
                                            .name$
                                            PropertyValue
                                            ,
                                            Parameter
                                            }};
                                        ASSIGN {
                                            Parameters
                                            ,
                                            CParameter
                                            };
                                        ASSIGN {
                                            COperation
                                            .
                                            hasParameter
                                            ,
                                            Parameters
                                            };
                                        ASSIGN {
                                            KeyISMObject
                                            .
                                            hasOperation
                                            ,
                                            COperation
                                            }},
                                {CDomain.hasDomainGoal.
                                    hasName == &quot;Sensing&
                                    quot; –&gt;
```

```
ASSIGN {
    COperation
    , CREATE
    {
    sensePropertyOf
    $
    KeyOSMObject
    .name$,
    Operation
    }};
ASSIGN {
    Parameters
    ,
    LIST:Parameter
    };
ASSIGN {
    CParameter
    , CREATE
    {$
    KeyOSMObject
    .name$
    Property ,

    Parameter
    }};
ASSIGN {
    Parameters
    ,
    CParameter
    };
ASSIGN {
    COperation
    .
    hasParameter
    ,
    Parameters
    };
```

```
                                        ASSIGN {
                                            KeyISMObject
                                            .
                                            hasOperation
                                            ,
                                            COperation
                                            }},
                                {CDomain.hasDomainGoal.
                                    hasName == &quot;Transfer
                                    &quot; -&gt;
                                        ASSIGN {
                                            COperation
                                            , CREATE
                                            {transfer
                                            $
                                            KeyOSMObject
                                            .name$To,

                                            Operation
                                            }};
                                        ASSIGN {
                                            Parameters
                                            ,
                                            LIST:Parameter
                                            };
                                        ASSIGN {
                                            CParameter
                                            , CREATE
                                            {$
                                            KeyOSMObject
                                            .name$
                                            TargetStructure
                                            ,
                                            Parameter
                                            }};
```

```
ASSIGN {
    CParameter
    .
    hasValueType
    , SELECT
    {source ,
    &quot;
    SELECT ?t
     WHERE {
    SameAs(?t
    , wm:$
    CDomain.
    hasDomainGoal
    .
    hasGoalState
    .
    realisedBy
    .
    producesState
    .
    containsInStructure
    .name$
    Class) }&
    quot;}};
ASSIGN {
    Parameters
    ,
    CParameter
    };
ASSIGN {
    COperation
    .
    hasParameter
    ,
    Parameters
    };
```

```
                                                    ASSIGN {
                                                        KeyISMObject
                                                        .
                                                        hasOperation
                                                        ,
                                                        COperation
                                                        }}};
                                                    ASSIGN {
                                                        KeyISMObjects
                                                        ,
                                                        KeyISMObject
                                                        }])</
                                                        ROREKnowledgeModel:hasConseq
                                                        >
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX ltm: &lt;http://jamesnaish.wordpress.com/
        LongTermMemory/phd.owl#&gt;
            PREFIX im: &lt;http://jamesnaish.wordpress.com/
                ROREKnowledgeModel.owl#&gt;
            SELECT ?d WHERE { Type(?d, im:Fact), PropertyValue(?
                d, im:instantiatesFactType, ltm:Domain) }</
                ROREKnowledgeModel:hasAntecedent>
    <ROREKnowledgeModel:hasNextProductionRule rdf:resource="&
        LongTermMemory;phd.owl#
        AgentObjectResponsibilityAssignmentScript"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    NoDataSourceOrSinkIndex -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    NoDataSourceOrSinkIndex">
    <rdf:type rdf:resource="&ROREKnowledgeModel;IndexDescription
        "/>
    <ROREKnowledgeModel:hasSourceFactTypeConstraint rdf:resource
        ="&LongTermMemory;phd.owl#Agent"/>
    <ROREKnowledgeModel:hasActivityConstraint rdf:resource="&
        LongTermMemory;phd.owl#GenerateSoftwareSpecifications"/>
    <ROREKnowledgeModel:hasPreconditionConstraint rdf:resource="
        &LongTermMemory;phd.owl#HasDataSinkCondition"/>
    <ROREKnowledgeModel:hasPreconditionConstraint rdf:resource="
        &LongTermMemory;phd.owl#HasDataSourceOrSinkCondition"/>
```

```
    <ROREKnowledgeModel:hasPhaseConstraint rdf:resource="&
        LongTermMemory;phd.owl#Requirements"/>
    <ROREKnowledgeModel:hasTargetFactTypeConstraint rdf:resource
        ="&LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    NoISMObjectsIndex -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    NoISMObjectsIndex">
    <rdf:type rdf:resource="&ROREKnowledgeModel;IndexDescription
        "/>
    <ROREKnowledgeModel:hasActivityConstraint rdf:resource="&
        LongTermMemory;phd.owl#GenerateSoftwareSpecifications"/>
    <ROREKnowledgeModel:hasTargetFactTypeConstraint rdf:resource
        ="&LongTermMemory;phd.owl#ISMObject"/>
    <ROREKnowledgeModel:hasPreconditionConstraint rdf:resource="
        &LongTermMemory;phd.owl#NoISMObjectsRule"/>
    <ROREKnowledgeModel:hasSourceFactTypeConstraint rdf:resource
        ="&LongTermMemory;phd.owl#Object"/>
    <ROREKnowledgeModel:hasPhaseConstraint rdf:resource="&
        LongTermMemory;phd.owl#Requirements"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    NoISMObjectsRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    NoISMObjectsRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;RuleCondition"/>
    <ROREKnowledgeModel:hasConditionRule rdf:resource="&
        LongTermMemory;phd.owl#NoISMObjectsRuleCondition"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    NoISMObjectsRuleCondition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    NoISMObjectsRuleCondition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;AnalysisRule"/>
```

```
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt; PREFIX ltm: &lt;http://
        jamesnaish.wordpress.com/LongTermMemory/phd.owl#&gt; ASK
        { Individual(?o), Type(?o, im:Fact), PropertyValue(?o,
        im:instantiatesFactType, ltm:ISMObject) }</
        ROREKnowledgeModel:hasAntecedent>
    <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
        >false</ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:enforcesGoalFactType rdf:resource="&
        LongTermMemory;phd.owl#ISMObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    NoMonitorReports -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    NoMonitorReports">
    <rdf:type rdf:resource="&ROREKnowledgeModel;IndexDescription
        "/>
    <ROREKnowledgeModel:hasActivityConstraint rdf:resource="&
        LongTermMemory;phd.owl#GenerateSoftwareSpecifications"/>
    <ROREKnowledgeModel:hasSourceFactTypeConstraint rdf:resource
        ="&LongTermMemory;phd.owl#KeyObject"/>
    <ROREKnowledgeModel:hasPreconditionConstraint rdf:resource="
        &LongTermMemory;phd.owl#NoMonitorReportsCondition"/>
    <ROREKnowledgeModel:hasTargetFactTypeConstraint rdf:resource
        ="&LongTermMemory;phd.owl#Report"/>
    <ROREKnowledgeModel:hasPhaseConstraint rdf:resource="&
        LongTermMemory;phd.owl#Requirements"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    NoMonitorReportsCondition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    NoMonitorReportsCondition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;RuleCondition"/>
    <ROREKnowledgeModel:hasConditionRule rdf:resource="&
        LongTermMemory;phd.owl#NoMonitorReportsRule"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    NoMonitorReportsRule -->
```

```
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    NoMonitorReportsRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;AnalysisRule"/>
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt; PREFIX ltm: &lt;http://
        jamesnaish.wordpress.com/LongTermMemory/phd.owl#&gt; ASK
        { Individual(?r), Type(?d, im:Fact), PropertyValue(?d,
        im:instantiatesFactType, ltm:Report) }</
        ROREKnowledgeModel:hasAntecedent>
    <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
        >false</ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:enforcesGoalFactType rdf:resource="&
        LongTermMemory;phd.owl#Agent"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    NotifiedSampleCommand -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    NotifiedSampleCommand">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#ISMObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    NotifiedSampleControlAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    NotifiedSampleControlAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ModelChunk"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#ControlInterface"/>
    <ROREKnowledgeModel:instantiatesChunkType rdf:resource="&
        LongTermMemory;phd.owl#GenericAlgorithm"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#NotifiedSampleCommand"/>
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#dispatchAllNotifiedSampleCommands"
        />
    <ROREKnowledgeModel:aggregatesChunkFacts rdf:resource="&
        LongTermMemory;phd.owl#dispatchNotifiedSampleCommand"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    OSMDomain -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    OSMDomain">
    <rdf:type rdf:resource="&ROREKnowledgeModel;
        FactTypeAggregation"/>
    <ROREKnowledgeModel:aggregatesFactTypes rdf:resource="&
        LongTermMemory;phd.owl#Domain"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Object -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Object">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasProperty"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ObjectGenerationRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ObjectGenerationRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ProductionRule"/
        >
    <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
        >ISMObjects &lt; (VAR {ISMObjects, LIST:ISMObject}; VAR {
        SpecObject, ISMObject}) &lt; ([Object &lt;- RESULTSET*:
        ASSIGN{SpecObject, CREATE {$Object.name$Class, ISMObject
        }};[Property &lt;- Object.ALL:hasProperty*: ASSIGN {
        SpecObject.hasISMProperty, CREATE {$Property.name$,
        ISMProperty}}; ASSIGN {$Property.name$.hasPropertyType,
        Property.hasValueType}]; ASSIGN {ISMObjects, SpecObject
        }])</ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt;
            PREFIX ltm: &lt;http://jamesnaish.wordpress.com/
                LongTermMemory/phd.owl#&gt;
            SELECT ?o WHERE { Individual(?o), Type(?o, im:Fact),
                 PropertyValue(?o, im:instantiatesFactType,
                ltm:Object) }
```

