A SOFTWARE COMPONENT MODEL WITH CONCURRENCY

A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy in the Faculty of Engineering and Physical Sciences

2009

By
Ioannis Ntalamagkas
School of Computer Science
# Contents

Abstract .......................................................... 11

Declaration ......................................................... 12

Copyright .......................................................... 13

Acknowledgements ................................................ 14

1 Introduction ...................................................... 15
   1.1 Components and component models ......................... 16
   1.2 What this thesis is about ................................... 17
       1.2.1 Contributions in this thesis ......................... 18
       1.2.2 Structure of the thesis ............................... 19

2 Background ...................................................... 21
   2.1 Components and component models ......................... 21
   2.2 Desiderata for a component model ......................... 23
       2.2.1 An idealised component life cycle .................. 25
   2.3 Related work ................................................ 27
       2.3.1 Software Architecture Description Languages ........ 28
       2.3.2 Pecos ................................................. 32
       2.3.3 SOFA 2.0 ............................................. 33
       2.3.4 Fractal ............................................... 34
       2.3.5 EJB .................................................. 35
       2.3.6 OSGi ................................................ 37
       2.3.7 COM/COM+ .......................................... 39
       2.3.8 .NET ................................................. 41
       2.3.9 Javabeans ............................................ 44
2.4 Discussion .................................................. 45
2.5 Chapter summary ......................................... 48

3 Our component model ..................................... 49
  3.1 Exogenous connection ................................. 50
  3.2 Components and connectors ......................... 51
    3.2.1 Formal definition ............................... 54
  3.3 Component composition .............................. 58
  3.4 Connector composition .............................. 61
  3.5 Concurrency at the model level .................... 63
  3.6 Chapter summary ...................................... 65

4 Atomic and composite components ....................... 67
  4.1 Passive atomic components .......................... 67
  4.2 Concurrency control ................................... 73
    4.2.1 The SOS paradigm and ESP .................... 76
    4.2.2 Applying SPs to our model .................... 79
    4.2.3 Formal semantics ............................... 81
    4.2.4 Discussion ...................................... 83
  4.3 Active atomic components ........................... 83
    4.3.1 Coloured Petri-net semantics .................. 86
    4.3.2 Interface generation ........................... 88
  4.4 Examples of atomic components ..................... 89
  4.5 Passive composite components ...................... 91
    4.5.1 Sequencer and pipe ............................. 92
    4.5.2 Cobegin and efficient cobegin ................ 100
    4.5.3 Selector and non-deterministic selector ........ 102
  4.6 Active composite components ....................... 104
    4.6.1 Stateful Facade ................................ 104
  4.7 Chapter summary ...................................... 108

5 Adapted components and component execution .......... 109
  5.1 Adapting components .................................. 109
    5.1.1 Adapting the control flow of a component .... 110
    5.1.2 Adapting the interface of a component ........ 114
  5.2 Transition into run-time phase ..................... 118
List of Tables

8.1 Summary of WCPs that can be expressed in our model. . . . . . . 181
8.2 Summary of WCPs that cannot be expressed in our model. . . . . 185
List of Figures

2.1 An idealised component life cycle. 26
2.2 C2 connection and composition. 29
2.3 Wright connection and composition. 30
2.4 Darwin connection and composition. 31
2.5 Acme connection and composition. 31
2.6 Pecos component model. 32
2.7 SOFA components and composition. 33
2.8 Fractal component model. 35
2.9 EJB components and composition. 36
2.10 OSGi components and composition. 38
2.11 COM/COM+ components and composition. 39
2.12 COM/COM+ containment and aggregation. 40
2.13 .NET components and their composition. 41
2.14 JavaBeans and their composition. 44
2.15 Categories based on composition (adopted from [77]). 45

3.1 Forms of communication. 50
3.2 Components and composition in our model. 52
3.3 Definition of a component in Z. 56
3.4 Different component types in our model in Z. 57
3.5 Typical composition in design phase using our model. 59
3.6 Building a system in the deployment phase. 60
3.7 Component and connector life cycles. 61
3.8 The observer design pattern as a composite composition connector. 62
3.9 Concurrent processes. 63

4.1 Passive atomic component. 68
4.2 Definition of a computation unit in Z. 69
4.3 Control flow of the passive primitive component.

4.4 Events of a typical method invocation in SOS (adopted from [93]).

4.5 Summary of the ESP syntax.

4.6 Synchronisation contracts for exogenous connectors.

4.7 Synchronisation contracts for composite components.

4.8 Definition of a synchronisation constraint in Z.

4.9 CPN semantics for the mutex synchronisation constraint.

4.10 Active atomic component.

4.11 Executing an active component.

4.12 Example behaviour of active computation processes.

4.13 Control flow of an example active component.

4.14 Interface generation for active atomic components.

4.15 A passive password checking component.

4.16 A simple flight monitoring component.

4.17 The sequencer and pipe composition connectors.

4.18 Password and fingerprint checking composite.

4.19 Authentication composite component.

4.20 Construction of a composite using the sequencer/cobegin in Z.

4.21 Construction of a composite using the pipe in Z.

4.22 CT-net for a binary sequencer/pipe.

4.23 The cobegin composition connector.

4.24 Password and fingerprint checking composite.

4.25 Efficient cobegin merging two out of three processes.

4.26 The selector composition connector.

4.27 The nd-selector composition connector.

4.28 Construction of a composite component using a (non-deterministic) selector in Z.

4.29 The facade composition connector.

4.30 Authenticated flight monitoring system.

4.31 Authenticated flight monitoring system.

4.32 Construction of a composite using a stateful facade in Z.

5.1 Guard adaptor.

5.2 Sequential loop with fixed number of iterations.

5.3 Finite parallel loop adaptor.

5.4 Merging input parameters adaptor.
Abstract

Component-Based Software Development (CBSD) represents a paradigm shift in software engineering, where the focus is on building systems out of pre-developed components. In CBSD, components developers are different from component users, where the former concentrate on building reusable components, whereas the latter concentrate on building customised systems out of components.

The different roles in component development give rise to the idealised component life cycle which consists of the design, the deployment and the run-time phases. Ideally, component composition should occur in both design and deployment phases and it should be supported by a proper composition theory.

Concurrency is an important issue in software engineering and in theoretical computer science. Decades of research have been devoted in finding efficient ways of discovering common concurrency errors, like deadlock, in abstract mathematical models and in software.

Existing software component models do not address composition and concurrency completely. In general, component models support composition in either design or deployment phase but not in both. Regarding concurrency, the support provided varies, ranging from complete formal models to leaving the concurrency aspect undefined and dependant on the implementation. More importantly, not all component models support both passive and active components.

In this thesis we define a software component model that supports composition of active and passive components using explicit composition operators, in both design and deployment phases of the idealised component life cycle. We also show that our composition connectors are control patterns and we define their composition. This means that composite connectors are composed out of simpler ones. This allows for the hierarchical construction of complex control structures that can be used in further connector and component compositions. Connector composition constitutes a unique feature of our model.
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

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the “Copyright”) and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii. The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the “Intellectual Property Rights”) and any reproductions of copyright works, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and exploitation of this thesis, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of School of Computer Science (or the Vice-President).
Acknowledgements

This thesis amalgamates into a single document the results of my PhD study. It also reflects the educational path that I have followed since the early years of my life. My parents, my family, my teachers, my friends, my colleagues, have guided and helped me throughout this trip. It is absolutely certain that without them I would not be in the place that I am today. To all these people, I can only express my gratitude. Thank you!!

First, I would like to thank my supervisor Dr. Kung-Kiu Lau, for offering me the opportunity to study for a PhD and for successfully supervising and supporting me throughout my studies. Kung-Kiu’s insights into research and constructive criticism carved my academic skills and taught me how to perform research. Thank you!

I wish to thank my partner, Hara, for supporting me in multiple ways during the long period of the PhD. Hara has made PhD difficulties appear less and of course, I can not thank her enough for wholeheartedly supporting me during the writing-up period. I would like to thank Andreas for being a friend, for genuinely understanding, and sharing, my complaints about the harsh PhD life and for making me laugh. I would like to thank Elton for our hiking Saturdays, for the strong Albanian raki and for tolerating the ‘bal–hYYYsa’ joke!

I am grateful to my former and my current colleagues in the University of Manchester, Cuong, Faris, Vladik, Keng-Yap, Perla, Ling, Lily, Tauseef and Azlin. Our long discussions generated new ideas, improved old ones and eliminated wrong directions. In essence, our debates shaped my work into its current form. Thank you guys for being friends – not only colleagues.

I am also indebted to my family for their continuous support during my long education. To my parents Miltiades and Maria, my sister Katerina, my brothers Kostas and Leonidas, my aunt Niki, my cousins Lambroula and Takis, my uncle Sotiris, all of you have made a contribution into what I deliver today.

To all of you, my friends and my teachers, thank you!
Chapter 1

Introduction

Since the early years of software engineering, there has been a constant demand on programmers to produce error-free software on time and within budget. To fulfil these demands, software engineers have developed numerous techniques for building software systems. From the waterfall model of the 1970’s we have recently moved to extreme programming techniques and agile methods [92]. Programming languages have also rapidly evolved, reflecting to some extent the changes in programming methods. The first evolution was with assembly languages that enabled programmers to program using symbolic names instead of machine code. The next paradigm shift was with procedural programming, where programs were organised in procedures. Procedural programming brought significant benefits to the programming community including code reuse, minimisation of the “spaghetti code” problem and more significantly, programs that are easier to write and debug. Object-oriented (OO) programming was the next significant step; programming units are now objects that encapsulate both data and procedures. OO programming led to better structured programs, more extensive software reuse and therefore significant gains in productivity.

However, all these programming paradigms and methods fall short when aiming for large scale software reuse, decoupling software production from deployment, or when composing software units developed by third-party suppliers. Component based software engineering (CBSE) or component based software development (CBSD) is the new paradigm shift that promises to dramatically change the way software systems are built.
1.1 Components and component models

The notion of a software component is apparently central in CBSE. It was first coined by Douglas McIlroy in his talk titled “Mass produced software components” [95], where software components were considered to be single program procedures. Component markets were envisaged where the merchandise would be families of procedures. Since that time, programming languages and software engineering methods have largely evolved and it has become clear that a market of procedures is too fine grained to become a reality. More recently, researchers in the field [123] also shared the vision of a component market where component developers and clients will be able to buy and sell components. Purchased components will be integrated by component clients into their applications by using predefined composition mechanisms. If this component market is large enough, it will be possible to build a software system from scratch using only purchased components.

It is clear that CBSE aims to revolutionise the way software systems are built. Instead of monolithic systems built from scratch with little or no re-use, systems could be built by assembling parts or components bought from independent developers [26, 24]. Such an approach reduces production costs and time-to-market as the system is built from pre-existing entities. It also enables software reuse as the same components are used in different systems. A software component must therefore be defined in a way that is easy to reuse.

From the above it is clear that software components do not suffice by themselves to make CBSE succeed. Composition or assembly of components into systems is also critical to the success of CBSE. Composition defines how to build systems out of components. Therefore, a comprehensive CBSE method should not only define what components are, but it is equally important to define how components can be composed to form systems. This is the role of software component models. A software component model provides the necessary infrastructure for storing components, retrieving them and composing them into systems according to a composition standard. This is the purpose of this thesis.
1.2 What this thesis is about

In this thesis we define a software component model with compositionality and explicit support for concurrency. As we mentioned earlier, component composition is central in CBSE and it is our belief that composition should constitute the cornerstone of any component model. We relate component composition to the life cycle of components. Components can be composed either during design time when they are being developed as generic, reusable entities, or during deployment time when they are being deployed into a specific environment with the purpose of building a specific system. As we shall show through our survey, component composition in both the design and deployment phases of the component life cycle, is an issue neglected by other component models. These, in general, support composition in either phase, but not in both. Consequently, we fill this gap by presenting a component model that supports composition during both design and deployment phases of the component life cycle.

Composition in our model is hierarchical and is performed via explicit composition operators. This means that we use explicit operators for coordinating the control and data flow for a set of components. When this happens, a composite, hierarchical component is created that can be similarly composed with other components via explicit operators. The control and data flow defined by our connectors can be concurrent. In fact, the sources of concurrency in our model are connectors and components. Components may define their own processes and execute in them, and these processes may be concurrent. We call these components active components. On the other hand, a passive component does not define its own processes, but its interface may allow concurrent invocations of its methods; such invocations are actions in concurrent processes. Composition mechanisms may also define concurrent processes. In our component model we define the composition of passive and active components in both phases of the component life cycle. Finally, in our model all concurrency is defined and managed at the modelling level via our components and connectors. This means that the implementation cannot create ‘new’ concurrency that is not already defined in the model.

Another important aspect of our component model is connector composition, where primitive connectors are composed into composite ones. As we mentioned above a primitive connector in our model is a (simple) control flow structure.
Therefore, via composing primitive connectors, we can build composite connectors of arbitrary complexity. Composite connectors can be (re-)used in other component, or connector, compositions. In this thesis we present a component model that provides a unifying modelling framework for connector and component composition with explicit support for concurrency.

1.2.1 Contributions in this thesis

The main contribution in this thesis is the definition of a component model with compositional semantics and explicit support for concurrency. This contribution can be split into three categories: (i) composition of components according to the idealised component life cycle, (ii) compositional approach to active and passive components, and (iii) composition of connectors. We further explain these contributions.

We mentioned earlier that in our model we can compose components in both the design and deployment phases of the component life cycle using explicit composition operators. As we show by surveying related work, Section 2.3, our approach is unique in that respect. Most of the connectors that we have defined are applicable to both design and deployment phases of the component life cycle. When a connector applies to only one of them, we mention it explicitly.

Regarding concurrency, the main contribution is the definition of a compositional approach to active and passive components. In our model we define both active and passive components and we explicitly distinguish between the two. In related component models, with the exception of Pecos [108], either all components are active, or they are all passive, or their exact nature can depend on their implementation. Although Pecos defines the composition of both active and passive components, this occurs only at design time and in fact, the deployment phase composition is undefined.

Finally, we have defined an algebraic approach to connector composition. We compose connectors to form composite connectors, where both the operator and the operands of the composition are connectors. As we shall show via the analysis of related work in the broader field of software engineering, our work constitutes a unique approach to software composition that further enhances software reuse.

In summary, we propose a software component model with a three-fold contribution to CBSE. First, we suggest a flexible approach to component composition taking into account the life cycle of components. Second, we explicitly distinguish
between passive and active components and we define their composition. Third, we define algebraic connector composition. The work presented in this thesis enhances the state of the art in CBSE, contributing a step towards the vision shared among researchers of the field, that of a software component market.

We should mention here that our work is based on previous research conducted by Lau and colleagues [83, 81, 82, 77, 76, 84]. These works defined a component model which has laid out the principles of the current work. In this work, we revise and extend the previous model in several directions. First, we make execution concurrent. Previously, the execution of only a single process was defined and all components were passive. Now, concurrent execution of processes is supported and components can be either active or passive. Second, although composition in both design- and deployment-phases had been identified and had had their importance highlighted [84], they were not incorporated into a single component model [76]. Precisely, there was no visual repository, assembler and builder, and the transition from design to deployment phase was a manual, ad-hoc process. In this work we unify design and deployment phase composition into a single model. The third extension to the previous model is related to connector composition, where the basic idea was presented in [77]. In the current work we have revised connector composition by defining its syntax and its (formal) semantics, and by integrating it into the component life cycle.

1.2.2 Structure of the thesis

In Chapter 2 we analyse the background material of our work. We adopt a definition for components and component models that we will use for the rest of the thesis. We then proceed and define the desiderata for a component model and we survey related work against them. This survey motivates our work by identifying omissions in the state of the art. This is the gap we intend to fill with the work presented in this thesis.

In Chapter 3 we present an overview of our component model. The basic constituent elements of our model, namely components and connectors are briefly presented. We also discuss component and connector composition in our model, illustrating their close relationship. Finally, in this chapter we define the terminology regarding concurrency that we will use in this thesis, and we briefly illustrate how our model supports concurrency.
In the following four chapters, we present in detail our model and its implementation. In Chapter 4 atomic or primitive components are discussed, as well as their composition. Concurrency is also introduced in this chapter. Specifically, we describe how concurrent processes are defined in our model and how we control concurrency. Chapter 5 focuses on the deployment phase of the component life cycle. Component adaptation is discussed and more importantly, we present how an already assembled system can start executing. In Chapter 6 we delineate connector composition. Precisely, we present the syntax and the semantics of connector composition and we explain the features of our model that make this possible. In Chapter 7 we show how we have realised our model. We demonstrate our implementation by means of a simple case study.

In Chapter 8 we conduct an experiment with the purpose of evaluating the expressiveness of our composition operators in terms of the control flow that they can define. In other words, we try to answer the question of whether our model is sufficiently expressively for the modelling and construction of a wide range of component-based systems. This evaluation is performed using engineering principles and not formal semantics. In Chapter 9 we provide a holistic evaluation of our work. We analyse it’s strengths and weaknesses in terms of the desiderata for a component model and the expressiveness of our composition. Finally, Chapter 10 concludes this thesis and suggests future research directions.
Chapter 2

Background

In this chapter we adopt a definition for components and component models that we will be using for the rest of this thesis. By surveying related work, we identify the required elements for a component model that are crucial in realising the aims of CBSE. These desiderata form the frame of reference that we compare ours and other component models against, and it is through these desiderata that we eventually evaluate our work. Related work is presented in this chapter. Our model and its evaluation are presented in the rest of the thesis.