```
            OR WHERE { Individual(?o), Type(?o, im:Fact),
                PropertyValue(?o, im:instantiatesFactType , ?t),
                PropertyValue(?t, im:subTypeOf, ltm:Object) }</
                ROREKnowledgeModel:hasAntecedent>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ObjectModel -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ObjectModel">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#aggregatesObjectSystem"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ObjectModelGenerationScript -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ObjectModelGenerationScript">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ProductionScript
        "/>
    <ROREKnowledgeModel:hasInputFactQuery rdf:datatype="&xsd;
        string">PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl#&gt;
            PREFIX ltm: &lt;http://jamesnaish.wordpress.com/
                LongTermMemory/phd.owl#&gt;
            SELECT ?o WHERE { Individual(?o), Type(?o, im:Fact),
                 PropertyValue(?o, im:instantiatesFactType ,
                ltm:Object) }
            OR WHERE { Individual(?o), Type(?o, im:Fact),
                PropertyValue(?o, im:instantiatesFactType , ?t),
                PropertyValue(?t, im:subTypeOf, ltm:Object) }</
                ROREKnowledgeModel:hasInputFactQuery>
    <ROREKnowledgeModel:hasFirstProductionRule rdf:resource="&
        LongTermMemory;phd.owl#ObjectGenerationRule"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ObjectProperty -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ObjectProperty">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
```

```
        <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
            LongTermMemory;phd.owl#hasValueType"/>
    </owl:NamedIndividual>

    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        ObjectPropertyStimulus -->
    <owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
        ObjectPropertyStimulus">
        <rdf:type rdf:resource="&ROREKnowledgeModel;
            FactEditingElicitationStimulus"/>
        <ROREKnowledgeModel:hasFactQuery rdf:datatype="&xsd;string">
            Source [PREFIX im: &lt;http://jamesnaish.wordpress.com/
            ROREKnowledgeModel.owl#&gt;
PREFIX ltm: &lt;http://jamesnaish.wordpress.com/LongTermMemory/phd.
    owl#&gt;
SELECT ?o
WHERE { Individual(?o), Type(?o, im:Fact), PropertyValue(?o,
    im:instantiatesFactType, ltm:Object) }
OR WHERE { Individual(?o), Type(?o, im:Fact), PropertyValue(?o,
    im:instantiatesFactType, ?t), PropertyValue(?t, im:subTypeOf,
    ltm:Object) }]</ROREKnowledgeModel:hasFactQuery>
        <ROREKnowledgeModel:hasElicitationStimulus rdf:datatype="&
            xsd;string">There are objects whose properties are
            insufficient. Please specify the properties for each
            object.</ROREKnowledgeModel:hasElicitationStimulus>
    </owl:NamedIndividual>

    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        ObjectPropertyValue -->
    <owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
        ObjectPropertyValue">
        <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
        <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
            LongTermMemory;phd.owl#valueIs"/>
    </owl:NamedIndividual>

    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        ObjectSystem -->
    <owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
        ObjectSystem">
        <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
```

```
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#aggregatesObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ObjectSystemModel -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ObjectSystemModel">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ModelType"/>
    <ROREKnowledgeModel:hasFactTypeAggregation rdf:resource="&
        LongTermMemory;phd.owl#OSMDomain"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Operation -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    Operation">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasAlgorithm"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasParameter"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasReturnValueType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Parameter -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    Parameter">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasValueType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    PrimaryState -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    PrimaryState">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#State"/>
```

```
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#containsInStructure"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#containsKeyObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Process -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Process"
    >
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasFirstStep"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ProcessModel -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ProcessModel">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#aggregatesProcess"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ProcessStep -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ProcessStep">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasNextStep"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#implementedBy"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Protocol -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Protocol
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#Interface"/>
```

```
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasMethod"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ProtocolMethod -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ProtocolMethod">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasParameter"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasPostcondition"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasPrecondition"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasReturnValueType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasTriggerCondition"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Relationship -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    Relationship">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasChildObject"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasParentObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    RelationshipGenerationRule -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    RelationshipGenerationRule">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ProductionRule"/
        >
```

```
    <ROREKnowledgeModel:hasConsequent rdf:datatype="&xsd;string"
        >ISMRelationships &lt; (VAR {ISMRelationships,
        LIST:ISMProperty}; VAR {Parent, ISMObject}; VAR {Child,
        ISMObject}) &lt; ([Relationship &lt;- RESULTSET*: ASSIGN{
        ISMRelationships, CREATE {$Relationship.name$Property,
        ISMProperty}}; ASSIGN {Parent, SELECT{target, &quot;
        PREFIX phd:&lt;http://www.rore.com/casestudies/&gt;
        SELECT phd:$Relationship.hasParent.name$&quot;}}; ASSIGN
        {Child, SELECT{target, &quot;PREFIX phd:&lt;http://www.
        rore.com/casestudies/&gt; SELECT phd:$Relationship.
        hasChild.name$&quot;}}; ASSIGN {$Relationship.name$
        Property.hasType, Child}; ASSIGN {Parent.hasProperty, $
        Relationship.name$Property}])</
        ROREKnowledgeModel:hasConsequent>
    <ROREKnowledgeModel:hasAntecedent rdf:datatype="&xsd;string"
        >PREFIX im: &lt;http://jamesnaish.wordpress.com/
        ROREKnowledgeModel.owl&gt; PREFIX ltm: &lt;http://
        jamesnaish.wordpress.com/LongTermMemory/phd.owl&gt;
        SELECT ?r WHERE { Type(?r, im:Fact), PropertyValue(?r,
        im:instantiatesFactType, ltm:Relationship) }</
        ROREKnowledgeModel:hasAntecedent>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    RemoteDataTransferProtocol -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    RemoteDataTransferProtocol">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Report -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Report">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#UserInterface"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasField"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#modelsObject"/>
```

```
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Requirements -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    Requirements">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Phase"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Resource -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Resource
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    ResourceModel -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    ResourceModel">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#aggregatesResource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    SecondaryState -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    SecondaryState">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#State"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasPropertyValue"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#stateOf"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#stateOverProperty"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    SenseAllPropertiesByDataSampleAlgorithm -->
```

```
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    SenseAllPropertiesByDataSampleAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;StringFact"/>
    <ROREKnowledgeModel:hasStringValue rdf:datatype="&xsd;string
        ">senseAllProperties(Objects[] os, Interface sensor) {
            foreach (Object o: os)     {          foreach (
        Property p: o.getProperties())     {
        senseProperty(p, sensor);    }      } }</
        ROREKnowledgeModel:hasStringValue>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    SensePropertyByDataSampleAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    SensePropertyByDataSampleAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;StringFact"/>
    <ROREKnowledgeModel:hasStringValue rdf:datatype="&xsd;string
        ">senseProperty(Property p, Interface sensor) {     p.set
        (sensor.readValue(p)); }</
        ROREKnowledgeModel:hasStringValue>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    SensorInterface -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    SensorInterface">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    State -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#State">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    StateTransition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    StateTransition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#conditionedBy"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#enactedBy"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#producesState"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#transformsStateOf"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#triggeredBy"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#triggersEndEvent"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#triggersStartEvent"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    StativeCondition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    StativeCondition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#hasConditionalState"/>
    <ROREKnowledgeModel:hasPropertyOfType rdf:resource="&
        LongTermMemory;phd.owl#isPrecondition"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    String -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#String">
    <rdf:type rdf:resource="&ROREKnowledgeModel;StringType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Structure -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    Structure">
```

```
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#Object"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    TransferAllCompleteObjectsAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    TransferAllCompleteObjectsAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;StringFact"/>
    <ROREKnowledgeModel:hasStringValue rdf:datatype="&xsd;string
        ">RemoteDataTransferProtocol.connect(HOST); for (
        TransferrableObject o: AllTransferrableObjects) {     o.
        transfer(RemoteDataTransferProtocol.instance, o.
        targetLocation); } RemoteDataTransferProtocol.instance.
        close();</ROREKnowledgeModel:hasStringValue>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    TransferCompleteObjectAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    TransferCompleteObjectAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;StringFact"/>
    <ROREKnowledgeModel:hasStringValue rdf:datatype="&xsd;string
        ">transferResource.transfer(this.location, targetLocation
        );</ROREKnowledgeModel:hasStringValue>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    UserInterface -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    UserInterface">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
    <ROREKnowledgeModel:subTypeOf rdf:resource="&LongTermMemory;
        phd.owl#Interface"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    Value -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#Value">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    aggregatesObject -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    aggregatesObject">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ISMObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    aggregatesObjectSystem -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    aggregatesObjectSystem">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ObjectSystem"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    aggregatesProcess -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    aggregatesProcess">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Process"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    aggregatesResource -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    aggregatesResource">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    allocateAllObjects -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    allocateAllObjects">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
    <ROREKnowledgeModel:hasProperty rdf:resource="&
        LongTermMemory;phd.owl#
        allocateAllObjectshasAlgorithmAllocateAllObjectsByClassificationAlgorithm
        "/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    allocateAllObjectshasAlgorithmAllocateAllObjectsByClassificationAlgorithm
    -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    allocateAllObjectshasAlgorithmAllocateAllObjectsByClassificationAlgorithm
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#
        AllocateAllObjectsByClassificationAlgorithm"/>
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasAlgorithm"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    allocateObject -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    allocateObject">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
    <ROREKnowledgeModel:hasProperty rdf:resource="&
        LongTermMemory;phd.owl#
        allocateObjecthasAlgorithmAllocateObjectByClassificationAlgorithm
        "/>
    <ROREKnowledgeModel:hasProperty rdf:resource="&
        LongTermMemory;phd.owl#
        allocateObjecthasParameterallocationRules"/>
```

```
    <ROREKnowledgeModel:hasProperty rdf:resource="&
        LongTermMemory;phd.owl#
        allocateObjecthasParameterpossibleStructures"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    allocateObjecthasAlgorithmAllocateObjectByClassificationAlgorithm
    -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    allocateObjecthasAlgorithmAllocateObjectByClassificationAlgorithm
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#
        AllocateObjectByClassificationAlgorithm"/>
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasAlgorithm"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    allocateObjecthasParameterallocationRules -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    allocateObjecthasParameterallocationRules">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#allocationRules"/>
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasParameter"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    allocateObjecthasParameterpossibleStructures -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    allocateObjecthasParameterpossibleStructures">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasParameter"/>
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#possibleStructures"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    allocationRules -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    allocationRules">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Parameter"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    conditionedBy -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    conditionedBy">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#StativeCondition"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    containsInStructure -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    containsInStructure">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Structure"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    containsKeyObject -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    containsKeyObject">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#KeyObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    detectedBy -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    detectedBy">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
```

```
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Object"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    dispatchAllNotifiedSampleCommands -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    dispatchAllNotifiedSampleCommands">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
    <ROREKnowledgeModel:hasProperty rdf:resource="&
        LongTermMemory;phd.owl#
        dispatchAllNotifiedSampleCommandshasAlgorithmdispatchAllNotifiedSampleC
        "/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    dispatchAllNotifiedSampleCommandsAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    dispatchAllNotifiedSampleCommandsAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;StringFact"/>
    <ROREKnowledgeModel:hasStringValue rdf:datatype="&xsd;string
        ">commandAllObjects(Objects[] os, Interface sensor) {
            Command c;      foreach (Object o: os)      {
        foreach (Property p: o.getProperties())        {
                    c = new Command(p, p.DataSource.getValue());
                commandObject(p, sensor);    }      } }</
        ROREKnowledgeModel:hasStringValue>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    dispatchAllNotifiedSampleCommandshasAlgorithmdispatchAllNotifiedSampleComm
    -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    dispatchAllNotifiedSampleCommandshasAlgorithmdispatchAllNotifiedSampleComm
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
```

```
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#
        dispatchAllNotifiedSampleCommandsAlgorithm"/>
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasAlgorithm"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    dispatchNotifiedSampleCommand -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    dispatchNotifiedSampleCommand">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
    <ROREKnowledgeModel:hasProperty rdf:resource="&
        LongTermMemory;phd.owl#
        dispatchNotifiedSampleCommandhasAlgorithmDispatchNotifiedSampleCommandAlgorit
        "/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    dispatchNotifiedSampleCommandhasAlgorithmDispatchNotifiedSampleCommandAlgorithm
    -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    dispatchNotifiedSampleCommandhasAlgorithmDispatchNotifiedSampleCommandAlgorithm
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#
        DispatchNotifiedSampleCommandAlgorithm"/>
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasAlgorithm"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    dispatchedBy -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    dispatchedBy">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Object"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    enactedBy -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    enactedBy">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Agent"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    fromState -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    fromState">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#State"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasAgentType -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasAgentType">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#AgentType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasAlgorithm -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasAttribute -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasAttribute">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
```

```
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#DatabaseAttribute"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasChildObject -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasChildObject">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Object"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasConditionalState -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasConditionalState">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#State"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasDataSink -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasDataSink">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasDataSource -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasDataSource">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Resource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasDomain -->
```

```
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasDomain">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Domain"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasDomainGoal -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasDomainGoal">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#DomainGoal"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasDomainType -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasDomainType">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#DomainType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasField -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#hasField
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasFirstStep -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasFirstStep">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ProcessStep"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasGenericProperty -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasGenericProperty">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ObjectProperty"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasGoalState -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasGoalState">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#GoalState"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasISMProperty -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasISMProperty">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ISMProperty"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasISMPropertyValue -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasISMPropertyValue">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Value"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasMethod -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasMethod">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
```

```
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ProtocolMethod"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasName -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#hasName"
    >
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasNextStep -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasNextStep">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ProcessStep"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasOperation -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasOperation">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasParameter -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasParameter">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Parameter"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasParentObject -->
```

```
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasParentObject">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Object"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasPostcondition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasPostcondition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasPrecondition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasPrecondition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasPrimaryState -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasPrimaryState">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#PrimaryState"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasProperty -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasProperty">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ObjectProperty"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasPropertyType -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasPropertyType">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasPropertyValue -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasPropertyValue">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ObjectPropertyValue"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasReturnValueType -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasReturnValueType">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasSecondaryState -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasSecondaryState">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#SecondaryState"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasStateTransition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasStateTransition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
```

```
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#StateTransition"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasStructure -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasStructure">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasSuperclass -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasSuperclass">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ISMObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasTable -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#hasTable
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#DatabaseTable"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasTriggerCondition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasTriggerCondition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    hasValueType -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    hasValueType">
```

```
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#String"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    implementedBy -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    implementedBy">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    isMonitored -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    isMonitored">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Boolean"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    isPrecondition -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    isPrecondition">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Boolean"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    modelledByState -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    modelledByState">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#State"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    modelsObject -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    modelsObject">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ISMObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    object -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#object">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Parameter"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    possibleStructures -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    possibleStructures">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Parameter"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    producesState -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    producesState">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#State"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    realisedBy -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    realisedBy">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#StateTransition"/>
```

```
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    satisfiedBy -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    satisfiedBy">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
    <ROREKnowledgeModel:hasProperty rdf:resource="&
        LongTermMemory;phd.owl#satisfiedByhasParameterobject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    satisfiedByhasParameterobject -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    satisfiedByhasParameterobject">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasParameter"/>
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#object"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    senseAllProperties -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    senseAllProperties">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
    <ROREKnowledgeModel:hasProperty rdf:resource="&
        LongTermMemory;phd.owl#
        senseAllPropertieshasAlgorithmSenseAllPropertiesByDataSampleAlgorithm
        "/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    senseAllPropertieshasAlgorithmSenseAllPropertiesByDataSampleAlgorithm
    -->
```

```
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    senseAllPropertieshasAlgorithmSenseAllPropertiesByDataSampleAlgorithm
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#
        SenseAllPropertiesByDataSampleAlgorithm"/>
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasAlgorithm"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    senseProperty -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    senseProperty">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
    <ROREKnowledgeModel:hasProperty rdf:resource="&
        LongTermMemory;phd.owl#
        sensePropertyhasAlgorithmSensePropertyByDataSampleAlgorithm
        "/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    sensePropertyhasAlgorithmSensePropertyByDataSampleAlgorithm
    -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    sensePropertyhasAlgorithmSensePropertyByDataSampleAlgorithm">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#SensePropertyByDataSampleAlgorithm
        "/>
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasAlgorithm"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    stateOf -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#stateOf"
    >
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
```