2.1 Components and component models

Currently there is no universally accepted terminology on the definition of a component and of a component model. The latest trend in software engineering, i.e. object orientation (OO), does not suffice for defining components as objects, for the fundamental reason that “objects are too fine-grained and their deployment requires too much understanding of an object’s working” [123]. Another reason is that in OO systems computational and compositional aspects are interwoven [115], i.e. OO systems are mainly built for solving a particular problem (providing ad hoc solutions) and not for creating reusable, composable software elements. A coarser grained approach that emphasises the nature of components as independent entities and their composition is necessary. With the above we do not mean to criticise OO as an implementation technology for components and actually, components in our model are implemented as collections of objects with specific relationships, see Chapter 7. But for the given reasons, an object cannot be a component, and OO does not define a component model. This view is further
supported by the discussion that follows, where we survey existing component definitions and we finally adopt one for our own use.

Agreeing on a universally accepted definition of what is a component, has been the source of great debate among the researchers of the community [35]. Commonly cited definitions include the ones given by Szyperski [123] and Meyer [99] which are quoted below:

“A software component is a unit of composition with a contractually specified interface and explicit context dependencies only. An interface is a set of named operations that can be invoked by the clients. Context dependencies are specifications of what the deployment environment needs to provide, such that the components can function. A software component can be deployed independently and is subject to composition by third parties.” [123].

“A component is a software element (modular unit) satisfying the following conditions: 1) It can be used by other software elements, its “clients”. 2) It possesses an official usage description, which is sufficient for a client author to use it. 3) It is not tied to any fixed set of clients.” [99].

Although the above definitions try to capture the nature of components, they do not mention how a component interacts with other components in its environment and specifically, how the component can be composed with other components in order to build a fully functional software system. Heineman and Counsell [68] recognised this fact by introducing the notion of a component model:

“A [component is a] software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.”

The notion of a component model in central in this definition and it defines what a component is and how it can be composed. Sommerville [119] and, Lau and Wang [84] have proposed the following definitions for a component model:

“A component model is a definition of standards for component implementation, documentation and deployment. These standards are for component developers to ensure that components can interoperate.
2.2 Desiderata for a component model

They are also for providers of component execution infrastructures who provide middleware to support component operation.” [119].

“A software component model should define: (a) the syntax of components, i.e. how they are constructed and represented; (b) the semantics of components, i.e. what components are meant to be; (c) the composition of components, i.e. how they are composed or assembled.” [84].

The above definitions shift the focus from a component to a component model so that defining a component outside the context of a component model would be meaningless. In this thesis we adopt the definition of a software component model given by Lau and Wang. This definition is more generic and it demands component composition to be explicitly defined, as it constitutes the cornerstone of component-based systems. It emphasizes the fact that we are not only interested in making components interoperate, but instead we are primarily interested in creating composite components and software systems out of components. Also by defining the syntax and semantics of a component model, component interfaces, their “usage description”, and how they can be deployed, are also implicitly defined. In other words, it is our belief that Lau and Wang’s definition captures the essence of the services offered by a component model in a unifying framework so that it is sufficiently generic to be universally applicable.

2.2 Desiderata for a component model

As already explained, the cornerstone of any CBSE method is the underlying component model that defines what components are, how they can used and how they can be composed or assembled. By defining the above elements complementary issues to components are addressed, for example how components can be constructed and how they can be deployed. The previously quoted definitions of components and component models [123, 99, 68, 119, 84] along with the discussions in [35] highlight the requirements necessary for a component model to achieve CBSE ambitions. As mentioned in [84] these can be given as follows. First, components should be pre-existing software units, that can be reused by being deployed in different applications by different developers. Second, component development and component use should be separate activities, and consequently
the stakeholders in these two activities should differ. This helps ensure that components developed are well defined and whenever possible they are independent from other components. Precise component definition and component independence make components more amenable to reuse by third parties. Third, in order to be used multiple times, components should be capable of being copied and deployed into different application environments, in an effort to increase code reuse. Fourth, components should be composable into composite components which in turn can be further composed into other (composite) components. Composition promotes reuse and offers a systematic approach to system construction.

Summarising the above, it is clear that a component model is a modelling and implementation framework that defines a component-based method. At the modelling level, component semantics are defined and a more abstract view of how a component-based system can be constructed is presented. This abstract view is important as it not only defines what kinds of systems can be constructed, but since, additionally, the implementation details are hidden, several (formal) checks can be performed at the system’s architectural view. Both angles are important. A component model should be sufficiently rich to accurately describe component behaviour and their composition at a reasonable level of abstraction. Formal checks may be applied to components individually and to the composition of components, thus increasing the level of trust towards both components [99] and systems built using components. This is particularly important in the context of concurrency. Concurrency errors are notoriously difficult to identify and reproduce. Model checking techniques [47] have resulted in the development of automated tools for that purpose. But because of the infamous state explosion problem, exhaustive model checking is mostly applicable to abstract, mathematical models and not at the source code level. Therefore it is important that a component model can describe, at a certain level of abstraction, the behaviour of a component based system.

Concerning concurrency, it is our belief that both active and passive components, and their composition, should be modelled because they express distinct requirements in terms of developed systems. Active components are usually more appropriate to reactive systems “whose role is to maintain an ongoing interaction with their environment, rather than produce a final value on termination” [91], whereas passive components are more suitable for modelling server-side systems where components passively wait to be invoked by external clients. This is for
instance the case with EJB [3, 41]. Consequently it is important that in a com-
ponent model both active and passive components and their composition are
explicitly defined.

Other than a modelling framework, a component model should additionally
offer an implementation framework in which components may be implemented,
composed, deployed into an execution environment and finally execute. It is a
dual relationship and it is our belief that a component model should support both,
albeit in a varying degree depending on the concrete aims of the model. If poor
implementation support is provided then the model’s practical applications are
limited and conversely, poor modelling capabilities reduce the range of systems
that can be modelled and can be subsequently verified.

As already explained, composition is one of the most important aspects of a
component model. Composition is inherently tied to the life cycle of components
and a component model should be as flexible as possible regarding when and
how composition may be performed. This, along with the component model
desiderata, forms the notion of an idealised component life cycle and is explained
below.

2.2.1 An idealised component life cycle
Composition of components should follow the component life cycle [46, 51]. In
essence there are two development stages with regards to components before a
component-based system starts execution (mentioned as binding time of composi-
tion in [24]). The two different stages are associated with two different stakehold-
ers of the component-based design process: component developers and component
users. The first ones define the first stage of component’s life cycle, namely develop-
ment of components, whereas the second ones define the second stage, namely
development with components.

In [84] it is claimed that in order to maximise reuse, composition should occur
during both these two stages of the component life cycle, before a system transits
into its run-time phase. These two stages are called design and deployment phases
of the component’s life cycle and are shown in Figure 2.1. In design phase com-
ponents are designed and implemented in source code, and may be also compiled
into binaries. Implementation may be provided in more than one programming
language in order to maximise design reuse. Designed components are stored in
a component repository. The focus of this phase is to build reusable components.
In the deployment phase, already built components and are retrieved from the repository and are deployed into an execution environment which makes them ready for execution. The purpose of this phase is to build the target system. Finally, in the run-time phase, the necessary resources are allocated to components, for example memory and other system resources, and the components start executing. We now explain these phases into detail.

In the design phase components are identified, implemented in possibly more than one implementation language, and are stored in a repository. A repository can be a registry or a directory, and its actual format is not important. What is important is that it must store and catalogue the deposited components and it must offer utilities for searching and retrieving components. A builder tool allows the construction and composition of components. For example, in Figure 2.1, the primitive or atomic component A is constructed from scratch and is stored in the repository. Atomic components are composed into composite components, for example component BC is built from the composition of B and C. Composite components are also stored in the repository, and can be later reused in another composition in the design phase, or they can be directly deployed and form part of a larger system. In the builder,s copying of components should be possible, so that for instance, the same component can be used in different composites.

The deployment phase is supported by an assembler tool. The assembler can retrieve components from the repository and deploy them in a deployment environment. Both atomic and composite components may be deployed. For
example, in Figure 2.1 atomic components A, B, D and composite component BC are deployed. During deployment, source code compilation may be needed, and this depends on the form of the components retrieved from the repository. In the deployment phase further composition is possible by deployment phase composition operators.

After the whole system has been assembled or composed, it can start executing. Deployed components now become component instances by acquiring all necessary resources needed to execute. For example, components are loaded into memory and they are “started” by the execution environment usually by executing some designated method of the component. In some models, dynamic composition and re-configuration is possible, for example [128]. However we do not include run-time composition in the idealised component life cycle, because at run-time “it is hard to define meaningful composition operators other than glue code” [84].

For the idealised component life cycle in Figure 2.1 we have used explicit composition operators in both design and deployment phases. Explicit composition operators are in accordance with the well-known software engineering principle of separation of concerns [55]. Explicit composition operators separate component communication and composition from the components themselves. Components need not be aware of other components, as intercomponent communication is handled by the connectors, and components can ‘concentrate’ on their primary role which is computation. This is a view shared by the community of coordination languages [63]. By separating coordination from computation, greater reusability is achieved by reusing the same coordination language in many different systems and by using a single coordination language for bridging different systems. In terms of components and connectors, it reinforces our view, as the same composition operators can be (re-) used across a range of systems, and can also be used to bridge between different systems. Finally the connectors should be sufficiently generic to support the composition of both active and passive components.

### 2.3 Related work

In this section we present related work in the light of the above desiderata for a software component model. Specifically, we survey software component models in terms of component composition and concurrency. Component composition is
analysed according to the idealised component life cycle. Concurrency is analysed in terms of support for active and passive components, and in terms of whether concurrency is defined and supported at the modelling level or is left undefined and dependant on the implementation. If formal verification is possible, then this is explicitly mentioned. Our discussion is based on the surveyed component models in [84] from which we choose representative examples that cover all distinct approaches in terms of composition and concurrency.

It is worth noting that our work also extends to connector composition, see Section 1.2. This introduces a new angle of comparison and analysis which we do not cover in this section. We have avoided doing so because in this section our focus is on the desiderata of component models, of which connector composition is not a part. We explain related work in terms of connector composition in Section 6.3.

### 2.3.1 Software Architecture Description Languages

In software Architecture Description Languages [118] (ADLs) there are three basic entities, components, connectors and architectural configurations [48, 97]. The semantics of components and connectors vary, but typically components and connectors are represented graphically as boxes and lines respectively, and an architectural configuration defines how these are linked together. Components perform computation and can store data, whereas connectors are used for realising inter-component data, and possibly control, flow.

Regarding composition, a common denominator for ADLs is that this occurs at the design phase only. Components are connected via connectors, according to the architectural configuration. However, components are only templates with specific interfaces whose behaviour may or may not be defined using a formalism. There exists no source or binary code for these components, and there is no repository where components can be stored and later on retrieved for further composition. Depending on the tool support, the source code of the architectural configuration may be generated, providing the skeletons for component implementations. In ADLs, the design-phase is typically followed by the run-time phase.

With respect to concurrency, all ADLs are inherently concurrent. Components are considered active entities that execute in their own active process(es). Connectors are also concurrent in that multiple control and data communications
2.3. RELATED WORK

can occur simultaneously. Since the focus of ADLs has traditionally been the definition of the high level architectural configuration, it is usually the case that a formalism has been applied for describing components and their interactions.

As representatives of ADLs we briefly present below the component life cycles and support for concurrency in C2 [125], Wright [18], Darwin [87] and Acme [62]. Other ADLs are similar with respect to composition, component life cycle and concurrency as is discussed in [97].

C2 is a “a component- and message- based” architectural style that defines a “network of concurrent components hooked together by message routing devices” [125]. Components in C2 are architectural units that communicate with other components only via connectors; the connectors in C2 are message buses. C2 components have two ports that are used for communicating through the connectors with other components. A typical architecture in C2 looks like the one in Figure 2.2(a). Components emit requests to the components above them through their top ports and reply to requests with notifications through their bottom ports. Sub-architectures in C2 can be seen as composite components.

For instance, component C5 in Figure 2.2(a) could be the composite one shown in Figure 2.2(b).

In C2, after an architecture has been described, the DRADEL tool [96] can be used for automatic generation of a skeleton implementation. Component functionalities (computations) are of course not implemented, and skeleton Java classes are generated instead. The developer needs to provide the source code for these functions. Once the missing parts are provided, the architecture can

\[\text{(a) An example architecture in C2} \]

\[\text{(b) Composition in C2} \]

Figure 2.2: C2 connection and composition.
be executed in the target execution environment, namely the Java Virtual Machine. Implementations of C2 also exist in C++ and Ada and it is possible for components written in different languages to communicate [52].

Regarding concurrency, all components in C2 are active. Conceptually, they define and execute on their own process. Components in C2 communicate via buses which are inherently concurrent in that messages can be exchanged concurrently and in multiple directions. Although C2 uses the Design-by-Contract paradigm [98] for defining components’ behaviour and architectural evolution, no modelling notation is used for explicitly describing concurrency.

In Wright [18], components and connectors are active entities. Components perform computation, whereas connectors define the inter-component communications. Components have ports which are connected to the roles of connectors according to an architectural configuration where output ports are connected to input roles in a one-to-one mapping and vice-versa. An example of an architectural configuration is shown in Figure 2.3(a). Figure 2.3(b) shows that composite components can be defined, where for instance component C3 in Figure 2.3(a) is shown to be a composite one. The main focus in Wright is the description of architectural configurations and their formal analysis. Specifically, components’ and connectors’ behaviour in Wright have been formalised in Communicating Sequential Processes (CSP) [70], a process algebra used for defining and checking concurrent systems. Components and connectors are only abstract templates and there is no tool that supports code generation out of Wright configurations. Therefore, a system developer has to manually implement the system architecture in an appropriate programming language.

In Darwin [87] components have provided and required services that are connected via bindings. An example is shown in Figure 2.4(a). In the figure it is also shown that components can only be bound inside a composite component,
2.3. RELATED WORK

i.e. a composite has to be formed whenever component services are bound. Figure 2.4(b) shows that more than one level of composition is possible.

With respect to concurrency, Darwin has been formalised in Finite State Processes (FSP) [90, 89] and π-calculus [100]. Consequently, several formal checks are possible at the architectural level. Similar to C2, so in Darwin, all components are active entities. After the system has been designed, it can be compiled into C++ source code [88] where the programmer can directly implement the provided services inside the generated C++ classes.

In Acme [62] components and connectors are explicit entities that are conceptually active. Components are units of computation and data storage, whereas connectors are used for inter-component communication. Similar to Wright, components in Acme have ports which are connected to the roles of the connectors as shown in Figure 2.5(a). However, and in contrary to Wright, in Acme it is possible for a connector role to connect to more than one port. Definition of composite components is also possible in Acme, see Figure 2.5(b).

In Acme, after the system architecture has been designed, it is analysed in AcmeStudio [116]. Since Acme lacks formal semantics, the analyses do not check for concurrency errors like deadlock and livelock. Instead, they are used for performance and schedulability evaluations. After the analyses have been performed,
ArchJava [15, 16] is used to create some ‘starter’ code, i.e. templates for the developer to complete their implementation. In other words, after the system’s architecture is completed, the next stage is run-time, provided that suitable implementations for components exist. The run-time environment is a Java Virtual Machine.

2.3.2 Pecos

Pecos [108, 64] is a software component model targeted at embedded systems and real-time devices. Components in Pecos are defined in the CoCo language. Components have (directed) interfaces from which they read, or to which they write data. Through component composition, Figure 2.6(a), components may communicate with each other. The composite component is written in CoCo and apart from the constituent components it also defines how they connect to each other. Connectors in Pecos are implicit, in the sense that connected ports define shared data which components use to communicate.

Regarding the component life cycle, Pecos defines no component repository. In the design phase, the architecture is defined and for the primitive components, the developer must provide the source code that implements the component’s functionality. The source code can be either in C++ or Java. Once the source code has been provided, the system (top-level composite component) can be compiled and start executing in the target execution environment.

In the Pecos component model, both active and passive components are explicitly modelled. Passive components are executed by a single process exclusively, the process of the composite component they are part of. This is because composite components in Pecos are always active and define their own process. Primitive active components may internally initiate additional processes but there are no constructs for this purpose at the component model level. The developers
of active atomic components must also define explicit synchronisation code using programming language constructs for safely communicating data with other components. This is because communication in Pecos occurs via shared data variables.