```
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Object"/>
</owl:NamedIndividual>

<!— http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    stateOverProperty —>
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    stateOverProperty">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ObjectProperty"/>
</owl:NamedIndividual>

<!— http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    targetLocation —>
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    targetLocation">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Parameter"/>
</owl:NamedIndividual>

<!— http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    transferAllCompleteObjects —>
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    transferAllCompleteObjects">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Operation"/>
    <ROREKnowledgeModel:hasProperty rdf:resource="&
        LongTermMemory;phd.owl#
        transferAllCompleteObjectshasAlgorithmTransferAllCompleteObjectsAlgori
        "/>
</owl:NamedIndividual>

<!— http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    transferAllCompleteObjectshasAlgorithmTransferAllCompleteObjectsAlgorithm
    —>
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    transferAllCompleteObjectshasAlgorithmTransferAllCompleteObjectsAlgorithm
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
```

```
        <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
            LongTermMemory;phd.owl#
            TransferAllCompleteObjectsAlgorithm"/>
        <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
            LongTermMemory;phd.owl#hasAlgorithm"/>
    </owl:NamedIndividual>

    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        transferCompleteObject -->
    <owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
        transferCompleteObject">
        <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
        <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
            LongTermMemory;phd.owl#Operation"/>
        <ROREKnowledgeModel:hasProperty rdf:resource="&
            LongTermMemory;phd.owl#
            transferCompleteObjecthasAlgorithmTransferCompleteObjectAlgorithm
            "/>
        <ROREKnowledgeModel:hasProperty rdf:resource="&
            LongTermMemory;phd.owl#
            transferCompleteObjecthasParametertargetLocation"/>
        <ROREKnowledgeModel:hasProperty rdf:resource="&
            LongTermMemory;phd.owl#
            transferCompleteObjecthasParametertransferInterfaceResource
            "/>
    </owl:NamedIndividual>

    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        transferCompleteObjecthasAlgorithmTransferCompleteObjectAlgorithm
        -->
    <owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
        transferCompleteObjecthasAlgorithmTransferCompleteObjectAlgorithm
        ">
        <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
        <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
            LongTermMemory;phd.owl#TransferCompleteObjectAlgorithm"/>
        <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
            LongTermMemory;phd.owl#hasAlgorithm"/>
    </owl:NamedIndividual>

    <!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
        transferCompleteObjecthasParametertargetLocation -->
```

```
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    transferCompleteObjecthasParametertargetLocation">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasParameter"/>
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#targetLocation"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    transferCompleteObjecthasParametertransferInterfaceResource
    -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    transferCompleteObjecthasParametertransferInterfaceResource">
    <rdf:type rdf:resource="&ROREKnowledgeModel;Property"/>
    <ROREKnowledgeModel:instantiatesPropertyType rdf:resource="&
        LongTermMemory;phd.owl#hasParameter"/>
    <ROREKnowledgeModel:hasPropertyValue rdf:resource="&
        LongTermMemory;phd.owl#transferInterfaceResource"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    transferInterfaceResource -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    transferInterfaceResource">
    <rdf:type rdf:resource="&ROREKnowledgeModel;ComplexFact"/>
    <ROREKnowledgeModel:instantiatesFactType rdf:resource="&
        LongTermMemory;phd.owl#Parameter"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    transformsStateOf -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    transformsStateOf">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#KeyObject"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    triggeredBy -->
```

```
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    triggeredBy">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Event"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    triggersEndEvent -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    triggersEndEvent">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Event"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    triggersStartEvent -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#
    triggersStartEvent">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Event"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    valueFor -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#valueFor
    ">
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Object"/>
</owl:NamedIndividual>

<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    valueIs -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#valueIs"
    >
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#Value"/>
</owl:NamedIndividual>
```

```
<!-- http://jamesnaish.wordpress.com/LongTermMemory/phd.owl#
    valueOf -->
<owl:NamedIndividual rdf:about="&LongTermMemory;phd.owl#valueOf"
    >
    <rdf:type rdf:resource="&ROREKnowledgeModel;PropertyType"/>
    <ROREKnowledgeModel:hasPropertyValueType rdf:resource="&
        LongTermMemory;phd.owl#ObjectProperty"/>
</owl:NamedIndividual>

<!--
///////////////////////////////////////////////////////////////////////////////

//
// Rules
//
///////////////////////////////////////////////////////////////////////////////

 -->
<rdf:Description rdf:about="urn:swrl#child">
    <rdf:type rdf:resource="&swrl;Variable"/>
</rdf:Description>
<rdf:Description rdf:about="urn:swrl#parent">
    <rdf:type rdf:resource="&swrl;Variable"/>
</rdf:Description>
<rdf:Description rdf:about="urn:swrl#f1">
    <rdf:type rdf:resource="&swrl;Variable"/>
</rdf:Description>
<rdf:Description rdf:about="urn:swrl#f2">
    <rdf:type rdf:resource="&swrl;Variable"/>
</rdf:Description>
<rdf:Description>
    <rdf:type rdf:resource="&swrl;Imp"/>
    <swrl:head>
        <rdf:Description>
            <rdf:type rdf:resource="&swrl;AtomList"/>
            <rdf:rest rdf:resource="&rdf;nil"/>
            <rdf:first>
                <rdf:Description>
                    <rdf:type rdf:resource="&swrl;
                        IndividualPropertyAtom"/>
```

```
                <swrl:propertyPredicate rdf:resource="&
                    ROREKnowledgeModel;instantiatesFactType"/
                    >
                <swrl:argument1 rdf:resource="urn:swrl#child
                    "/>
                <swrl:argument2 rdf:resource="urn:swrl#f2"/>
            </rdf:Description>
        </rdf:first>
    </rdf:Description>
</swrl:head>
<swrl:body>
    <rdf:Description>
        <rdf:type rdf:resource="&swrl;AtomList"/>
        <rdf:rest>
            <rdf:Description>
                <rdf:type rdf:resource="&swrl;AtomList"/>
                <rdf:first>
                    <rdf:Description>
                        <rdf:type rdf:resource="&swrl;
                            ClassAtom"/>
                        <swrl:classPredicate rdf:resource="&
                            ROREKnowledgeModel;Fact"/>
                        <swrl:argument1 rdf:resource="
                            urn:swrl#parent"/>
                    </rdf:Description>
                </rdf:first>
                <rdf:rest>
                    <rdf:Description>
                        <rdf:type rdf:resource="&swrl;
                            AtomList"/>
                        <rdf:rest>
                            <rdf:Description>
                                <rdf:type rdf:resource="&
                                    swrl;AtomList"/>
                                <rdf:first>
                                    <rdf:Description>
                                        <rdf:type
                                            rdf:resource="&
                                            swrl;
                                            IndividualPropertyAtom
                                            "/>
```

```
                <
                    swrl:propertyPredicate
                     rdf:resource="&
                    ROREKnowledgeModel
                    ;
                    instantiatesFactType
                    "/>
                <swrl:argument2
                    rdf:resource="
                    urn:swrl#f2"/>
                <swrl:argument1
                    rdf:resource="
                    urn:swrl#parent"/
                   >
            </rdf:Description>
        </rdf:first>
        <rdf:rest>
            <rdf:Description>
                <rdf:type
                    rdf:resource="&
                    swrl;AtomList"/>
                <rdf:rest
                    rdf:resource="&
                    rdf;nil"/>
                <rdf:first>
                    <rdf:Description
                        >
                        <rdf:type
                            rdf:resource
                            ="&swrl;
                            IndividualPropertyAton
                            "/>
                        <
                            swrl:propertyPredicat

                            rdf:resource
                            ="&
                            ROREKnowledgeModel
                            ;
                            subTypeOf
                            "/>
```

```
                                                <
                                                    swrl:argument1

                                                    rdf:resource
                                                    =”
                                                    urn:swrl#
                                                    f1”/>
                                                <
                                                    swrl:argument2

                                                    rdf:resource
                                                    =”
                                                    urn:swrl#
                                                    f2”/>
                                            </
                                                rdf:Description
                                                >
                                        </rdf:first>
                                    </rdf:Description>
                                </rdf:rest>
                            </rdf:Description>
                        </rdf:rest>
                        <rdf:first>
                            <rdf:Description>
                                <rdf:type rdf:resource=”&
                                    swrl;
                                    IndividualPropertyAtom”/>
                                <swrl:propertyPredicate
                                    rdf:resource=”&
                                    ROREKnowledgeModel;
                                    instantiatesFactType”/>
                                <swrl:argument1 rdf:resource
                                    =”urn:swrl#child”/>
                                <swrl:argument2 rdf:resource
                                    =”urn:swrl#f1”/>
                            </rdf:Description>
                        </rdf:first>
                    </rdf:Description>
                </rdf:rest>
            </rdf:Description>
        </rdf:rest>
        <rdf:first>
```