2.3.3 SOFA 2.0

Software Appliances 2.0 [38, 12] is the newer version of the SOFA [109] component model. Henceforward, whenever we mention SOFA, we mean SOFA 2.0 unless we explicitly say otherwise. Components in SOFA are either primitive, implemented in Java, or composites that consist of other components. For instance, in Figure 2.7 we may consider components C1, C2 and C4 as primitive ones, whereas component C3 is a composite. The boundary of a component is defined by its frame, which also defines its required and provided interfaces. Components communicate by binding their interfaces. Components can be bound inside composites, like C1 and C2 inside C3, provided they are at the same level of nesting. For example, were C2 composite, it would not be possible to compose its constituent sub-components with C1. Other than binding inside composite components, components can also be bound at the top-level of nesting, like C3 and C4 in the figure. External interfaces of sub-components may be forwarded to the frame of the composite via special kinds of bindings.\(^1\) The sub-components and the internal bindings of a composite component constitute its architecture.

Regarding the life cycle of SOFA components, in the design phase, primitive and composite components are defined, implemented and stored in a repository. Components can be retrieved from the repository, can be used in further compositions and the new composites are stored back into the repository. At deployment

\(^1\)For simplicity we name them bindings; the correct terminology is delegation and subsumption for provided and required interfaces respectively. For the same reason we do not discuss microcomponents. Microcomponents are similar to Fractal’s control interfaces, see Section 2.3.4, and they are related with controlling the component’s life cycle.
time, components are deployed into potentially distributed, deployment docks. No further composition is possible for deployed SOFA components. At run-time, components start executing inside the deployment dock, which provides the necessary infrastructure for starting, stopping and dynamically updating components. SOFA alleviates the lack of deployment phase composition by allowing for dynamic re-configurations which are managed through the deployment docks. The repository and the deployment docks collectively constitute a SOFANode which is the run-time environment for SOFA. Each deployment dock is a Java Virtual Machine, plus the SOFA run-time libraries.

In the SOFA component model, behaviour protocols have been defined for describing the interfaces of software components and for checking various levels of conformance during composition [110]. Specifically, behaviour protocols define the allowed interplay between method invocations on the provided interfaces and the triggered reactions on the required methods of the required interfaces. However, behaviour protocols only capture the allowed concurrency at the interface level of components, and concurrency and synchronisation are actually defined at the implementation level. They only describe the interface behaviour of components and they do not constitute a mechanism of defining and managing concurrent processes. Primitive components may spawn new processes at will and access to shared resources within primitive components is dealt with by the implementation of the component’s methods.

2.3.4 Fractal

In the Fractal [58, 36] component model, components are run-time entities consisting of a content and a membrane, as shown in Figure 2.8. A component’s content consists of other sub-components, except for primitive components that only contain some program implementing its provided and required interfaces. Without going into detail, membranes encapsulate the content and contain the internal and external interfaces of the component. External interfaces are accessible from outside the components, and internal interfaces are only accessed by sub-components. External interfaces of components are either their provided and required services, or controller interfaces used for maintaining the life cycle of the component. Controller interfaces are for example used for starting or stopping a component’s execution. Components communicate via bindings provided with required interfaces as shown in the figure. Bindings are actually method calls and
2.3. RELATED WORK

In Fractal, composition occurs in the design phase. Components are declared and implemented in the target implementation language using the Fractal API and the appropriate framework. No repository exists for storing and re-using components. Currently, there exist APIs and implementation frameworks for generating the implementation in C and Java [58]. After all components and their connections have been defined, the system can be compiled and run in the appropriate execution environment.

The behaviour protocols [110] that were defined for the SOFA component model, have been also applied to Fractal for performing similar checks for the composed components, like protocol conformance checking [74]. Behaviour protocols have also been used for checking that the implementation of primitive components complies to their specification [37]. In this case, the implementation has to be small in size, in the order of a few hundred lines of code, due to the state explosion problem. But similarly to the SOFA component model, behaviour protocols simply describe the expected interface behaviours of components. Concurrent processes and synchronisation are issues completely handled by the implementation.

2.3.5 EJB

Enterprise Javabeans (EJB) [3, 41] is a server-side software component model. An EJB component is a Java object hosted and managed by an EJB container.
which is part of a J2EE Server (for example of J2EE servers see [6, 7]). The EJB container provides an execution environment for EJB components and mediates client access. A typical configuration is shown in Figure 2.9. In the following, we avoid presenting a detailed description of EJB components and how they interact. Instead, we present an overview that suffices for the purpose of analysing the EJB component model w.r.t. composition and concurrency.

There are three kinds of EJB components. Entity beans, session beans and message driven beans. Entity beans model business data by providing an object view of data in a database. Session beans represent business logic and execute on behalf of external or internal clients. Message driven beans execute asynchronously on receipt of a client message. In EJB, components (objects) communicate via direct method calls. This is statically specified in the design phase, when the source code of the respective EJB classes is defined. Specifically, components do not directly reference other components, but they look them up by name and they rely on the container’s run-time services so that the appropriate component/ object is linked. A component may (indirectly) reference any number of components, and vice versa. Composition therefore occurs at the design phase.

After the source code of components is defined, components are deployed individually into the container, as the EJB container does not support the definition...
and storage of composite components. The container therefore acts as a deposit-only repository solely containing deployed primitive components. At run-time, the EJB container instantiates and executes EJB objects and the correct links (object references) are automatically created. In summary, each component developed into design phase is individually deployed into the EJB container and at run-time it starts executing within the same container.

EJB have introduced restrictions in the way concurrency is defined and handled. In EJB all beans are passive. Sessions and entity beans may be accessed concurrently by multiple processes, which are conceptually initiated by external component clients. In session beans this is achieved by multiple instances serving multiple clients concurrently. Entity beans on the other hand may be accessed from many clients, but the EJB container serialises all method calls to the beans. Beans may not spawn additional processes and the EJB container forbids that. Processes may be dynamically created, but this is performed at the EJB container level by executing message driven beans on a separate process. Consequently, component developers have no means of defining concurrent processes.

2.3.6 OSGi

The Open Services Gateway Initiative (OSGi) platform [19] defines an architecture for the development, deployment and management of services. The central role in the OSGi platform is played by a Java-based framework, or container, that manages the life cycle of components (for examples of containers see [4, 8]). Components in OSGi are termed bundles. Bundles are physically distributed as JAR files with a manifest file containing meta-information about the bundle, like the imported, or required, and exported, or provided, services. The container allows for dynamic loading and updating of bundles, as well as starting and stopping their execution. In order to be able to dynamically locate and link bundles, the container also offers some additional services. For example, it offers a registry where once a bundle is started, it can register the services that it offers and can also look for services from other bundles. The container itself executes inside a Java Virtual Machine.

\footnote{In [65] composite EJB components are defined. However, it is not clear how a composite can be used in further compositions. We classify this work as experimental and incomplete, hence we do not consider it in this discussion.}
Composition in OSGi is shown in Figure 2.10. Composition occurs at design time when individual bundles are developed. At design time, the Java classes that constitute a bundle are developed and then packed into a JAR file. In the figure we assume that Bundle2 and Bundle3 have exported some methods through their exported services. Bundle1 can import those services and can thus gain access to the exported methods. The import declarations are appropriately reflected in the manifest file of Bundle1 (and so are the import/export declarations in the other two bundles). At run-time, the correct object references are obtained through the OSGi container, and the referring bundle (Bundle1) can directly call methods of the referenced bundles as shown in the figure. Therefore, although the actual component bindings occur at run-time, composition of OSGi components occurs semantically at design time. Similar to EJBs, no composite bundles can be defined.

It should be mentioned here that in [54] a ‘composite’ bundle is defined. It is a JAR file containing a BPEL [21] description for invoking other OSGi bundles and a single activator class whose role is explained below (BPEL is a workflow language used for orchestrating web services). Once the bundle is started, the activator class notifies a specialised BPEL engine, which is implemented as another bundle, about its existence. The BPEL engine then creates a new bundle that implements the workflow defined by the composite service. The implemented ‘composite’ bundle can be located and used like any other bundle via the BPEL engine. However, in our opinion this is not a composite bundle, as it is semantically identical to any other (primitive) bundle using the services of other bundles.
The only difference is that its interface and the invocations to other bundles are defined in BPEL instead of plain Java.

Regarding concurrency, in OSGi a bundle can be active or passive, it can dynamically create new processes (Java threads) that concurrently invoke other bundles and a bundle can also be executed by concurrent processes. In fact, all concurrency and synchronisation is defined internally by the bundle by directly using constructs of the implementation language.

### 2.3.7 COM/COM+

In the Component Object Model (COM and COM+) [32] a component is an object residing in a COM server, which is either a dll or an exe file. In the following discussion the differences between COM and COM+ are of no importance, hence we use the term COM to collectively refer to both of them. COM objects implement a set of interfaces, each one containing at least one method. For example, in Figure 2.11, CompA implements Intrf1 and Intrf2, whereas CompB

![Diagram of COM/COM+ components and composition.](Figure 2.11: COM/COM+ components and composition.)

implements IntrfA and IntrfB, while both of them implement interface IUnknown. Actually, every COM component implements the IUnknown interface which relates to the COM component’s life cycle and additionally allows dynamic binding to component interfaces. Component interfaces are defined in the declarative Microsoft Interface Definition Language [9]. Then, with the help of a programming environment, for example Visual Studio [13], source code can be generated in different programming languages (for example C++, Delphi). After the developer implements the interface functionalities, the component is registered in the Windows registry using a unique identifier so that it can be dynamically located by other components.
Composition in COM occurs via interface pointers as shown in Figure 2.11. In the design phase, a component declares which component interfaces it will connect to. The actual interface pointers are bound at run time via the COM application server, as long as the referenced components are registered in the Windows registry. No composite components are formed – COM components simply call each other.

It might be argued that in COM composite components are defined, namely via component containment and component aggregation. In the follow discussion we show that neither of these terms defines composite components. In component containment, Figure 2.12(a), the outer component is the only component that holds interface pointers to its inner component instances. Other components/clients are still allowed to create different instances of the same component. In component aggregation, the outer component references the inner component, however some of the inner component’s interfaces are ‘exported’ to the outer component. In either case, the two components are developed and deployed independently. The outer component is not a composite component in the sense that it ‘contains’ the inner component, as the outer component can be deployed and distributed independently of the inner component. Instead, containment and aggregation are mechanisms for defining visibility aspects among COM components.

COM allows components to create new threads at the implementation level. Regarding synchronisation, COM uses the apartments model for synchronising access to components. In this model component clients are threads that access

![Figure 2.12: COM/COM+ containment and aggregation.](image-url)
COM components. These threads reside in different types of apartments. Components declare if they are thread safe or not\(^3\) and based on that, they can only be accessed from threads residing in compatible apartment types. But because a COM object can directly reference another COM object, see also Figure 2.11, it is possible for a thread to cross the boundaries of its apartment and (indirectly) invoke a COM object of an incompatible apartment type. For that reason, complicated rules need to be followed by component developers for ensuring correct concurrent behaviour.

2.3.8 .NET

In the .NET platform [111, 10] components are binary units called assemblies. Typical assemblies and their composition is shown in Figure 2.13. The assembly manifest contains metadata about the assembly itself. For example it contains its name, the files included in the assembly and references to other assemblies. Type metadata comprehensively describe all types that are in the assembly, which are implemented in Intermediate Language (IL) code. IL is an intermediate language into which all .NET languages (for example C# and Visual Basic) are compiled into. IL code is executed by the Common Language Run-time (CLR) which at run-time compiles the IL code into native, CPU and OS dependent, code. Finally, resources are external files needed for the execution of the assembly (for example image files).

\(^3\)Thread safety is implemented by the component developer using the facilities of the implementation language.
As shown in the figure, assemblies compose by directly referencing other assemblies. These references are defined in the design phase, when the assemblies are constructed using one of the .NET languages. Assembly referencing means that the IL code of one assembly can call directly methods of the IL code of another assembly. The physical form of an assembly is either a .dll or a .exe file and is deposited into a local or remote file system. After an assembly has been deposited into a repository, it is next executed in the CLR. The notion of a composite assembly does not exist.

.NET assemblies can be classified as application and library assemblies (.exe and .dll assemblies respectively). The difference is that the IL code of application assemblies executes on a separate operating system process, whereas the IL code of library assemblies executes on the invoking process, which is some other .exe assembly. Before classifying .NET components as active or passive, we recall for convenience the definitions of active and passive components given in Section 1.2. An active component may define its own concurrent processes and execute in them. A passive component on the other hand does not define its own processes, but it may be concurrently invoked by external concurrent processes. Therefore, a .exe assembly can only be classified as active as it does not need an external process in order to execute. This does not necessarily mean that the assembly is ‘actively’ doing something, as for example it may be idling while waiting to be invoked via some interprocess communication mechanism. Regarding .dll assemblies, their exact nature depends on how they are programmed internally. For example, a .dll assembly may, when initially invoked, create new threads that execute independently and continuously. From the above discussion it is obvious that in .NET, active and passive components are not explicitly distinguished and this distinction is in fact not important in the context of .NET.

With respect to synchronisation, .NET has eliminated the apartments model of COM and all components are considered to be executing in a multithreaded environment. As such, component developers need to ensure that components are thread safe. This can be done either manually by the component developers using the implementation facilities of .NET, for example semaphore and monitor objects, or automatically. Briefly, using the automatic way the component developer can declare the synchronisation behaviour required and the .NET run-time will enforce this. Automatic synchronisation is simple. The developer can declare
that instances (objects) of a class, which reside in the IL code of a .NET assembly, are synchronised. This means that only one thread can access this object at any time. The developer can also declare that multiple objects, which also reside in the IL code of an assembly, constitute a synchronisation domain. This means that a lock is used for all the objects, so that only a single thread can access objects in the domain. For more complicated synchronisation behaviour, a manual implementation is needed.
2.3.9 Javabeans

Javabeans [122] is a simple Java-based component model. Components in Javabeans are called beans. A bean is a plain Java class that has methods, properties and events. Methods provide the functionality of the bean, properties usually represent simple data of the bean, for example its name and its appearance (if it is a visual component), and events are generated by beans with the purpose of communicating with other beans. Beans can be developed in any Java IDE and are then packed into a JAR file. Beans are then loaded by a graphical tool, for example the Bean Builder [1].

Composition of beans occurs via events after the beans have been designed and before they start executing. Therefore beans compose at deployment phase as shown in Figure 2.14. In the figure there is a source bean and a target bean.

![Diagram of Javabeans composition](image)

**Figure 2.14: Javabeans and their composition.**

The source bean generates an event when ‘something’ happens. For example, if the bean is a visual component, let’s say a GUI element, it may fire an event when the mouse pointer is over it. Otherwise it may fire when some of its properties change. The target bean listens to these events through an event adaptor. The actual implementation of event adaptors is generated automatically by the bean builder when the component developer associates an event from a bean source with a method invocation into the target bean. After the beans have been composed, they start executing. The execution environment is a Java Virtual Machine.

In Javabeans all concurrency is defined and handled by the beans themselves. Beans can be active or passive entities, they can dynamically start the execution of multiple (Java) threads and a Javabeans can be concurrently invoked. In other words, the concurrent behaviour of a bean solely depends on how it is programmed internally. Consequently, synchronisation is similarly handled at the
As we explained in Section 2.2, composition forms an essential requirement for any component model and it should occur according to the idealised component life cycle. Regarding concurrency, both active and passive components should be defined as they are used for modelling different kinds of systems. Ideally, all concurrency should be defined at the modelling level where formal checks are possible, as opposed to the implementation level. In Section 2.3 we presented related software component models and we highlighted the aspects of composition and concurrency. In this section we categorise component models according to composition in the idealised component life cycle and we analyse them with respect to concurrency.

From the presentation in Section 2.3 it has become apparent that current component models only partially support composition according to the idealised component life cycle that was discussed in Section 2.2.1. Not all surveyed component models support design and deployment phase composition and some models do not even support the notion of a repository where components can be stored and retrieved for later composition. We summarise the surveyed component models in Figure 2.15, which has been previously presented in [84].

Figure 2.15: Categories based on composition (adopted from [77]).

Current component models can be classified into four categories. The first
category describes the models where components are developed and composed directly into a builder tool where there exists no notion of a repository. After components have been designed, they are implemented, usually with the help of a tool that generates skeletons for primitive, and composite, components. Components then execute inside the appropriate run-time environment. Architecture description languages, Pecos and Fractal fall into this category.

In the second category, components are developed inside a builder tool and are then deposited into a repository. Component composition occurs only when a new component is developed and no composite components are formed. This is because for models belonging into category 2, component composition occurs by components referencing other components via a unique identifier; components are still separate entities, and are deployed as separate entities into a deployment environment. In Figure 2.15, we collectively refer to the different deployment environments as the assembler tool, although no new composition is performed. After deployment, run-time follows, and component composition translates into direct method calls between components executing into appropriate run-time environments. The characterisation “Deposit-only repository” means that once a component has been deposited into the repository it can only be referenced by other, newly developed components and creation of composite components is not possible. To this category belong the EJB, OSGi, COM/COM+ and .NET component models. It is worth mentioning that the aforementioned component models actually create the component connections via object referencing at run-time. However, and as explained above, semantically components are composed at design-time when the component developer create a new component and declares which services of which other components then newly developed component will use.