```
                    <rdf:Description>
                        <rdf:type rdf:resource="&swrl;ClassAtom"/>
                        <swrl:classPredicate rdf:resource="&
                            ROREKnowledgeModel;Fact"/>
                        <swrl:argument1 rdf:resource="urn:swrl#child
                            "/>
                    </rdf:Description>
                </rdf:first>
            </rdf:Description>
        </swrl:body>
    </rdf:Description>
</rdf:RDF>
<!-- Generated by the OWL API (version 3.3.1957) http://owlapi.
    sourceforge.net -->
```

Appendix C

Autopilot Example Log

1. Cycle 1

1.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "false". Overall conclusion is: Model is not complete

1.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "false".

New Information Requirement Generated

Goal Fact Type: ISMObject

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#> PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#>  ASK { Individual(?o), Type(?o, im:Fact),
PropertyValue(?o, im:instantiatesFactType,
ltm:ISMObject) }
```

1.3. Attempted Chunk-based Inference

Inference Chunks Found

1.4. Attempted Rule-based Inference

Production Scripts Found

Attempting to fire the Production Script "HasISMObjectsRule".

Source Ont is: http://jamesnaish.wordpress.com/Model/AutopilotSourceModel.owl

Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP0480468299.owl

```
Object <- RESULTSET [
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
SpecObject.hasISMProperty=CREATE ($Property.name$,
ISMProperty)
: Success
$Property.name$.hasPropertyType=Property.hasValueType
: Success
SpecObject.hasISMProperty=CREATE ($Property.name$,
ISMProperty)
: Success
$Property.name$.hasPropertyType=Property.hasValueType
: Success
```

```
SpecObject.hasISMProperty=CREATE ($Property.name$,
ISMProperty)
: Success
$Property.name$.hasPropertyType=Property.hasValueType
: Success
SpecObject.hasISMProperty=CREATE ($Property.name$,
ISMProperty)
: Success
$Property.name$.hasPropertyType=Property.hasValueType
: Success
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
SpecObject.hasISMProperty=CREATE ($Property.name$,
ISMProperty)
: Success
$Property.name$.hasPropertyType=Property.hasValueType
: Success
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
```

```
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
SpecObject.hasISMProperty=CREATE ($Property.name$,
ISMProperty)
: Success
$Property.name$.hasPropertyType=Property.hasValueType
: Success
]: Success
ISMObjects=SpecObject
: Success
]: Success
```

The production script was fired successfully, producing the following facts:

```
PlaneClass
XGyroClass
SkyClass
XControllerClass
YControllerClass
YGyroClass
```

1.5. Attempted Integration

Integration was attempted using the "Additive" strategy.

Integration was successful.

2. Cycle 2

2.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "false". Overall conclusion is: Model is not complete

2.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasMonitorReportRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Report

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?r), Type(?d, im:Fact),
PropertyValue(?d, im:instantiatesFactType, ltm:Report)
}
```

2.3. Attempted Rule-based Inference

Production Scripts Found

Attempting to fire the Production Script "HasMonitorReportRule".

Source Ont is: http://jamesnaish.wordpress.com/Model/AutopilotSourceModel.owl

Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP0864148981.owl

```
CKeyObject <- RESULTSET [
CListReport=CREATE ($CKeyObject.name$ListReport,
Report)
: Success
CListReport.hasField=CREATE (has$CKeyObject.name$s,
String)
: Success
Reports=CListReport
: Success
CReport=CREATE ($CKeyObject.name$Report, Report)
: Success
CReport.hasField=CREATE ($CKeyObject.name$Name,
String)
: Success
```

```
CProperty <- CKeyObject.ALL:hasProperty [
CReport.hasField=CREATE
($CKeyObject.name$$CProperty.name$, String)
: Success
CReport.hasField=CREATE
($CKeyObject.name$$CProperty.name$, String)
: Success
CReport.hasField=CREATE
($CKeyObject.name$$CProperty.name$, String)
: Success
CReport.hasField=CREATE
($CKeyObject.name$$CProperty.name$, String)
: Success
]: Success
Reports=CReport
: Success
]: Success
```

The production script was fired successfully, producing the following facts:

```
PlaneListReport
PlaneReport
```

2.4. Attempted Integration

Integration was attempted using the "Additive" strategy.

Integration was successful.

3. Cycle 4

3.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "false". Overall conclusion is: Model is not complete

3.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Operation

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?r), Type(?d, im:Fact),
PropertyValue(?d, im:instantiatesFactType,
ltm:Operation)  }
```

3.3. Attempted Chunk-based Inference

Inference Chunks Found

3.4. Attempted Rule-based Inference

Production Scripts Found

Attempting to fire the Production Script "HasFunctionalResponsibilitiesRule".

Source Ont is: http://jamesnaish.wordpress.com/Model/AutopilotSourceModel.owl

Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP1672075311.owl

```
CDomain <- RESULTSET [
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Control? [
COperation=CREATE (issueCommandTo$KeyOSMObject.name$,
```

```
Operation)
: Success
Parameters=LIST:Parameter
: Success
CParameter=CREATE ($KeyOSMObject.name$Command,
Parameter)
: Success
Parameters=CParameter
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
]: Success
]: Success
KeyISMObjects=KeyISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Sensing? [
COperation=CREATE (sensePropertyOf$KeyOSMObject.name$,
Operation)
: Success
Parameters=LIST:Parameter
: Success
CParameter=CREATE ($KeyOSMObject.name$Property,
Parameter)
: Success
Parameters=CParameter
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
]: Success
```

```
]: Success
KeyISMObjects=KeyISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Sensing? [
COperation=CREATE (sensePropertyOf$KeyOSMObject.name$,
Operation)
: Success
Parameters=LIST:Parameter
: Success
CParameter=CREATE ($KeyOSMObject.name$Property,
Parameter)
: Success
Parameters=CParameter
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
]: Success
]: Success
KeyISMObjects=KeyISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Control? [
COperation=CREATE (issueCommandTo$KeyOSMObject.name$,
Operation)
: Success
```

```
Parameters=LIST:Parameter
: Success
CParameter=CREATE ($KeyOSMObject.name$Command,
Parameter)
: Success
Parameters=CParameter
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
]: Success
]: Success
KeyISMObjects=KeyISMObject
: Success
]: Success
```

Source Ont is: http://jamesnaish.wordpress.com/Model/AutopilotSourceModel.owl

Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP1672075311.owl

```
CDomain <- RESULTSET [
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Control? [
COperation=CREATE (command$KeyOSMObject.name$Objects,
Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
]: Success
AgentISMObjects=AgentISMObject
: Success
```

```
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Sensing? [
COperation=CREATE (sense$KeyOSMObject.name$Objects,
Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
]: Success
AgentISMObjects=AgentISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Sensing? [
COperation=CREATE (sense$KeyOSMObject.name$Objects,
Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
]: Success
```

```
AgentISMObjects=AgentISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Control? [
COperation=CREATE (command$KeyOSMObject.name$Objects,
Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
]: Success
AgentISMObjects=AgentISMObject
: Success
]: Success
```

The production script was fired successfully, producing the following facts:

```
PlaneClass
PlaneClass
PlaneClass
PlaneClass
YControllerClass
XGyroClass
YGyroClass
XControllerClass
```

3.5. Attempted Integration

Pairing Facts "PlaneClass" and "PlaneClass".

Pairing Facts "PlaneClass" and "PlaneClass".

Pairing Facts "PlaneClass" and "PlaneClass".

Pairing Facts "PlaneClass" and "PlaneClass".

Pairing Facts "XControllerClass" and "XControllerClass".

Pairing Facts "XGyroClass" and "XGyroClass".

Pairing Facts "YControllerClass" and "YControllerClass".
Pairing Facts "YGyroClass" and "YGyroClass".
Integration was attempted using the "Additive" strategy.
Integration was successful.

4. Cycle 6

4.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true". Overall conclusion is: Model is not complete

4.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true".

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true".

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true".

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

4.3. Attempted Chunk-based Inference

Inference Chunks Found

4.4. Attempted Integration

Pairing Facts "senseAllProperties" and "sensePlaneObjects".

Pairing Facts "senseProperty" and "sensePropertyOfPlane".

Integration was attempted using the "Additive" strategy.

Integration was successful.

5. Cycle 8

5.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

5.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "true".

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "true".

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

5.3. Attempted Integration

Pairing Facts "dispatchAllNotifiedSampleCommands" and "commandPlaneObjects".