In the third category, with sole representative Javabeans, components are deposited into the repository and composition occurs in the assembler. Composite components in Javabeans are undefined. In the last category, components are constructed in a builder tool and are then deposited into a repository. At that stage, components can be retrieved and used in further compositions by forming composite components. Composite components can be also stored into the repository where they can be (re-) used in further compositions. Components are then deployed into the appropriate deployment environment, which we refer to as assembler in Figure 2.15. In the assembler no new composition is possible. After
2.4. DISCUSSION

deployment, component instances start executing. SOFA is the sole component model belonging to this category.

The fact that current component models do not support the idealised component life cycle motivates us to develop a new component model. As will be described in Chapter 3, we have developed a component model that supports composition in both design and deployment phases.

Regarding concurrency, there are several observations that can be made. First, in some component models all components are exclusively considered as active entities (ADLs). In contrast, in EJB all components are passive. From the rest of the component models only Pecos explicitly distinguishes between active and passive components, since in SOFA, Fractal, OSGi, COM, .NET and Javabeans the exact nature of a component depends on its implementation.

In terms of formalisation and of the checks that can be performed on a system built from components, Darwin and Wright ADLs have used process algebras for defining and analysing the concurrent behaviour of a system. Behaviour protocols in SOFA and in Fractal have been used for defining and checking protocol conformance among composed components. However, in that work, checks for common concurrency errors for atomic components, like deadlock or livelock, are not defined. In the rest of the component models, no formalism has been used for specifying their concurrent behaviour.

We notice that the surveyed component models cover the whole range in terms of active and passive components and in terms of formal checks that can be performed at the modelling level. As we explained earlier, it is our belief that both active and passive components are needed for defining different kinds of component-based systems, and actually it may be necessary for a single system to combine both types of components. In addition to that, concurrency should be defined at the modelling level. These requirements and the fact that there is not a single component model covering them, have driven us into the definition of the concurrency aspect of our component model. As we shall show in Chapter 3, in our component model we define and explicitly distinguish between active and passive components. We have used a process algebra for describing the behaviour of active components so that formal checks can be conducted. Composition of active and passive components is also possible according to the idealised component life cycle.
2.5 Chapter summary

In this chapter we surveyed the definitions for a software component and we realised that it is only meaningful that a software component is defined as part of a software component model. We subsequently adopted a definition for a software component model that we will follow for the rest of this thesis. Then we explained what the desiderata for a software component model are. In terms of composition, these are reflected into the idealised component life cycle. In terms of concurrency, it means that both active and passive components are modelled and that some formal notation is used for checking against concurrency errors. The above desiderata were used as the referencing framework while presenting related work. In the discussion that followed, we summarised the related work and briefly motivated some features of our component model.

In the next four chapters we explain our component model. We start in Chapter 3 with an overview of our model and in the following three chapters more details are given. Finally, in Chapter 7 the implementation of our model is presented.
Chapter 3

Our component model

In this chapter we present an overview of our component model. As we mentioned earlier, our component model is based on previous research conducted by Lau and colleagues [83, 81, 82, 77, 76, 84]. Since our work builds on them, some of the features of our model inevitably constitute parts of previous works. Specifically, the ideas of exogenous connection, separation of computation from control, hierarchical composition, the idealised component life cycle and connector composition have all been earlier introduced. It is their refinement and extension into novel directions that constitute the contributions of this thesis. More specifically, we have introduced concurrency into the model by defining active atomic and composite components and concurrent connectors. Previously, single-threaded only execution was permitted, and the revised model allows and caters for multi-threaded execution. We have additionally formalised the component type system in Z [121], we have integrated design and deployment phases into a single modelling and implementation framework and finally, we have thoroughly investigated connector composition. For precisely defining the semantics of connector composition we have introduced formal, Coloured Petri net semantics [73]. In the rest of the thesis we try to avoid repeating the differences between the “old” and the “new” model, except when there is risk of confusion.

This chapter is structured as follows. In Section 3.1 we discuss exogenous connection, which forms the basic principle behind our connectors. Next, in Section 3.2, we describe the basic elements of our model, namely components and connectors. Component composition is presented in Section 3.3 where more details on the idealised component life cycle are given. Naturally, connector composition follows in Section 3.4 and the connector life cycle is discussed and
related to the component life cycle. Concurrency forms an important aspect of our model and the concurrency principles of our model are introduced in Section 3.5. Section 3.6 concludes this chapter.

### 3.1 Exogenous connection

In general, components communicate via direct message passing, indirect message passing, or by exogenous connections. These approaches are shown in Figure 3.1. In direct message passing, components communicate by sending a targeted message from one component to another directly, Figure 3.1(a). This is the approach followed when components are objects, or collections of them, and communication is achieved by direct method calls. For example this is how components communicate in EJB [41], OSGi [19], COM [32] and .NET [10]. This form of communication is also used in the Fractal [58] component model, and in the Darwin ADL [89]. In the aforementioned cases there are no explicit connectors.

In indirect message passing, components initiate communication by sending a message to a shared medium (connector), and it is then either the connector that passes this message to the appropriate component, or another component that requests to read the message left on the connector (Figure 3.1(b)). Communication via events like in the Wright ADL [18] and in Javabeans [122], via shared data spaces like in coordination languages [63, 106] and via shared channels like in the C2 ADL [125] and in the Pecos component model [108] fall into this category.
the SOFA [39] component model a range of explicit connectors connectors exist
that model different kinds of communication, for example (possibly remote) pro-
cedure calls, or bus communications. In the aforementioned component models,
all communications are initiated by components, and via connectors, messages
are directed to other components.

Using connection by message passing, both direct and indirect, connected
components initiate communication to another component, so their connection
affects control flow, and possibly data flow, as well as computation. Connected
components are thus tightly coupled, albeit to a varying degree depending on
the level of indirection in the message passing; and control and computation are
mixed up. In order to minimise coupling, and to maximise separation of control
from computation, it is necessary to control the components *exogenously*, i.e. from
outside of the components. The merits of exogenous connection and control have
been recognised in the Reo coordination language [23] and by the community that
combines the CSP and B-machines formalisms [117, 45]. In our approach [83]
we use connectors that encapsulate control and data flow between connected
components. The idea is that in connected components, the connectors, rather
than the components themselves, initiate method calls in the components, so that
any control flow between the components is encapsulated by the connectors. This
is illustrated in Figure 3.1(c). In such a scheme, connected components react to
their connector only, and not directly with each other. In other words, connectors
are the only means of communication among components and all communication
is initiated by the connectors. Components encapsulate computation, while the
connectors encapsulate control. For example, in Figure 3.1(c), the connector *Con1*
sends messages to components *A* and *B*, but *A* and *B* do not directly interact with
each other. Method calls may be accompanied by data flow between *A* and *Con1*
or between *B* and *Con1*, but again not between *A* and *B* directly.

3.2 Components and connectors

We have shown that our model is fundamentally different from most other com-
ponent models in terms of how inter-component communication is achieved: con-
nectors initiate method calls on components and coordinate all control and data
flow between them. We proceed by describing in detail the basic elements of our
model, namely connectors and components.
Components in our model are the basic unit of computation. We have defined three kinds of components, atomic, composite and adapted components. Atomic components are shown in Figure 3.2(a) and are either passive or active. A passive atomic component consists of a computation unit $CU$ and an invocation connector. A computation unit provides a set of methods (or services) and it encapsulates computation. Encapsulation at the level of the computation unit means that $CU$’s methods do not call methods in other computation units; rather, when invoked, all their computation occurs in $CU$. The invocation connector is an exogenous connector that is connected to a computation unit $CU$ so as to provide access to the methods of $CU$. The invocation connector encapsulates control, which means that it is the only entity with access to the computation unit. The passive atomic component is a well encapsulated entity, meaning that it can be accessed only through its interface, which “hides” both the computation unit and the invocation connector. Because it is passive, the passive atomic component does nothing on its own; instead it waits to be called (through composition operators, see below) in order to perform some computation.

Active atomic components consist of a set of active computation processes $ACPs$ and a channels connector $Ch$. The active computation processes execute continuously on the own thread of control and periodically interact with the channels connector so that their control signals interact and data i/o is performed. The channels connector provides the interface to the $ACPs$ and provides the means through which they can interact to other components – via composition operators,
see below. Similar to the CU, ACPs encapsulate computation because all their computation is performed internally, without interaction with other components, or ACPs in other (active atomic) components. Similar to the passive atomic component, the active atomic component is a well encapsulated entity because it can only be accessed through its interface. Finally, the active computation processes are actually active CSP processes executing in parallel. CSP [70] is a process algebra used to model concurrent systems. The use of CSP gives great expressive power to our model for defining concurrent systems. Additionally, due to the formal, mathematical underpinnings of CSP it is possible to perform some automated checks for the detection of common concurrency problems, like deadlock [5].

Atomic components may be composed into composite components using explicit composition connectors or composition operators. A composition connector is exogenous and encapsulates control. It is used to define and coordinate the control and data flow for a set of components. A generic composition is shown in Figure 3.2(b), where the composition connector \( C \) connects two atomic components \( \text{Comp1} \) and \( \text{Comp2} \) into a composite. The two atomic (sub-) components can be both active, both passive, or a combination of them. The result of the composition is a composite component. The control flow of the composite component is defined by the composition connector \( C \). In our model we have pre-defined a number of generic composition operators that describe different forms of control and data flow. The interface of the composite component depends on the composition connector used. Composite components can be either active or passive. This depends on the type of the composition operator and of the sub-components (active or passive).

Composition using explicit composition operators preserves encapsulation. This means that the encapsulation of atomic components is not violated during composition. Furthermore, connector \( C \) shown in Figure 3.2(b) can only access the sub-components of the composite \( \text{Comp1} \) and \( \text{Comp2} \). Therefore the control defined by the composition connector \( C \), is encapsulated within the composite component. The composite component is only accessible via its interface. Constituents sub-components may or may not be available after composition (\( \text{Comp1} \) and \( \text{Comp2} \) in the figure). This depends on whether composition occurs in the

\[\text{For the rest of this discussion we will use the terms composition connector and composition operator interchangeably.}\]
design or in the deployment phase of the component life cycle. Design and deployment phases were introduced in Section 2.2.1. More details on this issue given in Section 3.3.

As an example of the control flow that can be defined by a composition operator, we discuss the pipe. The pipe is n-ary, meaning that it composes n components $C_1, \ldots, C_n$. It calls methods in $C_1, \ldots, C_n$ in that order and passes the results of calls to methods in $C_i$ to those in $C_k$, $k > i$. Another example of a composition operator is the cobegin. The cobegin defines concurrent control and it invokes components $C_1, \ldots, C_n$ in parallel without passing any data among the concurrently executing components. All composition operators defined in our model are presented in detail in sections 4.5 and 4.6.

Finally, and as shown in Figure 3.2(c), we include in our model a special class of unary connectors called adaptors. The purpose of adaptors is to adapt a component for use in a specific system. They express unary form of control, for example a guard adaptor may block or not the incoming control flow based on a boolean condition. Similar to the other component types, adapted components encapsulate control.

We proceed by giving a formal or mathematical definition of our components in order to unambiguously define what a component is in our model. After providing a formal definition we elaborate on component composition and we relate it to the idealised component life cycle that was discussed in Section 2.2.1.

### 3.2.1 Formal definition

Having informally introduced components and connectors, we may proceed and present their formal definitions. The type system is given in Z [121] and a short introduction to Z is given in Appendix A.

Purpose of the Z formalisation is to provide unambiguously the syntax of our component model. The formalism describes the class of all possible systems conceivable, i.e. the buildable systems for any given repository. Our type system does not distinguish between components in the design, deployment and runtime phases of their life cycle. Although these distinctions are important for a component model, they do not fall within the scope of our Z formalisation. In other words, the entities of the type system, be it connectors or components, refer to entities that are in any phase of their life cycle. Distinctions between component-life cycle phases are made informally and are discussed whenever the
related points arise. In the rest of this section we provide the basic notions for components and connectors.

In our type system, the given sets include basic types, connectors, logical predicates, systems that are built using our model, variables and data values:

\[
[BASIC\_TYPES, CONNECTOR, LOGIC\_PRED, SYSTEM, VARIABLES, DATAVALUES]
\]

The basic types include, strings, integers, and so on. For the purposes of this presentation it suffices to consider that the basic types only consist of strings and integers:

\[
\text{STRING, INT : } \mathbb{P} \text{ BASIC\_TYPES}
\]

\[
\langle \text{STRING, INT} \rangle \text{ partition BASIC\_TYPES}
\]

Given sets represent the types that cannot be decomposed into other types. We have also considered connectors, logical predicates and systems to be given sets in our formalisation. Of course, these types have internal structure and their own semantics. However, given the narrow scope of our Z formalisation, considering them as non-decomposable types represents the correct level of abstraction. Finally, variables and data values will be used later when we define a component and its data.

A component is a composite type that has a name, a connector, a level in the hierarchy, a non-empty interface (a set of interface elements), some data and a synchronisation constraint. This is captured by the Z schema in Figure 3.3. In the figure we present only the variables of each schema and we have omitted schema invariants. Schema invariants are subsequently discussed informally. In the rest of the thesis, we follow the above presentation scheme whenever formal Z definitions are presented.

For the schemata presented in Figure 3.3, the following invariants hold:\(^2\)

- The name of a component is used for uniquely identifying a component.
- Each component is uniquely associated with a connector and a level. The level of every component is at least one, where one is the level of atomic components.

\(^2\)The type system, including all invariants, are enforced by the implementation.
CHAPTER 3. OUR COMPONENT MODEL

COMPONENT

name : STRING
conn : CONNECTOR
level : \mathbb{N}_1
interface : \mathbb{P}\text{InterfElem}
data : DATA
scs : \mathbb{P}\text{SyncConstraint}

InterfElem

name : STRING
input : seq(\text{BASIC\_TYPES})
output : seq(\text{BASIC\_TYPES})

DATA

vars : \mathbb{P}\text{VARIABLES}
vals : \mathbb{P}\text{DATAVALUES}
evaluate : \text{VARIABLES} \rightarrow \text{DATAVALUES}

Figure 3.3: Definition of a component in Z.

- The interface of a component is a set of at least two interface elements. One element is the nilIE that denotes the empty interface element, and when it is invoked it does nothing. It is used during composition, Section 4.5, and it is defined such that its input and output sequences are empty.
- Each interface element consists of a name, a (possibly empty) sequence of input types, and a (possibly empty) sequence of output types.
- No two interface elements may have the same name.\(^3\)
- Component data consist of a set of variables and data values, and of an evaluation function that associates variables to values. Because our components can be accessed by concurrent processes, synchronisation constraints are necessary for these data. Details on how concurrency is defined and controlled in our model, are given in Section 3.5, and the type equations for synchronisation constraints are given in Figure 4.8.

\(^3\)We introduce this restriction to simplify the type equations of our model. This is obviously a minor point and does not affect the systems that can be built using our model.
Every component has its own set of (data) variables \textit{vars}. This is the definition of \textit{data encapsulation} in our model.

Every component has a different connector. This means that every component defines its own control flow.

The four different component types shown in Figure 3.2 can be expressed formally by partitioning components into different subsets, each one denoting a different component subtype. Although Z does not support sub-typing, we use this term to emphasize the fact that different connectors are used differently in component composition, yielding different “kinds” of components. For example atomic, adapted and composite components are all component “sub”-types. In Z, this is captured using sub-set relationships.

The partitioning of components into the possible sub-types is shown in Figure 3.4. In the figure, four sub-types of components are defined: passive atomic

\[
\begin{array}{l}
P_{\text{ATOM}}, A_{\text{ATOM}} : \mathbb{P} \text{COMPONENT} \\
\text{COMPOSITE, ADAPTED} : \mathbb{P} \text{COMPONENT} \\
\langle P_{\text{ATOM}}, A_{\text{ATOM}}, \text{COMPOSITE}, \text{ADAPTED} \rangle \\
\text{partition} \text{COMPONENT}
\end{array}
\]

Figure 3.4: Different component types in our model in Z.

components, active atomic components, composite components and adapted components. The following invariants hold:

- The level of atomic components is always one.
- Composite components are derived from the composition of other components with explicit composition operators as was shown in Figure 3.2(b).
- Adapted components are derived by applying a unary, adaptation connector to a single component, as was shown in Figure 3.2(c).
- Consequently, the level of composite and adapted components is always greater than one.
- Components are uniquely identified by their names.
- Finally, it is not possible to have composite and adapted components, unless primitive components have been defined.
3.3 Component composition

A basic characteristic of our approach is compositionality of components and connectors. Compositionality of components is shown in Figure 3.2(b) and conveys the basic property of our components that they can be composed via composition operators. In this section we discuss component composition in the light of the idealised component life cycle that was presented in Section 2.2.1. Compositionality of connectors is discussed in Section 3.4.

According to the idealised component life cycle, the ‘life’ of a component begins in the design phase, where it is being designed and implemented in perhaps more than one languages. The design phase corresponds to the development of components that are re-usable entities. Designed components are stateless. They obtain state when they get deployed into an execution environment. The deployment phase corresponds to development with components, where components are ready-to-execute software entities. By instantiating them with appropriate memory and scheduling resources, or any other resources deemed necessary, components transit into the run-time phase of their life cycle where they start executing.

We have argued that components should be composed into composite components in both phases as this maximises re-use. The purpose of composition in the design phase is to construct components that are sufficiently generic to be re-used in further compositions and that can be deployed in different deployment environments. In the design phase whenever a composition operator is applied to a set of components, a new composite is created and is automatically stored to the repository. In our model, the constituent components of the composite are copies of the components that they consist of. For example, in the repository in Figure 2.1 composite BC consists of copies of both B and C. Using copies has two consequences. First, components B and C can be used in different compositions. Second, component copies are completely independent from the component they originate from. Their internal structure, including source code is copied into a new component. Consequently, when B and BC get deployed and later on they get executed, they are completely independent and the execution of one does not affect another. In other words, design phase composition results to composite components where at each level of their composition there is a composite component. This is shown in Figure 3.5.

In the figure it can be seen that whenever a composition (or an adaptation)
3.3. COMPONENT COMPOSITION

Figure 3.5: Typical composition in design phase using our model.

operator is applied, a composite (or adapted) component is formed whose internal structure is hidden. We have used a rectangle with a solid border for that purpose. Constituents sub-components are considered inaccessible. A composite component is a well encapsulated entity that is accessed only at its interface level. The interface is defined by the composition operator and different operators define different interfaces.

In Figure 3.5 the hierarchical nature of our component model is also presented. The hierarchy results from our components being compositional, i.e. the result of a composition can be further composed. Components and connectors have levels. Components at level 1 are atomic components and only the invocation connector and the channels connector can be level 1 connectors, see also Figure 3.2(a). As can be noticed in Figure 3.5, hierarchies do not have to be symmetric. For example, the level 4 connector in the figure composes a level 3 and a level 2 component.

The focus so far has been on composite components and their compositional nature. However, components in our model can be atomic, composite or adapted, Figure 3.2. Adapted components are also compositional, because at their interface level they are similar to other components. In Figure 3.5 an adapted component of level two is also used. Adapted components and adaptation connectors are more likely to be used during the deployment phase. This is because in the deployment phase components are deployed and composed, having specific systems in mind. Hence component adaptation is more meaningful during the deployment phase. This does not mean that adaptation cannot occur in the design phase, but instead it means that adaptation is more ‘natural’ in the deployment phase.

A typical system built in the deployment phase is shown in Figure 3.6. As can be seen in the figure the component hierarchy is retained, but it is not as
straightforward as the one in Figure 3.5. Composition still occurs, but the result is not a composite that can be deposited into a repository. Instead, it should be viewed as a temporary composite, or subsystem until the whole system is built. As a result, whenever a composition operator is used, it is no longer necessary to create a ‘perfectly’ encapsulated composite component whose internal structure is hidden. Composites in the deployment phase have their internal structure visible. The composite can be still used in composition, but additionally the sub-components can be used in other compositions. In other words, in deployment phase ‘sharing’ of sub-components is possible.

The above difference between the design and deployment phases i.e. whether sub-component sharing is possible, is the major difference between design and deployment phase composition in our model. Other than that, there are a number of more subtle differences. For example, adaptation connectors are more suitable for the deployment phase but they can still be used in both phases. More importantly, we have defined a special class of connectors that can be top-level connectors only, see Section 5.2.1. These connectors can only be applied during the deployment phase and they are used to create systems.

We mentioned earlier that our formal semantics in Z do not capture the component life cycle. However, and as is shown in Figure 3.3, components contain data. Existence of data explicitly differentiates between the components in the two phases. The design phase is independent of any programming language, and component templates do not hold any data. Regarding data, and to be more precise, components in the design phase may declare a set of variables, Figure 3.3, but all these variables evaluate to nil. In other words, component data in the design phase simply provide placeholders for the actual data that are instantiated
3.4 Connector composition

In our component model connectors are control structures that compose components into composite ones. In this section we demonstrate that it is possible to compose connectors into composite connectors. Composite connectors are larger control structures that can be used in further connector composition or they can be used for composing components into composites. In this section we provide a brief introduction to the topic and a full treatment is provided in Chapter 6.

In Figure 3.7 we replicate from Figure 2.1 component composition as it occurs according to the idealised component life cycle. We extend this figure to include the connector life cycle and show how these two life cycles relate to each other. As can be seen in Figure 3.7, we have introduced a repository of connector templates. As already explained connectors in our model represent control and communication structures used for inter-component communication and composition.

Connector templates are stored in the repository. Connector templates are generic control structures defined independently of the components or connectors when the component gets deployed.
that they can be used to compose. Similar to the component builder tool, there is a connector builder tool that retrieves connectors from their repository and composes them into composite connectors. During connector composition a connector is used as an operator, \( C_2 \) in the figure, and a sequence of connectors, \( C_1 \) and \( C_3 \), are used as operands. The result is a composite composition connector, \( C_4 \), that is stored in the connector repository.

Whenever a connector template is used as an operator during connector or component composition, it needs to be specialised for the specific composition. If its arity is not fixed, then this is done. For component composition, some additional specialisation is needed. As we mentioned earlier, whenever a connector is used for component composition, a composite component is formed whose interface depends on the interfaces of the sub-components and on the control flow of the composition operator. The information that relates the interfaces of the sub-components to the interface of the composite component is stored in the connector. It is the inclusion of this information that differentiates a connector used for component composition from a connector used for connector composition. In Figure 3.7 different symbols are used to emphasize this difference.

In the figure, a binary ‘version’ of connector \( C_2 \) is obtained for composing two connector templates in the connector builder. Additionally, a binary instance of \( C_2 \) is obtained for deployment phase component composition. Connector instances are used for component composition, whereas only connector templates are used for connector composition, and their difference was explained above. Similarly, binary instances of \( C_4 \) are used for both design and deployment phase compositions.

As a concrete example of connector composition we present the composition of a pipe with a cobegin. It has been shown in [77] that composite connectors are design patterns [61] and they provide powerful composition operators. By composing a pipe with a cobegin (Figure 3.8) we obtain the observer pattern.

![Figure 3.8: The observer design pattern as a composite composition connector.](image-url)
After reading a value published by the publisher component, the pipe transfers this value to the cobegin connector, which notifies all subscribers concurrently.\textsuperscript{4}

In the connector builder tool a binary pipe and an \( n \)-ary cobegin are composed to form the \((n + 1)\)-ary observer. The observer connector template can be used for connector and component composition. Regarding component composition, it can be used for both phases of the component life cycle. Whenever it is used as a component composition operator it must be made specific for that composition by fixing its arity. For example, it is not possible to create the composite shown in Figure 3.8(b) where the number of subscribers is not fixed. The figure only shows that there can be any number of subscribers and a design or deployment phase composite will have that number fixed.

\section*{3.5 Concurrency at the model level}

An important feature of our component model is that it defines concurrency at the modelling level in terms of components and connectors. Before we describe how our component model supports concurrency, we introduce the terminology that we will adopt for concurrent processes.

A process is a sequence of actions,\textsuperscript{5} and concurrent processes allow simultaneous execution of two or more actions. Actions are basic computational tasks that are atomic and indivisible. Broadly speaking, there are four kinds of exe-

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figures/concurrent_processes.png}
\caption{Concurrent processes.}
\end{figure}

\textsuperscript{4}In \cite{77}, the observer pattern is defined as a composition of a pipe with a sequencer, which is slightly inaccurate.

\textsuperscript{5}A process may have finite or infinite states; for simplicity but without loss of generality we will consider only finite-state processes.
cution for concurrent processes: (i) parallel processes; (ii) interacting processes; (iii) branching processes; and (iv) joining processes (Figure 3.9).

*Parallel* processes execute completely independently from one another; their actions do not interact at all. *Interacting* processes interact via synchronisation between their actions. Synchronisation imposes a temporal ordering among the actions of the concurrently executing processes. The direction of the dotted arrow in the figure denotes that the action where the arrow points to cannot start until the action at the other end of the arrow occurs first. For example, action \( d \) of \( P_2 \) cannot start until action \( a \) of \( P_1 \) starts, while actions \( s \) start and end concurrently, therefore they occur concurrently and an external observer notices a single action \( s \). We use the first type of interaction to denote indirect synchronisation among concurrent processes, usually shared-data synchronisation, for example synchronisation via semaphores or monitors. We use the second type of interaction, i.e. on action \( s \) between \( P_1 \) and \( P_2 \) to indicate direct interaction among the participating processes. In this case the processes may communicate data values directly to each other. This type of interaction is typically the case for processes interacting via shared channels.

A *branching* process (Figure 3.9(iii)) splits into two sub-processes, whereas a *joining* process joins two sub-processes (Figure 3.9(iv)). Branching and joining could involve more than two sub-processes, but for simplicity we consider only two, i.e. we only consider the usual *fork* and *join*.

A component model defines components and their composition mechanisms [84, 68]. Therefore, in a component model, components and composition mechanisms are the sources of concurrent processes. Components may define their own processes and execute on them, and these processes may be concurrent. These are the *active* atomic components that were shown in Figure 3.2(a) executing on their own active computation processes.

On the other hand, a *passive* component does not define its own processes, but its interface may allow concurrent invocations of its methods; such invocations are actions in concurrent processes. In our model all our connectors support concurrent processes, by means of concurrent invocations. Because our components contain data, Figure 3.3, concurrent invocations raise synchronisation issues, specifically when the invocations access the same data. For this reason components contain a set of *synchronisation constraints* that describe how concurrent processes may invoke a component. Synchronisation constraints are a
form of concurrency control and since control in our model is defined and handled by connectors, it is the connectors that enforce synchronisation constraints for components.

Apart from the means of concurrent invocations, our composition operators support other means of concurrent processes. For example, the cobegin connector mentioned earlier branches each incoming process into \( n \) sub-processes, where \( n \) is the number of connected sub-components and when the sub-components finish executing it joins them into a single process. In chapters 4 and 5 we show precisely how concurrent processes are defined in our model. Being part of the model, concurrency control is also explained. In essence, all concurrency is defined at the component model level which constitutes an advantage of our approach. This is because several checks regarding safety and liveness can be performed at the abstract, model level. Instead, when concurrency is a property of the implementation language and not of the model, these checks cannot be performed, and there is always the risk that the implementation violates some properties of the model.

3.6 Chapter summary

In this chapter we provided an overview of our component model. Its principles were introduced, namely structured, hierarchical composition, separation of control from computation, and encapsulation of control and computation in connectors and components. The basics of our type system were presented by defining in Z the type of the generic component. In our component model, concurrency is defined at the modelling and not at the implementation level. It is expressed by explicit constructs that define concurrency, specifically active computation processes and concurrent connectors. Since concurrency is defined in the modelling level, it must also be controlled at the same level. This is the reason for introducing synchronisation constraints.

Furthermore, our component model complies to the idealised component life cycle that was introduced in Section 2.2.1. In our model, composition in both design and deployment phases is supported, and there are features that explicitly distinguish them. Finally, the fact that we have separated control from computation via encapsulating control in explicit connectors, has enabled the definition of compositional semantics for connectors. In this chapter, composite connectors,
and their life cycle, have been introduced.

In the following chapters we present the above features of our model in detail. We start in Chapter 4 by presenting atomic components and their composition. Concurrency is also introduced.
Chapter 4

Atomic and composite components

In Section 3.2 we briefly presented that in our component model there are three types of components: atomic components, composite components and adapted components. In this chapter we focus on atomic and composite components. Component adaptation mostly ‘fits’ in the deployment phase of the idealised component life cycle, and is discussed in detail, along with component execution, in Chapter 5.

Starting from atomic components, either active or passive, composite components can be built using explicit composition operators. In this chapter we explain in detail active and passive atomic components and how they are composed into composite ones. Through this discussion, we explain how concurrency is defined, and handled, by both atomic and composite components.

4.1 Passive atomic components

Passive atomic components were shown in Figure 3.2(a). We now present a more detailed view of them. Figure 4.1 shows the structure of a passive atomic component and a typical execution. A passive atomic component does not define its own executing process. It consists of a (passive) computation unit and an invocation connector. The computation unit contains a set of operations, $m1()$ and $m2()$ in Figure 4.1, and performs them when invoked by the invocation connector. The invocation connector provides the interface for the component and is activated by processes from outside the component. It is the external
processes that access the computation unit and force the computation to take place.

These processes are typically initiated by external users of the system and without them a passive component does nothing. They are usually parallel, as in Figure 3.9(i). As will be discussed later, occasionally these processes may access shared component data and therefore synchronisation is required, as is the case between actions $a$ and $d$ in Figure 3.9(ii).

In Figure 4.1, three processes $P_1$, $P_2$ and $P_3$ access the atomic component concurrently, the first two invoking $m_1$ and the third one invoking $m_2$. As shown in the figure, for each method invocation we consider that the only actions of interest are the input, the execution and the output of the method. Therefore the process executing the method, say $m$, will execute $m.in \rightarrow m.exec \rightarrow m.return$, where $\rightarrow$ is read ‘then’. Although these processes are independent, they may need to synchronise before actually executing a method (because they may contain their own data [82], as also shown in the figure). For example, $P_1$ and $P_2$ synchronise before executing method $m_1$, and $P_2$ may not execute $m_1$ unless $P_1$ executes $m_1$ first. In order to describe the synchronisation requirements for a component, we have introduce synchronisation constraints which are presented in detail in Section 4.2.

**Z type system**

We have formalised in Z the computation unit and we have provided a function that defines the relationship of atomic components to invocation connectors and computation units. Before defining how atomic components are created, we need to define computation units.

A computation unit defines a non-empty set of operations, which are exposed by the invocation connector at the interface of the atomic component. Each operation is defined as in the usual definition of methods in programming languages.
4.1. PASSIVE ATOMIC COMPONENTS

like Java. Therefore operations of computation units have a name, a sequence of
input parameters and at most one result or output parameter:\footnote{Although in modern
programming languages input and output parameters can be of more
complicated types, such as arrays or arbitrary objects, we have limited our approach to
basic types only. An implementation of the model can certainly overcome this limitation. Our basic
implementation only supports basic types.}

\[
\begin{array}{l}
\text{Operation} \\
\quad \text{name : STRING} \\
\quad \text{inputParams : seq(BASIC\_TYPES)} \\
\quad \text{outputParams : } \mathbb{P} \text{ BASIC\_TYPES} \\
\quad \# \text{outputParams} \leq 1
\end{array}
\]

The Z schema for computation units is shown in Figure 4.2. The following

\[
\begin{array}{l}
\text{CU} \\
\quad \text{name : STRING} \\
\quad \text{operations : } \mathbb{P}_1 \text{ Operation} \\
\quad \text{constraints : } \mathbb{P}_1 \text{ SyncConstraint} \\
\quad \text{association : } \text{Operation} \mapsto \text{SyncConstraint}
\end{array}
\]

Figure 4.2: Definition of a computation unit in Z.

constraints on the variables of the computation units hold:

- computation units have unique name, i.e. no two computation units share
  the same name,

- the non-empty sets of operations and synchronisation constraints, are re-
  lated by the \textit{association} function that maps every operation of the compu-
  tation unit, to a synchronisation constraint (see Figure 4.8) that has the
  same name with that operation.

In order to relate an invocation connector, with a computation unit and a
passive atomic component, function \textit{createAtomic} is introduced that “creates” a
passive atomic component out of a computation unit and an invocation connector:
Certainly in Z, and in set-theory in general, a function does not “create” result(s) in the same sense that functions in programming languages do. Instead, they describe the relationship between members of different (or the same) sets. We nevertheless name this function `createAtomic` because this formalisation guides the implementation, and at the implementation level we create an atomic component from a computation unit and an invocation connector. We are using the same naming convention throughout the thesis.

Function `createAtomic` uniquely associates pairs of invocation connectors and computation units to atomic components. This function is partial because in each atomic component a different invocation connector and a different computation unit is used. This restriction does not mean that there can not be computation units offering the same operations – it means instead that the two computation units must have different names. Therefore, a pair of an invocation connector with a computation unit can define at most one atomic component. The following constraints for function `createAtomic` hold:

- All atomic components are created by using the `createAtomic` function.
- The interface of the atomic component is generated from the operations of the computation unit. Precisely, all operations of the computation unit are mapped using an one-to-one fashion to interface elements of the atomic component.
- The synchronisation constraint of the atomic component is that of the computation unit. As will be explained in Section 4.2, this is because the invocation connector reads and enforces the synchronisation constraints of the computation unit.

As with all constraints of the Z schemata, the model’s implementation is responsible for enforcing them. For example, whenever a computation unit is associated with an invocation connector with the purpose of defining a passive atomic component, copies of them are used with unique names. This is in order to guarantee the uniqueness of computation units and invocation connectors demanded by the constraints.
Coloured Petri-net semantics

The Z formalism represents the static structure of components and connectors and describes their interface. However, the Z formalism does not explain the control flow of atomic components. For that reason we have introduced the CP-net formalism [73]. CP-nets is a modelling language typically used to describe concurrent and distributed systems. CP-nets offer a graphical notation for modelling concurrent systems. In this thesis we use CP-nets to unambiguously model the control flow in connectors and components in our model. CP-nets are also particularly useful for modelling the control flow of composite connectors and this is further discussed in Chapter 6. In Appendix B we provide a detailed introduction to CP-nets. Here we present the CP-net for passive atomic components and briefly discuss the CP-net semantics.