Pairing Facts "dispatchNotifiedSampleCommand" and "issueCommandToPlane".

Integration was attempted using the "Additive" strategy.

Integration was successful.

6. Cycle 10

6.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

6.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

6.3. Attempted Chunk-based Inference

Inference Chunks Found

6.4. Attempted Rule-based Inference

Production Scripts Found

6.5. Attempted Elicitation

Elicitation Stimuli Found

Attempting to fire the Elicitation Stimulus "FunctionalResourceConfigurationStimulus", a fact-editing stimulus

The elicitation stimulus was fired successfully, producing the following facts:

```
SensorInterface
```

```
ControlInterface
```

6.6. Attempted Integration

Pairing Facts "ControlInterface" and "ControlInterface".

Pairing Facts "SensorInterface" and "SensorInterface".

Integration was attempted using the "Additive" strategy.

Integration was successful.

7. Cycle 12

7.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

7.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

7.3. Attempted Elicitation

Elicitation Stimuli Found

7.4. Attempted Elicitation

Elicitation Stimuli Found

Attempting to fire the Elicitation Stimulus "DataSourceAndSinkStimulus", a fact-editing stimulus

The elicitation stimulus was fired successfully, producing the following facts:

```
NotifiedSampleCommand
SkyClass
PlaneClass
```

```
XControllerClass
XGyroClass
YControllerClass
YGyroClass
```

7.5. Attempted Integration

Pairing Facts "NotifiedSampleCommand" and "NotifiedSampleCommand".

Pairing Facts "PlaneClass" and "PlaneClass".

Pairing Facts "SkyClass" and "SkyClass".

Pairing Facts "XControllerClass" and "XControllerClass".

Pairing Facts "XGyroClass" and "XGyroClass".

Pairing Facts "YControllerClass" and "YControllerClass".

Pairing Facts "YGyroClass" and "YGyroClass".

Integration was attempted using the "Additive" strategy.

Integration was successful.

8. Cycle 14

8.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "DataSinksSpecifiedRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "true". Overall conclusion is: Model is complete thus far

Appendix D

File Transfer Example Log

1. Cycle 1

1.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "false". Overall conclusion is: Model is not complete

1.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "false".

New Information Requirement Generated

Goal Fact Type: ISMObject

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#> PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#>  ASK { Individual(?o), Type(?o, im:Fact),
PropertyValue(?o, im:instantiatesFactType,
ltm:ISMObject) }
```

1.3. Attempted Chunk-based Inference

Inference Chunks Found

1.4. Attempted Rule-based Inference

Production Scripts Found

Attempting to fire the Production Script "HasISMObjectsRule".

Source Ont is: http://jamesnaish.wordpress.com/Model/FileTransferSourceModel.owl

Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP0604616089.owl

```
Object <- RESULTSET [
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
```

```
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
]: Success
```

The production script was fired successfully, producing the following facts:

```
DownloadFileClass
RemoteDirectoryClass
UploadFileClass
FileUploaderClass
FileDownloaderClass
LocalDirectoryClass
```

1.5. Attempted Integration

Integration was attempted using the "Additive" strategy.

Integration was successful.

2. Cycle 2

2.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "false". Overall conclusion is: Model is not complete

2.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasMonitorReportRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Report

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?r), Type(?d, im:Fact),
PropertyValue(?d, im:instantiatesFactType, ltm:Report)
}
```

2.3. Attempted Chunk-based Inference

Inference Chunks Found

2.4. Attempted Rule-based Inference

Production Scripts Found

Attempting to fire the Production Script "HasMonitorReportRule".

Source Ont is: http://jamesnaish.wordpress.com/Model/FileTransferSourceModel.owl

Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP1301040187.owl

```
CKeyObject <- RESULTSET [
]: Success
```

The production script was fired successfully, producing the following facts:

2.5. Attempted Integration

Integration was attempted using the "Additive" strategy.

Integration was successful.

3. Cycle 4

3.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "false". Overall conclusion is: Model is not complete

3.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Operation

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?r), Type(?d, im:Fact),
PropertyValue(?d, im:instantiatesFactType,
ltm:Operation)  }
```

3.3. Attempted Rule-based Inference

Production Scripts Found

Attempting to fire the Production Script "HasFunctionalResponsibilitiesRule".

Source Ont is: http://jamesnaish.wordpress.com/Model/FileTransferSourceModel.owl

Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP2045294436.owl

```
CDomain <- RESULTSET [
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Allocation? [
COperation=CREATE (allocateTo$KeyOSMObject.name$,
Operation)
: Success
Parameters=LIST:Parameter
: Success
```

```
CParameter=CREATE ($KeyOSMObject.name$TargetStructure,
Parameter)
: Success
CParameter.hasValueType=SELECT (SELECT ?t WHERE {
SameAs(?t,
wm:$CDomain.hasDomainGoal.hasGoalState.realisedBy.prod
ucesState.containsInStructure.name$Class) })
: Success
Parameters=CParameter
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
]: Success
]: Success
KeyISMObjects=KeyISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Allocation? [
COperation=CREATE (allocateTo$KeyOSMObject.name$,
Operation)
: Success
Parameters=LIST:Parameter
: Success
CParameter=CREATE ($KeyOSMObject.name$TargetStructure,
Parameter)
: Success
CParameter.hasValueType=SELECT (SELECT ?t WHERE {
SameAs(?t,
wm:$CDomain.hasDomainGoal.hasGoalState.realisedBy.prod
ucesState.containsInStructure.name$Class) })
: Success
Parameters=CParameter
```

```
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
]: Success
]: Success
KeyISMObjects=KeyISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Transfer? [
COperation=CREATE (transfer$KeyOSMObject.name$To,
Operation)
: Success
Parameters=LIST:Parameter
: Success
CParameter=CREATE ($KeyOSMObject.name$TargetStructure,
Parameter)
: Success
CParameter.hasValueType=SELECT (SELECT ?t WHERE {
SameAs(?t,
wm:$CDomain.hasDomainGoal.hasGoalState.realisedBy.prod
ucesState.containsInStructure.name$Class) })
: Success
Parameters=CParameter
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
]: Success
]: Success
KeyISMObjects=KeyISMObject
: Success
```

```
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Transfer? [
COperation=CREATE (transfer$KeyOSMObject.name$To,
Operation)
: Success
Parameters=LIST:Parameter
: Success
CParameter=CREATE ($KeyOSMObject.name$TargetStructure,
Parameter)
: Success
CParameter.hasValueType=SELECT (SELECT ?t WHERE {
SameAs(?t,
wm:$CDomain.hasDomainGoal.hasGoalState.realisedBy.prod
ucesState.containsInStructure.name$Class) })
: Success
Parameters=CParameter
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
]: Success
]: Success
KeyISMObjects=KeyISMObject
: Success
]: Success
```

Source Ont is: http://jamesnaish.wordpress.com/Model/FileTransferSourceModel.owl

Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP2045294436.owl

```
CDomain <- RESULTSET [
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
```

```
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Allocation? [
COperation=CREATE (allocate$KeyOSMObject.name$Objects,
Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
]: Success
AgentISMObjects=AgentISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Allocation? [
COperation=CREATE (allocate$KeyOSMObject.name$Objects,
Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
]: Success
AgentISMObjects=AgentISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
```

```
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Transfer? [
COperation=CREATE (transfer$KeyOSMObject.name$Objects,
Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
]: Success
AgentISMObjects=AgentISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Transfer? [
COperation=CREATE (transfer$KeyOSMObject.name$Objects,
Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
]: Success
AgentISMObjects=AgentISMObject
: Success
]: Success
```

The production script was fired successfully, producing the following facts:

```
UploadFileClass
DownloadFileClass
UploadFileClass
DownloadFileClass
FileUploaderClass
FileDownloaderClass
FileUploaderClass
FileDownloaderClass
```

3.4. Attempted Integration

Pairing Facts "DownloadFileClass" and "DownloadFileClass".
Pairing Facts "DownloadFileClass" and "DownloadFileClass".
Pairing Facts "FileDownloaderClass" and "FileDownloaderClass".
Pairing Facts "FileDownloaderClass" and "FileDownloaderClass".
Pairing Facts "FileUploaderClass" and "FileUploaderClass".
Pairing Facts "FileUploaderClass" and "FileUploaderClass".
Pairing Facts "UploadFileClass" and "UploadFileClass".
Pairing Facts "UploadFileClass" and "UploadFileClass".
Integration was attempted using the "Additive" strategy.
Integration was successful.

4. Cycle 6

4.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true". Overall conclusion is: Model is not complete

4.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

4.3. Attempted Chunk-based Inference

Inference Chunks Found

4.4. Attempted Integration

Pairing Facts "allocateAllObjects" and "allocateDownloadFileObjects".

Pairing Facts "allocateAllObjects" and "allocateUploadFileObjects".

Pairing Facts "allocateObject" and "allocateToDownloadFile".

Pairing Facts "allocateObject" and "allocateToUploadFile".

Integration was attempted using the "Additive" strategy.

Integration was successful.

5. Cycle 8

5.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

5.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

5.3. Attempted Chunk-based Inference

Inference Chunks Found

5.4. Attempted Integration

Pairing Facts "transferAllCompleteObjects" and "transferDownloadFileObjects".

Pairing Facts "transferAllCompleteObjects" and "transferUploadFileObjects".

Pairing Facts "transferCompleteObject" and "transferDownloadFileTo".

Pairing Facts "transferCompleteObject" and "transferUploadFileTo".

Integration was attempted using the "Additive" strategy.

Integration was successful.

6. Cycle 10

6.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

6.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

6.3. Attempted Chunk-based Inference

Inference Chunks Found

6.4. Attempted Rule-based Inference

Production Scripts Found

6.5. Attempted Elicitation

Elicitation Stimuli Found

Attempting to fire the Elicitation Stimulus "FunctionalResourceConfigurationStimulus", a fact-editing stimulus

The elicitation stimulus was fired successfully, producing the following facts:

```
RemoteDataTransferProtocol
ClassificationRuleSource
```

6.6. Attempted Integration

Pairing Facts "ClassificationRuleSource" and "ClassificationRuleSource".

Pairing Facts "RemoteDataTransferProtocol" and "RemoteDataTransferProtocol".

Integration was attempted using the "Additive" strategy.

Integration was successful.

7. Cycle 12

7.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

7.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

7.3. Attempted Chunk-based Inference

Inference Chunks Found

7.4. Attempted Rule-based Inference

Production Scripts Found

7.5. Attempted Elicitation

Elicitation Stimuli Found

Attempting to fire the Elicitation Stimulus "DataSourceAndSinkStimulus", a fact-editing stimulus

The elicitation stimulus was fired successfully, producing the following facts:

```
AllocationRule
FileUploaderClass
DownloadFileClass
RemoteDirectoryClass
LocalDirectoryClass
FileDownloaderClass
UploadFileClass
```

7.6. Attempted Integration

Pairing Facts "AllocationRule" and "AllocationRule".
Pairing Facts "DownloadFileClass" and "DownloadFileClass".
Pairing Facts "FileDownloaderClass" and "FileDownloaderClass".
Pairing Facts "FileUploaderClass" and "FileUploaderClass".
Pairing Facts "LocalDirectoryClass" and "LocalDirectoryClass".
Pairing Facts "RemoteDirectoryClass" and "RemoteDirectoryClass".
Pairing Facts "UploadFileClass" and "UploadFileClass".
Integration was attempted using the "Additive" strategy.
Integration was successful.

8. Cycle 14

8.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "false". Overall conclusion is: Model is not complete

Fired Analysis Rule "DataSinksSpecifiedRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Appendix E

Order Management Example Log

1. Cycle 1

1.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "false". Overall conclusion is: Model is not complete

1.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "false".

New Information Requirement Generated

Goal Fact Type: ISMObject

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#> PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#>  ASK { Individual(?o), Type(?o, im:Fact),
PropertyValue(?o, im:instantiatesFactType,
ltm:ISMObject) }
```

1.3. Attempted Chunk-based Inference

Inference Chunks Found

1.4. Attempted Rule-based Inference

Production Scripts Found

Attempting to fire the Production Script "HasISMObjectsRule".

Source Ont is: http://jamesnaish.wordpress.com/Model/OrderManagementSourceModel.owl

Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP0390117878.owl

```
Object <- RESULTSET [
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
```

```
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
SpecObject.hasISMProperty=CREATE ($Property.name$,
ISMProperty)
: Success
$Property.name$.hasPropertyType=Property.hasValueType
: Success
SpecObject.hasISMProperty=CREATE ($Property.name$,
ISMProperty)
: Success
$Property.name$.hasPropertyType=Property.hasValueType
: Success
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
```

```
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
```

```
: Success
SpecObject=CREATE ($Object.name$Class, ISMObject)
: Success
Property <- Object.ALL:hasProperty [
]: Success
ISMObjects=SpecObject
: Success
]: Success
```

The production script was fired successfully, producing the following facts:

```
TCardPoolClass
DispatchAreaClass
ProductionLineOperativeClass
OrderDownloaderClass
TCardClass
ProductionLineClass
OrderServerClass
OrderClass
OrderScheduleClass
TCardPrinterClass
DeliveryVanClass
OrderSchedulerClass
OrderManagementSystemClass
OrderPoolClass
DrillClass
```

1.5. Attempted Integration

Integration was attempted using the "Additive" strategy.

Integration was successful.

2. Cycle 2

2.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "false". Overall conclusion is: Model is not complete

2.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasMonitorReportRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Report

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?r), Type(?d, im:Fact),
PropertyValue(?d, im:instantiatesFactType, ltm:Report)
}
```

2.3. Attempted Chunk-based Inference

Inference Chunks Found

2.4. Attempted Rule-based Inference

Production Scripts Found

Attempting to fire the Production Script "HasMonitorReportRule".

Source Ont is: http://jamesnaish.wordpress.com/Model/OrderManagementSourceModel.owl

Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP1819903711.owl

```
CKeyObject <- RESULTSET [
CListReport=CREATE ($CKeyObject.name$ListReport,
Report)
: Success
CListReport.hasField=CREATE (has$CKeyObject.name$s,
String)
: Success
Reports=CListReport
: Success
CReport=CREATE ($CKeyObject.name$Report, Report)
: Success
CReport.hasField=CREATE ($CKeyObject.name$Name,
```

```
String)
: Success
CProperty <- CKeyObject.ALL:hasProperty [
]: Success
Reports=CReport
: Success
CListReport=CREATE ($CKeyObject.name$ListReport,
Report)
: Success
CListReport.hasField=CREATE (has$CKeyObject.name$s,
String)
: Success
Reports=CListReport
: Success
CReport=CREATE ($CKeyObject.name$Report, Report)
: Success
CReport.hasField=CREATE ($CKeyObject.name$Name,
String)
: Success
CProperty <- CKeyObject.ALL:hasProperty [
]: Success
Reports=CReport
: Success
]: Success
```

The production script was fired successfully, producing the following facts:

```
OrderListReport
OrderReport
DrillListReport
DrillReport
```

2.5. Attempted Integration

Integration was attempted using the "Additive" strategy.

Integration was successful.

3. Cycle 4

3.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "false". Overall conclusion is: Model is not complete

3.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Operation

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?r), Type(?d, im:Fact),
PropertyValue(?d, im:instantiatesFactType,
ltm:Operation)  }
```

3.3. Attempted Chunk-based Inference

Inference Chunks Found

3.4. Attempted Rule-based Inference

Production Scripts Found

Attempting to fire the Production Script "HasFunctionalResponsibilitiesRule".

Source Ont is: http://jamesnaish.wordpress.com/Model/OrderManagementSourceModel.owl

Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP1239670094.owl

```
CDomain <- RESULTSET [
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Transfer? [
COperation=CREATE (transfer$KeyOSMObject.name$To,
```

```
Operation)
: Success
Parameters=LIST:Parameter
: Success
CParameter=CREATE ($KeyOSMObject.name$TargetStructure,
Parameter)
: Success
CParameter.hasValueType=SELECT (SELECT ?t WHERE {
SameAs(?t,
wm:$CDomain.hasDomainGoal.hasGoalState.realisedBy.prod
ucesState.containsInStructure.name$Class) })
: Success
Parameters=CParameter
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
]: Success
]: Success
KeyISMObjects=KeyISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Allocation? [
COperation=CREATE (allocateTo$KeyOSMObject.name$,
Operation)
: Success
Parameters=LIST:Parameter
: Success
CParameter=CREATE ($KeyOSMObject.name$TargetStructure,
Parameter)
: Success
CParameter.hasValueType=SELECT (SELECT ?t WHERE {
SameAs(?t,
```

```
wm:$CDomain.hasDomainGoal.hasGoalState.realisedBy.prod
ucesState.containsInStructure.name$Class) })
: Success
Parameters=CParameter
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
]: Success
]: Success
KeyISMObjects=KeyISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Transfer? [
COperation=CREATE (transfer$KeyOSMObject.name$To,
Operation)
: Success
Parameters=LIST:Parameter
: Success
CParameter=CREATE ($KeyOSMObject.name$TargetStructure,
Parameter)
: Success
CParameter.hasValueType=SELECT (SELECT ?t WHERE {
SameAs(?t,
wm:$CDomain.hasDomainGoal.hasGoalState.realisedBy.prod
ucesState.containsInStructure.name$Class) })
: Success
Parameters=CParameter
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
```

```
]: Success
]: Success
KeyISMObjects=KeyISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
KeyISMObject=SELECT (SELECT ?o WHERE { Individual(?o),
SameAs(?o, tm:$KeyOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Manipulation? [
COperation=CREATE
(manipulatePropertyOf$KeyOSMObject.name$, Operation)
: Success
Parameters=LIST:Parameter
: Success
CParameter=CREATE ($KeyOSMObject.name$property,
Parameter)
: Success
Parameters=CParameter
: Success
CParameter=CREATE ($KeyOSMObject.name$PropertyValue,
Parameter)
: Success
Parameters=CParameter
: Success
COperation.hasParameter=Parameters
: Success
KeyISMObject.hasOperation=COperation
: Success
]: Success
]: Success
KeyISMObjects=KeyISMObject
: Success
]: Success
```