A passive atomic component is represented as the simple CPN shown in Fig. 4.3(a). It consists of an input place \( In \), an output place \( Out \) and an execution place \( exec \). Places in CP-nets have types and for the case of the passive atomic component they are typed \( N \times CID \), denoting pairs of natural numbers with case ids which are integers. The case id uniquely identifies a user request and we consider each user request to have a different case id. Places can hold tokens of the respective type. Therefore a pair of a natural number and a case id is used to identify the process id that actually serves the user request. Two token examples are \((0, 1)\) and \((3, 100)\) denoting two process ids of user requests 1 and 100. Transitions \( T1 \) and \( T2 \) are responsible for moving tokens between the three places \( In \), \( Out \) and \( exec \). The movement of tokens around places represents the control flow of a component.

![Figure 4.3: Control flow of the passive primitive component.](image-url)
As the component is passive, it waits for control to arrive via a composition connector in place $In$. Then the control reaches the $exec$ place which means that the computation unit is executing. Finally, when execution finishes, transition $T2$ fires and control returns to the $Out$ place. The name of the places correspond to the events generated whenever an external thread invokes a method, Figure 4.1. For example, occurrence of the event $m1.in$ means that the control process has reached place $In$. Concurrency is modelled by allowing multiple tokens in several places, therefore transitions $T1$ and $T2$ may fire concurrently.

The $exec$ place represents an abstraction of the actual computation unit. When modelling a computation unit and its operations at a more concrete level, the abstract view provided in Figure 4.3(a) does not suffice. For example, if we consider a computation unit that has two operations, $m1$ and $m2$, the more detailed view of the computation unit shown in Figure 4.3(b) is needed. The actual methods are represented by transitions. This is because transitions in CP-nets “do something” in the sense that they can transform input tokens to different output tokens. Thus transitions can be used to model computation and therefore we use them to model the actual operations of the computation unit.

In order to model an atomic component with the aforementioned computation unit, place $exec$ in Figure 4.3(a) will need to be replaced by the detailed CP-net in Figure 4.3(b). The replacement is straightforward. Namely, the destination of the outgoing arc of transition $T1$ will be re-directed to place $InCU$. Conversely, the source of the input arc of transition $T2$ will be re-directed to place $OutCU$, a single CP-net will be then formed.

Figure 4.3 introduces some conventions that are used throughout this thesis when drawing the CP-nets of components and connectors, and are explained below. Firstly, in the CP-net representation we deliberately omitted the data flow, as well as the actual data values that are ‘carried’ by the control threads. In our model, control ‘carries’ the i/o data for components, therefore data and control flow coincide. Regarding actual data i/o values, we omit them from our figures because the purpose of using CP-nets is to show the control flow in our components/connectors in an unambiguous and concise way. Data i/o values have to do with the actual computation and (usually) they do not affect control flow. Whenever data i/o affects the control flow this is clearly explained in the accompanying text.

The second convention followed is that each component and connector consists
of a single input $In$ and a single output $Out$ place. As will be shown later, in our model it is possible for multiple processes to concurrently serve a user request via branching processes, see Figure 3.9(iii). When a single process is branched into many processes, the sub-processes share the same case id but different natural number, so that they can be distinguished. In other words the pairs $(i, c), \ i \in \mathbb{N}, \ c \in CID,$ represent control tokens, and their flow in a given CP-net starting from the place $In$ to the place $Out$, represents the control flow of the connectors, or components, which is the behaviour of the process executing that connector, or component. The presence of tokens in places collectively, designates the state of the connector or the component.

The third convention followed, regards the initial assignment of tokens to places. As has been explained earlier, all our components support concurrent processes. To reflect that in the CP-net semantics, means that in the input place $In$ there can be multiple pairs, each one denoting a different executing process. However, since these processes may be only passed to the $In$ place by a higher level connector, initially the place is empty. Control processes are actually initiated by the top level connector of a system, and this is the only case where there are tokens in the $In$ place of a connector. These are the control tokens that via composition, flow into the $In$ places of components. Other than that, there are cases where there exist tokens in the initial marking of a connector or of a component. When this occurs, the role of these tokens is clearly explained.

A final note can be made regarding the use of transitions to represent the operations of the computation unit, as was shown in Figure 4.3(b). Since the tokens represent control, the operations of the computation units cannot change these tokens. However, if we extend our notation to include data values, tokens will be used to represent data as well as control, and will be of the form $(i, c, d), \ d \in DATA.$ In this case the outgoing arc of the operations of the computation unit would be $(i, c, f(d))$, where $f$ is the function representing the data transformation of the computation unit.

\section{4.2 Concurrency control}

The fact that our components are accessed by concurrent threads and that they contain data, necessitates a form of concurrency control, i.e. we need to control the way the data are being accessed by concurrent threads. In this section we
explain how concurrency control is performed in our model for passive components. Concurrency control is a term used in the field of databases and defines the method for coordinating concurrent user accesses to a database system. The main difficulty in concurrency control “is to prevent database updates performed by one user from interfering with database retrievals and updates performed by another” [30]. In the context of a component model, we define as concurrency control the methods used to ensure that when users are concurrently invoking the operations of a component, the write operations to some component data do not occur in parallel with the read or write operations to the same set of data. To be useful in practice, concurrency control must be done with minimal latency i.e. the time it takes for a single request to be processed must be minimal. This restriction is placed so that the concurrency control method is efficient. For example, a concurrency control method that forces concurrent user requests to be served sequentially regardless of the data that they access does not yield minimal latency. Minimal latency is achieved when a user thread is forced to wait executing a component, only if its execution would lead to one thread writing to some component data while another thread is reading or writing to the same data. The terms threads and processes are used interchangeably.

In order to define a concurrency control method for our component model we must first decide on the entity or entities that perform concurrency control. Since in our model data are held by components, concurrency control can be either performed by the components themselves, or by the entities that wish to access the same component concurrently. Different entities are different threads originating from some higher level connector. In Figure 4.1 these are the concurrent processes \( P_1, P_2 \) and \( P_3 \), that are concurrently invoking the passive atomic component. However, different threads are completely unaware of each other according to the semantics of our model. Forcing these threads to coordinate directly with each other before accessing a component implies that this coordination is performed by supporting run-time tools which are usually part of the execution environment. However, it is our belief that this should be done at the model level and not at the implementation level. This is because it provides clear semantics, it does not introduce unnecessary complications at the implementation, and mostly it avoids the pitfall that the implementation introduces (synchronisation) behaviour that is not defined by the model.

In our component model, starting from the lowest level of atomic components
and due to the property of components being well encapsulated entities, it is only the invocation connector that has access to the operations of the computation unit\(^2\). Based on this property, we need to define a synchronisation mechanism that is appropriate for atomic components. Given that concurrency control must be done by the atomic component itself, there are two possibilities for the entity performing it: it can be either the computation unit or the invocation connector. We argue that synchronisation code should be separated from the computational code, that is, synchronisation should not be done by the computation unit itself. This separation follows the well-known software engineering principles of separation of concerns and modularity, and it naturally fits with the basic principles of our model where control and computation are separated. Additionally, in the context of synchronisation mechanisms, numerous works have shown the merits of separating synchronisation from computation code [31, 93, 104, 105, 106, 27, 28]. We refer hereafter to synchronisation code as synchronisation constraints.

For the definition of our synchronisation constraints we must take into account some established requirements for a synchronisation mechanism. According to [31], a synchronisation mechanism must be expressive enough to describe (i) exclusion of a thread from accessing a resource, and (ii) priority for a certain class of threads over another that wish to access a resource concurrently. In that work, a resource (data) is not accessed directly but through a set of operations, and therefore a resource conforms to the notion of an abstract data type, as does our computation unit. The information needed to express the two classes of synchronisation problems falls into six categories:

1. The name of the invoked operation.
2. The relative time of the invocations.
3. The parameters of each invocation.
4. The “synchronisation state” of the resource, i.e. how many threads access each operation.
5. The local state of the resource, i.e. its data.
6. History information about the resource, i.e. which operations have already terminated their execution, by how many threads.

Therefore a synchronisation mechanism needs to utilise all six kinds of information.

\(^2\)Note that the computation unit has also access to the operations of itself, but that is discussed later.
In our model and at the lowest level of atomic components, data are accessed by the computation units which also conform to the abstract data type notion. Therefore the synchronisation constraints must have access to the above six types of information. Since the invocation connector is the sole entity accessing the computation unit, the invocation connector has this information, whose granularity is at the level of method invocations. Some primitive mechanisms, like semaphores [56] and monitors [69] are too low-level. On the other hand, mechanisms like the Universe Model [27, 28] or Abstract Behaviour Types [23] are better suited for concurrency between components, and not for concurrency within components. On the other hand, the SOS synchronisation paradigm (SOS – Service-Object Synchronisation) along with ESP (an Extension to Scheduling Predicates) that were proposed by McHale [93], provide the right level of granularity, they provide access to the six required resources of information and can be readily adapted to the context of our component model. Therefore our synchronisation constraints are based on the SOS paradigm and ESP. In the sub-sections that follow we briefly explain SOS and ESP and then show how we adapted ESP in the context of exogenous connectors.

4.2.1 The SOS paradigm and ESP

The SOS paradigm has the following characteristics:

1. It shows that all information needed for a synchronisation mechanism to be expressive (Bloom’s six types of information), derives from a single source, namely the arrival, start and termination events of method calls. This means that for every component, its top-level connector has all necessary information to define and enforce any synchronisation constraint for that component, and also that this top-level connector is the only entity accessing that information.

2. It supports a strict separation between the computation and the synchronisation code (including the component’s data).

3. It provides synchronisation at the level of method invocations.

4. It demands a mechanism to cause a pending invocation to start execution.

ESP is a sample synchronisation mechanism that illustrates the strengths of the SOS paradigm.
SOS uses an event-based model to represent the lifespan of method invocations and this is summarised in Fig. 4.4. In the figure, the server is the entity that provides a set of methods, and the client is the entity making the method call. On the server side, a method call is modelled with an arrival event. The method call might be delayed for several synchronisation reasons, and some time later it starts execution, and after performing some computation it terminates. On the client side, there are only two relevant events: the call of a method and the return of the method call. In the SOS model, all events occur interleaved, i.e. if two events occur at the same time, they are (perhaps arbitrarily) ordered. In the model all method calls, even calls by an entity to itself, generate events. Events in SOS are parameterised by the method name and are associated with actions in the following way: event → action. For instance we could write:

\[
\text{arrival(Read)} \rightarrow \text{numberOfReaders} = \text{numberOfReaders} + 1;
\]

The SOS paradigm demonstrates that all six types of information mentioned earlier derive from a single source, namely the method invocation. Information available at events, apart from the method name, includes the invocation parameters and the relative timing of events (all these are available at the arrival event). The “synchronisation state” of the resource, i.e. how many threads currently access it, and the “history information about the resource”, i.e. how many times each operation has terminated can be calculated by using counters on events. Therefore, five out of six types of information are available at events. Regarding the requirement that synchronisation code should have access to instance variables (private data of the component), the SOS paradigm supports the use of synchronisation variables to capture this information. McHale shows in \[93\] that

\footnote{This means that self-calls of the computation unit must also generate events. This issue is further discussed in Section 4.2.4.}
synchronisation variables and events suffice to capture all information available in instance variables. As a result, SOS can be used to express complex synchronisation schemes on shared resources (server objects in SOS, or computation units in our model).

The principles of SOS are embodied in a sample synchronisation mechanism, ESP (an Extension of Scheduling Predicates), that is targeted to object-oriented languages. Its syntax (in pseudo code) is be summarised in Figure 4.5. In the

```java
1: class Foo{
2: //The computation part of the code goes here.
3: variables
4: operations
5: synchronisation //The synchronisation code goes here.
6: variables
7: operations
8: events → actions
9: guards
10: }
```

Figure 4.5: Summary of the ESP syntax

figure, a class in ESP has a separate synchronisation part. This part can maintain its own (synchronisation) variables, operations, actions that happen on the occurrence of certain events, and guards. In this discussion we only present the guards; the interested reader is referred to [93] for a complete presentation. The guards have the form ‘method: booleanExpression’.

Upon the arrival of a method call on the server object, the booleanExpression is evaluated. If it evaluates to true, the method starts execution, else the method gets suspended until the evaluation is true. How this is accomplished is left to the implementation. Scheduling Predicates [94] (SP) is a sub-part of ESP. The boolean expressions in the guard statements can be written using SPs. We focus on SPs because our model currently support only SPs as a proof of concept for its synchronisation capabilities. Inclusion of the whole power of ESP is a part of future work, Section 10.2.

Scheduling predicates is a counter-based synchronisation mechanism. Synchronisation counters are variables counting how many times a method of a software module has been invoked, has terminated, and so on. The counters used are:
arrival(method): total number of invocations that have arrived to execute method
wait(method): total number of invocations that are waiting to execute method
start(method): total number of invocations that have started executing method
exec(method): total number of invocations that are currently executing method
term(method): total number of invocations that have terminated execution of method

From the above counters, it is only the arrival, start and term ones that need to be maintained, since the following equations hold:

\[
\begin{align*}
\text{wait}(\text{method}) &= \text{arrival}(\text{method}) - \text{start}(\text{method}) \\
\text{exec}(\text{method}) &= \text{start}(\text{method}) - \text{term}(\text{method})
\end{align*}
\]

Scheduling predicates can be used to specify method guards. For example in order to specify a simple read-write policy, we would write:

Read: \(\text{exec}(\text{Write}) = 0\)
Write: \(\text{exec}(\text{Read, Write}) = 0\)

\(\text{exec}(\text{Read, Write})\) is a shorthand for \(\text{exec}(\text{Read}) = 0 \land \text{exec}(\text{Write}) = 0\). We will introduce more complicated SP structures when it is necessary to do so.

Regarding the events and actions, some default associations can be made, namely the arrival, start and termination events of a method call update the relevant counters. For example, the start event increases by one the exec counter, and decreases by the same amount the wait counter.

4.2.2 Applying SPs to our model

In this subsection we describe how we applied SPs to exogenous connectors. Figure 4.6 shows how synchronisation constraints are applied to atomic components. Similar considerations apply to composite components as will be explained later. In atomic components, synchronisation constraints are completely separated from the computation unit. The computation unit simply states its synchronisation
requirements at the granularity of method invocations. The invocation connector reads the synchronisation constraints and enforces them. Since in our model we currently support only SPs, the syntax of the synchronisation constraint uses only guards for the methods of the computation unit. Associations between events and actions are automatically created for increasing or decreasing the relevant counters as was earlier explained. We should mention here that synchronisation constraints per se make no guarantee of progress \[29\]. That is, when multiple processes contend for executing the computation unit, some processes may be indefinitely blocked from doing so. In order to counteract this issue our implementation applies basic scheduling techniques.

For every operation of the computation unit there are three counters associated: EXEC, START, ARRIVAL. By default, these counters are appropriately modified when the relevant events of the method invocation occur, Figure 4.4. For every method name, a guard is associated with it and a logical expression is written. For the example in Figure 4.6 there are the following guards defined:

\[
\begin{align*}
\text{m1}: & \quad \text{exec(m1) = 0} \\
\text{m2}: & \quad \text{true}
\end{align*}
\]

The guards simply state that no two threads (processes) may execute method m1 concurrently, and that method m2 has no concurrency constraints. In the figure, processes P1 and P2 try to invoke method m1 concurrently. Events in, exec and out correspond to events arrival, start and term of the SOS paradigm. Therefore, if we consider that all three processes concurrently invoke the atomic component, thus creating three concurrent events m*.in, P3 will proceed uninterrupted whereas one of P1 and P2 will be non-deterministically chosen to execute method m1. After P1 executes m1, P2 may start.

Regarding composite components, since composite components may contain
data, synchronisation constraints are also necessary. This is shown in Figure 4.7. In the figure the assumption is made that at each level of composition there are

![Figure 4.7: Synchronisation contracts for composite components.](image)

...data defined. Therefore the data must be accompanied by some synchronisation constraints, which are read and enforced by the composition connector. The synchronisation constraints refer to the methods at the level of the composite component.

In general, due to the property of encapsulation in our component model, the top-level connector is the only entity that has access to the methods offered by the component. Furthermore the top-level connector has all the necessary information to enforce any synchronisation constraint (all six types of information). From these two remarks it follows naturally that it is the top-level connector that applies concurrency control for the component. As a final remark, in the case where a synchronisation contract is not needed, it is equivalent to:

\[
\begin{align*}
\text{method1:} & \quad true \\
\text{method2:} & \quad true \\
& \quad \vdots \\
\text{methodN:} & \quad true
\end{align*}
\]

### 4.2.3 Formal semantics

We have presented the synchronisation constraints for components and explained them in detail. In this section present their type equations in Z, we explain ...
how they relate to the computation unit and to components, based on the Z schemata already presented in figures 4.2 and 3.3. Finally we present a special synchronisation constraint that we call the mutex synchronisation constraint and show its CP-net semantics.