Source Ont is: http://jamesnaish.wordpress.com/Model/OrderManagementSourceModel.owl
Temp Ont is: http://jamesnaish.wordpress.com/Temporary/TEMP1239670094.owl

```
CDomain <- RESULTSET [
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
```

```
edBy.transformsStateOf
: Success
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Transfer? [
COperation=CREATE (transfer$KeyOSMObject.name$Objects,
Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
]: Success
AgentISMObjects=AgentISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Allocation? [
COperation=CREATE (allocate$KeyOSMObject.name$Objects,
Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
]: Success
AgentISMObjects=AgentISMObject
```

```
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Transfer? [
COperation=CREATE (transfer$KeyOSMObject.name$Objects,
Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
]: Success
AgentISMObjects=AgentISMObject
: Success
KeyOSMObject=CDomain.hasDomainGoal.hasGoalState.realis
edBy.transformsStateOf
: Success
AgentOSMObject=CDomain.hasDomainGoal.hasGoalState.real
isedBy.enactedBy
: Success
AgentISMObject=SELECT (SELECT ?o WHERE {
Individual(?o), SameAs(?o,
tm:$AgentOSMObject.name$Class) })
: Success
FIRING CONDITIONAL:[
CDomain.hasDomainGoal.hasName is Manipulation? [
COperation=CREATE
(manipulate$KeyOSMObject.name$Objects, Operation)
: Success
AgentISMObject.hasOperation=COperation
: Success
]: Success
```

```
]: Success
AgentISMObjects=AgentISMObject
: Success
]: Success
```

The production script was fired successfully, producing the following facts:

```
DrillClass
OrderClass
OrderClass
TCardClass
ProductionLineOperativeClass
OrderSchedulerClass
OrderDownloaderClass
TCardPrinterClass
```

3.5. Attempted Integration

Pairing Facts "DrillClass" and "DrillClass".
Pairing Facts "OrderClass" and "OrderClass".
Pairing Facts "OrderClass" and "OrderClass".
Pairing Facts "OrderDownloaderClass" and "OrderDownloaderClass".
Pairing Facts "OrderSchedulerClass" and "OrderSchedulerClass".
Pairing Facts "ProductionLineOperativeClass" and "ProductionLineOperativeClass".
Pairing Facts "TCardClass" and "TCardClass".
Pairing Facts "TCardPrinterClass" and "TCardPrinterClass".
Integration was attempted using the "Additive" strategy.
Integration was successful.

4. Cycle 6

4.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

4.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

4.3. Attempted Chunk-based Inference

Inference Chunks Found

4.4. Attempted Integration

Pairing Facts "transferAllCompleteObjects" and "transferOrderObjects".

Pairing Facts "transferCompleteObject" and "transferOrderTo".

Integration was attempted using the "Additive" strategy.

Integration was successful.

5. Cycle 8

5.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

5.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

5.3. Attempted Chunk-based Inference

Inference Chunks Found

5.4. Attempted Integration

Pairing Facts "allocateAllObjects" and "allocateOrderObjects".

Pairing Facts "allocateObject" and "allocateToOrder".

Integration was attempted using the "Additive" strategy.

Integration was successful.

6. Cycle 10

6.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

6.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

6.3. Attempted Chunk-based Inference

Inference Chunks Found

6.4. Attempted Integration

Pairing Facts "manipulateAllObjects" and "manipulateTCardObjects".

Pairing Facts "manipulateObject" and "manipulatePropertyOfTCard".

Integration was attempted using the "Additive" strategy.

Integration was successful.

7. Cycle 12

7.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

7.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "HasISMObjectsRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

7.3. Attempted Chunk-based Inference

Inference Chunks Found

7.4. Attempted Rule-based Inference

Production Scripts Found

7.5. Attempted Elicitation

Elicitation Stimuli Found

Attempting to fire the Elicitation Stimulus "FunctionalResourceConfigurationStimulus", a fact-editing stimulus

The elicitation stimulus was fired successfully, producing the following facts:

```
ManipulationDeviceCommsProtocol
ClassificationRuleSource
RemoteDataTransferProtocol
```

7.6. Attempted Integration

Pairing Facts "ClassificationRuleSource" and "ClassificationRuleSource".

Pairing Facts "ManipulationDeviceCommsProtocol" and "ManipulationDeviceCommsProtocol".

Pairing Facts "RemoteDataTransferProtocol" and "RemoteDataTransferProtocol".
Integration was attempted using the "Additive" strategy.
Integration was successful.

8. Cycle 14

8.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

8.2. Attempted Quality Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "false".

New Information Requirement Generated

Goal Fact Type: Resource

Goal Postcondition:

```
PREFIX im:
<http://jamesnaish.wordpress.com/ROREKnowledgeModel.ow
l#>  PREFIX ltm:
<http://jamesnaish.wordpress.com/LongTermMemory/phd.ow
l#> ASK { Individual(?p), Type(?p, im:Property),
PropertyValue(?ismprop, im:instantiatesFactType,
ltm:ISMProperty), PropertyValue(?resource,
im:instantiatesFactType, ltm:Resource),
PropertyValue(?ismprop, im:hasProperty, ?p),
PropertyValue(?p, im:hasPropertyValue, ?resource),
PropertyValue(?p, im:instantiatesPropertyType,
ltm:hasDataSource)  }
```

8.3. Attempted Elicitation

Elicitation Stimuli Found

8.4. Attempted Elicitation

Elicitation Stimuli Found

Attempting to fire the Elicitation Stimulus "DataSourceAndSinkStimulus", a fact-editing stimulus

The elicitation stimulus was fired successfully, producing the following facts:

```
OrderScheduleClass
OrderDownloaderClass
OrderServerClass
OrderClass
OrderSchedulerClass
ManipulationScriptGenerator
TCardPoolClass
TCardClass
AllocationRule
```

```
DeliveryVanClass
DrillClass
ProductionLineClass
OrderPoolClass
TCardPrinterClass
DispatchAreaClass
ProductionLineOperativeClass
OrderManagementSystemClass
```

8.5. Attempted Integration

Pairing Facts "AllocationRule" and "AllocationRule".
Pairing Facts "DeliveryVanClass" and "DeliveryVanClass".
Pairing Facts "DispatchAreaClass" and "DispatchAreaClass".
Pairing Facts "DrillClass" and "DrillClass".
Pairing Facts "ManipulationScriptGenerator" and "ManipulationScriptGenerator".
Pairing Facts "OrderClass" and "OrderClass".
Pairing Facts "OrderDownloaderClass" and "OrderDownloaderClass".
Pairing Facts "OrderManagementSystemClass" and "OrderManagementSystemClass".
Pairing Facts "OrderPoolClass" and "OrderPoolClass".
Pairing Facts "OrderScheduleClass" and "OrderScheduleClass".
Pairing Facts "OrderSchedulerClass" and "OrderSchedulerClass".
Pairing Facts "OrderServerClass" and "OrderServerClass".
Pairing Facts "ProductionLineClass" and "ProductionLineClass".
Pairing Facts "ProductionLineOperativeClass" and "ProductionLineOperativeClass".
Pairing Facts "TCardClass" and "TCardClass".
Pairing Facts "TCardPoolClass" and "TCardPoolClass".
Pairing Facts "TCardPrinterClass" and "TCardPrinterClass".
Integration was attempted using the "Additive" strategy.
Integration was successful.

9. Cycle 16

9.1. Attempted Completeness Analysis

Analysis Rules Found

Fired Analysis Rule "DataSourcesSpecifiedRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasMonitorReportRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "DataSinksSpecifiedRule" with result "false". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasFunctionalResponsibilitiesRule" with result "true". Overall conclusion is: Model is not complete

Fired Analysis Rule "HasISMObjectsRule" with result "true". Overall conclusion is: Model is not complete