In Figure 4.8 the Z schema of a synchronisation constraint is shown. Every synchronisation constraints has a name, three counters and a logical predicate. Obviously, in what we call “synchronisation constraint” we include the guard of a method and all the associated counters. When we presented the type equations for the computation units, Figure 4.2, we also discussed the relationship between computation units and synchronisation constraints.

In Figure 3.3 where the Z schema for a component was shown, every component is associated with a possibly empty set of synchronisation constraints: \( \text{scs} : \mathbb{P} \text{SyncConstraint} \). This is because it is not compulsory for a component to have data. If for a method there is no synchronisation constraint, then there is no concurrency control performed for that method. We can now introduce additional constraints for the schema in Figure 3.3:

- Each synchronisation constraint corresponds to a unique interface element.
- There can be no two synchronisation constraints for a single component with the same name. This is because the name of each synchronisation constraint corresponds to exactly one interface element, and no two interface elements exist with the same name.

A special case of a synchronisation constraint for a component is the mutex synchronisation constraint. Mutex stands for mutual exclusion and it means that when a single process is executing the component, no other process may start execution. We can model this constraint using CP-nets because the method name invoked is irrelevant. Recall that in the CP-nets for our model, tokens represent control and no data. Therefore, we cannot distinguish between different

```latex
\begin{array}{l}
\text{SyncConstraint} \\
\quad \text{name} : \text{STRING} \\
\quad \text{exec, arrival, start} : \mathbb{N} \\
\quad \text{pred} : \text{LOGIC\_PRED}
\end{array}
```

Figure 4.8: Definition of a synchronisation constraint in Z.
method calls and this is actually the sole synchronisation constraint that we can model using CP-nets. As shown in Figure 4.9 only one process at a time may be executing the primitive passive component. The MUTEX fusion place is initialised with single token to ensure that.

4.2.4 Discussion

We mentioned before that the top-level connector is the only entity that has access to the methods offered by the component. There is a case where this claim is not true, namely self-calls made by the computation unit which are not noticed by the invocation connector. For composite components, this problem does not exist because when two components get composed they are not allowed to call each other: all method calls are initiated by the composition operator. Additionally, components in our model cannot call themselves.

Since self-calls of the computation unit are not noticed by the invocation connector, we overcome this problem by requiring self-calls from the computation unit to be made only to auxiliary methods of the computation unit that do not appear in the component’s interface. The guard of an auxiliary method is always true and is therefore omitted from the synchronisation contract. Any synchronisation constraints for the helper methods will need to be expressed in the synchronisation contract for the methods at the component’s interface.

4.3 Active atomic components

Figure 4.10 shows the structure of an active atomic component and a typical execution. An active atomic component consists of active computation processes and a channels connector. The active computation processes (\( AP_1 \) and \( AP_2 \) in the
We define active computation processes as a non-empty set of CSP processes executing in parallel. CSP [70] is a process algebra used to model concurrent systems. Informally, every CSP process consists of a set of synchronous channels used to communicate with other processes. The channels connector provides the interface for the component and access to some channels of the CSP processes. The active computation processes inside the active component may execute their actions in parallel, i.e. without any synchronisation, but they may also synchronise their actions, like actions $s$ in Figure 3.9(ii). During this synchronisation, processes may exchange data. This type of synchronisation is not possible in passive components. Additionally, it is possible that processes inside active components branch and join as shown in Figures 3.9(iii) and 3.9(iv) respectively. This is because the CSP semantics allow it.

In Figure 4.10 the active computation processes $AP_1$ and $AP_2$ execute in parallel and communicate via synchronous shared channels. Specifically they synchronise on channel $sync$. During this synchronisation, a single action is observed, and the two processes may exchange data values. Active processes may also spawn new active processes, or may also terminate. External processes may interact with some of the channels of the composed CSP processes by invoking the active component with the name of the CSP channel they want to interact with, along with any input parameters. The channels connector then forwards these calls to the CSP processes, and the external, calling processes synchronise with them. For the active atomic component in Figure 4.10, we consider that the only channels that external clients may interact with are channels $f$ and $g$ of the processes $AP_1$ and $AP_2$ respectively. These channels are exposed as methods $*f()$ and $*g()$. 

In Figure 4.10: Active atomic component.
4.3. **ACTIVE ATOMIC COMPONENTS**

and \( *g() \) respectively, in the interface of the atomic component (the asterisks are explained later).

The channels connector may be executed by concurrent external processes that wish to interact with the active computation processes. In the figure we only show a single external process \( P_1 \) calling method \(*f\). When \( P_1 \) calls method \(*f\), its execution \((f.\text{exec})\) will synchronise with the execution of channel \( f \) of the CSP process \( AP_1 \). This means that \( P_1 \) suspends until process \( AP_1 \) synchronises on channel \( f \), and vice versa: \( AP_1 \) cannot proceed, unless a call to method \(*f\) is made.

A distinguishing feature of active components is that because all component data are defined and accessed exclusively by the active computation processes, all synchronisation is handled by the computation processes themselves. Therefore, synchronisation constraints are meaningless for active components. More importantly, active components can respond to an endless sequence of inputs. An active component only needs sufficient input data in order to execute continuously, because it executes on its own processes. Therefore, it is possible that each single call to the active component synchronises with more than one channel of the active processes in sequence. In Figure 4.11, the external process \( P_1 \) declares that within a single call to the active component it will communicate with channels \( f, g, f \), and therefore it synchronises with channels \( f, g \) and \( f \) in sequence. It first synchronises with \( AP_1 \) on channel \( f \), then with \( AP_2 \) on channel \( g \) and finally with \( AP_1 \) on channel \( f \) again. To denote this capability of active components, we prefix with asterisks the methods at their interface, and as shown in Figure 4.11 it is reflected in the way they can be invoked. Because active components define no synchronisation constraints, it is possible to have “composite” \( in \) actions, otherwise the first action of the calling process \((*f * g *f).\text{in}\) would conflict with the ESP semantics, namely in ESP composite actions cannot be defined.
4.3.1 Coloured Petri-net semantics

We use CP-nets to illustrate the behaviour of active computation processes and how the channels connector interacts with them. Although we abstract from the exact interaction mechanisms, which will be discussed when we present the implementation of active atomic components in Section 7.2.1, CP-nets allow us to convey the principles of the channels connector’s operation.

As mentioned earlier, active computation processes are CSP processes. Because CSP is based on blocking, synchronous communication, it is always the case that when the external and internal processes wish to interact, they block waiting for each other until they both reach an appropriate state. Before describing the control flow of an active component, we present in Figure 4.12 the behaviour of an example active computation process. In general an active computation process may be in one of the following three states: it may be performing some computation, or may be blocked waiting to output the result of its computation, or it may be blocked waiting for some input data before commencing computation (we consider that interprocess communication occurs while ACPs compute). These states are represented respectively by places \texttt{compute}, \texttt{blockOut} and \texttt{blockIn} in the figure. Therefore when a token is in the respective place, it is considered that an active computation process is in that state. For the example active computation process in Figure 4.12, we arbitrarily chose to iterate over these three states. Certainly, this is a very generalised view since active processes need not follow this behaviour pattern. This view however suffices for this presentation.

In the initial marking in Figure 4.12, we have arbitrarily assigned one active computation process in each state. Token \texttt{ACP1} represents the control id of the first active computation process and similar hold for \texttt{ACP2} and \texttt{ACP3}. All processes have different tokens. In the figure, transitions \texttt{ACT1} and \texttt{ACT3} depend
on external processes for firing. The external processes are blocked waiting to write some values to the input parameters of the active computation processes during transition $ACT_1$, or they are blocked waiting to retrieve some output parameters during transition $ACT_3$. The external processes are forwarded in the dotted places by higher level connectors.

In Figure 4.13 the control flow of an example active component is presented,

![Control Flow Diagram](image)

Figure 4.13: Control flow of an example active component.

whose active computation processes are the ones presented in Figure 4.12. The initial marking is omitted for diagrammatic clarity, but similar to Figure 4.12, we can assume that there is an arbitrary distribution of tokens in places `compute`, `blockOut` and `blockIn`.

At run-time, the tokens representing the external processes will be forwarded in place $In$, via higher-level connectors. In the lower part in Figure 4.13 the active computation processes execute continuously within a loop and periodically interact with the channels connector. In the upper part of the channels connector, for each incoming process, a decision is made whether it should write to or read from the active computation processes. Although this decision appears non-deterministic, due to the abstraction of data values, the resolution is actually made by the name of the ‘invoked’ channel by the external process, for example name $*f$ in Figure 4.10. Because of the simplified view, Figure 4.13 does not represent the case of multiple channel invocations, $*f*g*f$ within a single call.
Once this decision is made, the external process enters the intermediate part of the channels connector where it must synchronise with an active computation process. Note that for the synchronisation to occur, an active computation process must be in the appropriate state. For example, if the external process is blocked waiting to read some output data of the active processes (BL_RD), then there must be at least one active process blocked waiting to output (blockOut), otherwise the external process is blocked indefinitely. When synchronisation occurs, the external process is returned to the output place. Place Compl is a fusion place and is used for diagrammatic clarity: all fusion places of the same name coincide. Therefore, in the figure, after ACT1 and ACT3 fire, tokens \( (j, c') \) return to the same place.

### 4.3.2 Interface generation

We mentioned earlier that the channels connector provides an interface for the active component and that it consists of some of the channels of the active computation processes. For example, in Figure 4.10 the active component has two methods, \(*\text{AP1.f}()\) and \(*\text{AP2.g}()\). Here we provide an overview of the interface generation process.

Active computation processes are CSP processes. CSP processes communicate via synchronous, shared channels. Channels in CSP may consume input data, provide output values, they may be bi-directional, or they may simply communicate signals. When the component developer defines the CSP processes for an active atomic component, they must also define what channels are ‘connected’ to each other. The channels that are used solely for interprocess communication are hidden, whereas the rest are exposed in the composite’s interface. This follows the well established software engineering principle of information hiding [107].

For each channel that is not hidden, an interface element is created for the atomic component. That interface element has the same name with the ‘exposed’ channel and it may define input and output parameters depending on the type of the channel. In Figure 4.14 we present a possible connection diagram for the active computation processes \( \text{AP}_1 \) and \( \text{AP}_2 \), that were presented in Figure 4.10. All channels in the figure perform no data i/o, hence the generated methods are \(*\text{AP1.f}()\) and \(*\text{AP2.g}()\). The asterisk before the method name indicates that this method belongs to an active component, and therefore it corresponds to a channel of a CSP process. During interface generation we also precede the channel
name with the name of the computation process that it belongs to.

## 4.4 Examples of atomic components

In this section we illustrate the usage of passive and active primitive components by means of two simple examples. First, we consider the password authentication component AC1 (with invocation connector I1) shown in Figure 4.15(a). This component associates in its internal data a password with a user id. On its interface it provides two methods, one for associating a user id with a password (insertPwd), and one for checking whether the given password matches with the given username (checkPwd). Figure 4.15(b) shows the synchronisation constraints associated with the computation unit. The constraints declare the predicates for each method that when true allow the process wishing to execute the method to do so. Therefore, for method insertPwd it is declared that a process may execute it, only if there are no other processes executing any other component’s method \((EXEC(insertPwd) = 0 \land EXEC(checkPwd) = 0)\). For checkPwd, a process may execute it only if there are no other processes executing insertPwd. When two processes try to invoke two methods of the atomic component concurrently, and the synchronisation constraints do not allow their concurrent execution, then one

---

\[\text{EXEC(insertPwd) = 0} \land \text{EXEC(checkPwd) = 0}\]
process is non-deterministically chosen to proceed and to actually execute the method.

Finally, all executing processes do not interact directly with each other. Synchronisation only occurs when accessing shared data, in which case the synchronisation occurs between the connector that enforces the synchronisation constraints, and the executing processes.

For an active primitive component, we consider a simple flight monitoring component shown in Figure 4.16. This active component (FLIGHTS) is used to monitor the arrival and departure times of planes. To that end, we use an active CSP process to model each plane ($PL_0\ldots PL_n$), because a plane at any time may update its arrival or departure time. These processes communicate along channels $arr_i$ and $dep_i$, the expected arrival or departure time respectively. The controller process $CTRL$ receives these communication events and updates both the arrival and departure buffer processes $ARR$ and $DEP$. For example, if $PL_i$ announces an arrival time $newTime$ on channel $arr_i$, then $CTRL$ communicates the pair $(i, newTime)$ on channel $addArr$ to process $ARR$ and the pair $(i, -1)$ to process $DEP$. The value $(-1)$ is used to denote that the specified plane is not departing. Similarly, a plane can ‘announce’ a new departure time.

Buffers $ARR$ and $DEP$ can be queried to provide all arrivals and departures. Their channels $allArvls$ and $allDprts$ are exposed to the interface of the active component, as methods that return an array of integers (in this simplified example we assume that time is expressed as an integer value).

Because all planes in the active component update themselves autonomously, every time the active component is called it provides the latest arrivals and departures information. This component may be called simultaneously by multiple external processes. This is natural, as many users may simultaneously request
flight information. Callers of this component may query both arrivals and departures information within a single call, by simply calling it with "*ARR.allArvls *DEP.allDprts" as input names, similar to "*f*g*f" in Figure 4.11.

Finally, we have model-checked the CSP specifications of the active component for deadlock and livelock freedom using FDR [5].

4.5 Passive composite components

As was briefly discussed in sections 3.2 and 3.4, in our model we use explicit composition operators or composition connectors that compose passive and active atomic components into composite ones. This was briefly shown in figures 3.2(b) and 3.5.

Atomic components are composed via their interfaces into composite components. Our composition operators allow composite components to be accessed by parallel processes (Figure 3.9(i)). Occasionally, and because composite components may define their own data [82], synchronisation constraints are necessary, Section 4.2.2. Therefore, concurrent processes need to synchronise their actions, as happens between actions a and d in Figure 3.9(ii). Additionally, we have defined composition operators that explicitly branch and join incoming processes to the sub-components, as shown in Figure 3.9(iii),(iv) respectively.

A distinguishing feature of our composition operators is that they preserve encapsulation of their sub-components. This means that the sub-components are accessed solely via their interfaces and that no other connector may access these sub-components directly. In addition to that, all our components are similar, in the sense that they define a uniform interface through which they may be invoked, i.e. a set of methods. These facts makes hierarchical composition possible, where a hierarchy of composite components is created incrementally, until the whole system is built, as was presented for example in Figure 3.5.

Composition operators coordinate control (and data) flow between the sub-components. Whenever composition occurs, the result is a composite component with an interface based on the interfaces of the connected components and on the composition operator used. A composition connector encapsulates control. It is used to define and coordinate the control for a set of components.

In our model we have defined a set of composition operators, each one defining different control and/or data flow. In this chapter we present the control and data
flow of all the composition connectors in detail, and we show for each case how the interface of composite components is generated from the interfaces of the constituent ones.

4.5.1 Sequencer and pipe

For sequencing we use the sequencer and pipe composition connectors. The sequencer is a connector that composes \( n \geq 2 \) components \( C_1, \ldots, C_n \) and can call methods in \( C_1, \ldots, C_n \) in that order. Similarly, the pipe calls methods in the same order \( C_1, \ldots, C_n \), but can also pass the results of calls to methods in \( C_i \) to those in \( C_k, k > i \). Figure 4.17 shows the execution paths of the concurrent processes

![Diagram of sequencer and pipe composition connectors.](image)

Figure 4.17: The sequencer and pipe composition connectors.

accessing a composite component that is defined using a sequencer or a pipe.\(^6\) The two processes \( P_1 \) and \( P_2 \) access the composite component in parallel. The connector forwards the two processes to execute the atomic components AC1 and AC2 sequentially. The pipe additionally can pass the result(s) of the invocation of AC1 as input parameters to methods in AC2. After both components terminate, the processes exit the sequencer. In the figure, processes accessing the composite component are independent of each other, unless they need to synchronise when accessing shared data inside the atomic components.

The sequencer and the pipe connectors do not have to call all connected sub-components every time they receive a call. For example, an \( n \)-ary pipe, \( n > 2 \), may only call \( C_1, C_3 \). The component sequences that are called are defined by the system developer during composition. We have introduced this behaviour for convenience during composition. For example, when composing components \( C_1, C_2, C_3 \) in that order, we automatically eliminate the need for

\(^6\)For simplicity, we do not consider the case where the composite component defines its own data. In that case an ordering would be imposed among the concurrent processes to protect the composite’s data.
the compositions \( \langle C_1, C_2 \rangle, \langle C_1, C_3 \rangle, \langle C_2, C_3 \rangle \) since they are already “contained” in the first composition.

When using the sequencer during composition no data are passed between the composed components whereas this is not the case with the pipe. This is reflected in the interface of the composite component. Because no data are communicated among the sub-components, the i/o parameters at the composite’s interface are a result of concatenating the i/o parameters of the sub-components. For example, in Figure 4.18 we present a composite component built from two

atomic components: a password checker AC1 and a fingerprint checker AC2, with invocation connectors I1 and I2 respectively. AC1, along with its synchronisation constraints, have been presented in Figure 4.15. In short, it provides methods for associating a username with a given password, and for checking whether a given username matches with the given password. AC2 provides similar functionality for fingerprint authentication and has similar synchronisation constraints. In the figure we show its interface.

AC1 and AC2 are composed into the composite CC1 using the sequencer connector SEQ. The connector sequences the calls to both components. During composition, the component developer decides which pairs of methods make sense for the composite and gives them an appropriate name. At most, the Cartesian product of the sub-components’ methods may appear at the composite component’s interface. The generated interface of CC1 is shown in Figure 4.18. It consists of two methods: insert and check. When method insert is invoked, a username is associated with the given password and fingerprint by sequentially invoking the relevant methods of AC1 and AC2. Method check acts similarly and it returns two boolean values, corresponding to the authentication results based on the password and fingerprint inputs. An example of a method that is not sensible for the composite would be the one that invokes methods insertPwd
and checkFngrPrt of AC1 and AC2 respectively.

It should be clarified at this point that the sequencer and pipe connectors are pre-defined and pre-implemented generic entities, that are parameterised by a) the components that are composed, and b) a map that defines which interface element at the composite’s level, maps to which interface elements of the constituent components. For example, in Figure 4.18 the map would associate method insert of the composite component, to methods insertPwd and insertFngrPrt of the constituents. For the pipe connector, additional mappings are included that define which output parameters of earlier invocations of the sub-components, are used for which input parameters of later invocations of the sub-components. All these parameters (components and mappings) are provided by the component/system developers, and are checked by the composition operator that they comply with the semantical rules of the components. Therefore, the component/system developer cannot invalidate, nor, override the semantical rules of the connectors through these parameters. For example, for the composition in Figure 4.18, it is always the case that methods of AC1 and AC2 are called in that order. Similar to the above hold for all the connectors in our model.

As mentioned earlier, our component model’s semantic rules state that during composition using a sequencer, the i/o parameters of the paired methods of the sub-components are concatenated at the interface of the composite. but this is not the exact case in the figure. For example, the check method of the composite should have been defined as \[\text{check} : \{\text{bool, bool}\} \rightarrow \text{String, Pass, String, FngrPrt}].\] Instead it has only three input parameters because the two strings are merged into one. This is because the two input strings represent the user id which is the same for both sub-components. The generated interface of CC1 seems to violate the semantical rules for the sequencer composition. However, for developing convenience we allow this merging of input parameters. During execution, the same String input parameter is used for the invocations of both sub-components. The semantics of this merging is explained by the introduction an extra, hidden, adaptation step. Specifically, after forming the composite component with an interface conforming to the sequencer’s semantics, so that the check has four input parameters, a specialised adaptor connector is used that merges input parameters. This is only a convenience that our implementation offers to the component developer. The relevant adaptor and when this implementation ‘trick’ is usually applied, is discussed in detail in Section 5.1.2.
4.5. PASSIVE COMPOSITE COMPONENTS

The interface of the composite component composed using the pipe connector, is similar to the interface of the composite component generated using sequencer instead. When the result of the invocation of method \( m_1 \) of component \( C_i \) is passed as input to method \( m_2 \) of component \( C_k \), \( k > i \), the output parameters of both methods are concatenated and the input parameter of \( m_2 \) is removed from the composite’s interface. Concatenation of output parameters is applied in order to allow the construction of equivalent composites, either using an \( n \)-ary pipe, or by consecutively using \( n - 1 \) binary pipes.

In Figure 4.19 we present a composite authentication component \( \text{AUTH1} \). It consists of a card reader \( \text{AC3} \), with invocation connector \( I_3 \) and of the composite component \( \text{CC1} \) that was presented in Figure 4.18. The output value of \( \text{AC3} \) (String) becomes the input of \( \text{CC1} \). As a result a String is no longer required as input for \( \text{AUTH1} \). When composite \( \text{AUTH} \) is given a card number representing a user id, a password and a fingerprint it checks if they match with the registered data and returns two boolean values and a string. Because \( \text{AUTH1} \) is a generic, reusable component, we expose the \text{insert} method for associating a card number with a password and a fingerprint on its interface.

![Figure 4.19: Authentication composite component.](image)

Similar to the use of the sequencer connector in Figure 4.18, given the interfaces of \( \text{AC3} \) and \( \text{CC1} \), method \text{check} in the interface of \( \text{AUTH1} \) is defined with an extra output parameter: \( \text{[bool,bool,String]} \) \text{check(int,Pass,Fngrt)}. This is according to the pipe semantics, as the output \text{String} of the \text{getUser} method can be used in subsequent compositions. However, as a convenience to the component developer, an implementation could offer the option to remove output parameters from the interface of \( \text{AUTH1} \) in a hidden step. Semantically, after forming the composite component with an interface conforming to the pipe’s semantics, an appropriate adaptor connector would be used to remove output parameters.
parameters. The relevant adaptor along with the above process are discussed in detail in Section 5.1.2.

**Type system**

The type system formalises the previous discussion on how the interface of a composite component can be created from the interfaces of the sub-components. We have formalised this as a function that given the sub-components and the composition operator, the interface of the composite component may be obtained. We have provided the formal definition in Z for all composition and adaptation connectors in our model with the purposes of a) providing unambiguous semantics, and b) guiding the implementation of the model so that it has strong, formal foundations.

Function \( cbSeqComposition \) in Figure 4.20 defines the composition for the

\[
\begin{align*}
\text{COMPOSITE} & : \text{COMPONENT} \\
\text{SEQ}, \text{CB} & : \text{CONNECTOR}, \quad \text{SEQ} \cap \text{CB} = \emptyset \\
\text{cbSeqComposition} & : \{ \text{conn} : \text{CONNECTOR} \mid \text{conn} \in \text{SEQ} \cup \text{CB} \} \\
& \times \{ \text{comps} : \mathcal{P}(\text{COMPONENT}) \mid \# \text{comps} \geq 2 \} \\
& \mapsto \text{COMPOSITE}
\end{align*}
\]

Figure 4.20: Construction of a composite using the sequencer/cobegin in Z.

connectors sequencer and cobegin. Here, we explain how the composite’s interface is constructed using the sequencer. The reasons explaining why the interface of the cobegin is created using the same function are given in Section 4.5.2.

Function \( cbSeqComposition \) uniquely associates a sequencer, or a cobegin, and a set of at least two components, with a composite component. The composite component is at one level more than the largest level of the constituent components, thus avoiding a cyclic definition. The interface of the composite component has the following characteristics:

- The resulting composite’s interface is at most the concatenation of the cartesian product of the interfaces of the constituent components.
- Every interface element of the composite is constructed by concatenating exactly one interface element from each constituent component. This does not contrast what we said earlier, namely that the sequencer connector
4.5. PASSIVE COMPOSITE COMPONENTS

does not have to invoke all sub-components. Recall from the discussion of Figure 3.3, that every component has an empty interface element, \textit{nilIE}, whose invocation does nothing. Therefore, although in the Z semantics every sub-component is invoked, the implementation need not follow this rule, while still conforming to our model’s semantics.

- Concatenation of two interface elements yields another interface element with any name, whose i/o sequences are the concatenations of the i/o sequences of the constituent elements.

One last remark regarding the composition function of the sequencer, is that it does not enforce the constituent components to be used only in a single composite. This ‘relaxed’ composition with sub-component ‘sharing’ is applicable to the deployment phase only, Section 3.3. Since in our type system we do not distinguish between these phases, this is left to the implementation.

Function \textit{pipeComposition} in Figure 4.21 creates a composite component out

\begin{verbatim}
pipeComposition : PIPE
    \times\{comps : seq(COMPONENT) | \#comps \geq 2\}
    \times seq(\mathbb{P}((\text{InterfElem} \times \text{InterfElem}) \times \mathbb{P}(\mathbb{N} \times \mathbb{N})))
    \mapsto \text{COMPOSITE}
\end{verbatim}

Figure 4.21: Construction of a composite using the pipe in Z.

of a sequence of the constituent components and a description of how the output parameters of invocations earlier in the component sequence, are passed as input parameters to invocations later in the component sequence. For convenience we name this information as \textit{pipings}. Each set in the \textit{pipings} sequence contains pairs of interface elements with pairs of natural numbers. This sequence must be have length one less than the component sequence and defines how sets of interface elements will be piped together. For example, for the composition in Figure 4.19, \textit{pipings} sequence contains a single set with the following single element:

\[((\text{getUser}, \langle \text{int} \rangle, \langle \text{String} \rangle), (\text{insert}, \langle \text{String}, \text{Pass}, \text{FngrPrt} \rangle, \langle \rangle)), \{(0, 0)\}\)

This element denotes that the output of \texttt{getUser} interface element at index zero, is piped to the \texttt{insert} interface element at index zero again.
Connector Template nets

As already explained in Section 4.1, we use CP-nets to represent the control flow of connectors and components. The importance of using CP-net semantics for our connectors lies in that we can define the control flow of the connectors unambiguously, but most importantly they can be used for defining the control flow of composite composition connectors which were introduced in Section 3.4. To that end, we have defined the CP-nets for all our composition and adaptation connectors. To be precise, and due to inefficiencies in CP-nets, see below, we have defined Connector Template nets (CT-nets) for capturing the control flow behaviour of our connectors. CT-nets are an extended version of CP-nets with compositional semantics. With that we mean that we have defined rules for composing CT-nets into composite ones.

According to the extended idealised component life cycle in Figure 3.7, we have introduced in our model a repository of connector templates. These templates are generic because they can be used to compose either connectors or components. Although CP-nets can be used for defining the behaviour of our basic connectors, CP-net semantics do not define their composition in order to form larger CP-nets. CT-nets fill in this gap.

In the current section we briefly describe CT-nets by the means of the sequencer connector. Actually, in the thesis we only present the CT-net for the sequencer and pipe composition connectors. The formal definition of CT-nets and the CT-nets of all composition and adaptation connectors in our model are given in [80]. A detailed analysis of CT-nets and how they are composed, are discussed in detail in Chapter 6.

In Figure 4.22 we present the CT-net for a binary sequencer/pipe. The same CT-net is used for both connectors because evidently the define identical control flow. We first discuss a simplified version of the binary sequencer template given in Figure 4.22(a).

The CT-net of the sequencer/pipe consists of three parts. The upper part defines the unique input and output places of the connector. The middle part, enclosed in a dashed rectangle, defines the control flow of the sequencer. The lower part consists of a set of composition places. The upper part is common to the CT-nets of all connectors/components in our model, and has been explained

---

7It may be argued that hierarchical Petri nets do define composition. We explain why this is not the case in Section 6.4.
4.5. PASSIVE COMPOSITE COMPONENTS

when we introduced the CT-net for passive atomic components in Section 4.1: processes from higher level connectors are placed in the \( \text{In} \) place and are returned to the \( \text{Out} \) place. The control flow of the sequencer connector defines that each control token in place \( \text{In} \) is eventually placed to the \( \text{Out} \) place, after flowing through the composition places \( \text{CP1} \) and \( \text{CP2} \) in sequence. This is because, for each token in place \( \text{In} \), transitions \( T_1-T_3 \) will fire in that order.

Composition places have an important role in the definition of the behaviour of connectors. They are ‘special’ places that can be either replaced by components to describe the behaviour of composite components, or, and as will be explained in Chapter 6, they can be replaced by other connectors to describe the behaviour composite connectors. It should be mentioned here, that instead of the two composition places \( \text{CP1} \) and \( \text{CP2} \), we could have used four places, \( \text{CP1.in} \) and \( \text{CP1.out} \), and, \( \text{CP2.in} \) and \( \text{CP2.out} \), in place of \( \text{CP1} \) and \( \text{CP2} \) respectively. \( \text{CPi.in} \) and \( \text{CPi.out} \), \( i \in 1,2 \), would then represent the input/output places of the replacing connector/component, and would probably make connector composition a more straightforward task. However, by keeping a single composition place instead of two, composition connectors are simpler to model and describe. This is because the connector’s behaviour is described in a single, complete diagram, instead of a diagram whose behaviour depends on how \( \text{CPi.in} \) connects to \( \text{CPi.out} \) in the replacing CT-net. Finally, with the current approach, we can use available tools ([49]) for simulating their behaviour, and confirming that their CP-net definition reflects their informal semantics.

Figure 4.22(a) presented a simplified version of the binary sequencer as an aid to the explanation. In Figure 4.22(b) we present the accurate version of the
binary sequencer template. In the figure we have introduced two memory places $\text{mem}_1$ and $\text{mem}_2$ and their role is explained as follows. Composition places have been used to represent either components or connectors. During deployment phase composition components may be ‘shared’ among different connectors, see also Figure 3.6. In terms of CT-nets, this translates into the fact that the control flows of two connectors ‘share’ the same composition places. Therefore memory places have been introduced so that connectors are able to distinguish the ‘correct’ tokens in shared composition places. Correct tokens are the ones that originate from the connector itself and not from another connector that used the same the place. For instance, the role of the $\text{mem}_1$ place is to ensure that when transition $T_2$ fires, it will only get from place $\text{CP}_1$ the tokens that were placed there as part of the control flow of itself, i.e. after $T_1$ fired. All our connector templates define memory places as shown in Figure 4.22(b) because their composition places can be potentially shared with other connectors. A full treatment of this issue, including the exact mechanism the memory places work and its ramifications in composition, are discussed in Section 6.2.

4.5.2 Cobegin and efficient cobegin

The \textit{cobegin} composition connector explicitly introduces concurrency, as shown in Figure 4.23. The cobegin connector [79] composes $n > 1$ components. For each incoming process to the composite component, it branches into $n$ processes, each one executing one of the sub-components in parallel. Before the cobegin returns, the executing processes must join as shown in the figure, and the union of the results is returned to the calling connector. As also evident in the figure, concurrently executing components do non-communicate with each other because that would break their encapsulation. Using the cobegin connector, the system developer can explicitly define branching and joining processes (Figure 3.9(iii),(iv)) in
the same connector. Allowing branching and joining points in separate connectors would make our approach not compositional. For example, for a single process starting the execution of the component defined using a branching connector, more than one processes would return.

As an example of using the cobegin connector, we revisit the password and fingerprint checking composite that was shown in Figure 4.18 and we replace the sequencer connector with the cobegin as shown in Figure 4.24. The cobegin connector is more efficient than the sequencer: for each incoming request, AC1 and AC2 are accessed concurrently instead of sequentially. Not surprisingly, the interface of the composite component CC2 is the same with the interface of the composite CC1 that was built using the sequencer connector in Figure 4.18.

A variation of the cobegin connector is the efficient cobegin, whose type equation is shown below:

\[
\text{effCbComposition} : \text{EFFCB} \times \mathbb{N} \\
\times \{ \text{comps} : \mathbb{P}(\text{COMPONENT}) \mid \# \text{comps} \geq 2 \} \\
\rightarrow \text{COMPOSITE}
\]

The control flow of the cobegin connector defines that when all of the sub-components finish executing, all returned process are joined and a process with the initial id returns to the caller. However, in certain cases it is desirable that a set of sub-components, say \( n \), start executing in parallel, and the execution process returns to the caller when a certain number, say \( m \), of sub-components return. This is typical in cases where the component-based developer introduces an array of components for performing the same computation, with the intention of using only some of the returned results. This is usually done for efficiency, i.e. the sub-components ‘compete’ to return as fast as they can, hence we name this connector efficient cobegin, and we symbolise it with \( \text{effCb}_{n}^{m} \). The remaining
returned results are ignored.

A two out of three efficient cobegin $\text{eff\text{Cb}}_{2}^{3}$ is shown in Figure 4.25. In the figure, the ternary efficient cobegin branches three concurrent processes and returns when two of them return, hence the number 2 in the processes’ join point. The number of joining processes must be statically fixed whenever this connector is used to compose components, either in design or in deployment phase, and it cannot be deferred until run-time. This is because in our model the interface of each component is affected by the number of joining processes. From the constituent sub-components, only methods with identical i/o parameters and types are used in the construction of the composite. The same input parameter(s) are used for all sub-components and $m$ output parameter(s) are merged. The reason all output parameters are merged, is because although all sub-components execute with the same input data, they may return different results. Users of the composite may need, for example, to calculate the average of the returned results before proceeding.

Finally, we notice that the generated composite’s interface is different from that when the cobegin is used, even though an $n$ out of $m$ efficient cobegin has identical control flow to the cobegin.

### 4.5.3 Selector and non-deterministic selector

For branching, we use the selector and the non-deterministic selector connectors. A selector connector that composes components $C_1, \ldots, C_n$ simply selects one component depending on a selection condition. For each executing process, exactly one of the sub-components executes as shown in Figure 4.26. The selection is solely determined by the calling process. If an interface element of a sub-component is unique to that sub-component, then it may appear as is at the composite’s interface. This is the case for interface elements of $m2$ and $m3$ of
sub-components AC1 and AC2 in the figure. When an external processes invokes for instance m2, then it is deterministically decided which of the sub-components will execute.

In the case where the sub-components have an interface element in common, for example m1 in the figure, then additional input parameters conds are needed to make the selection. Then based on the conds, the selector connector decides which sub-component to execute.

The non-deterministic selector [79] selects one of the lower level components that it is connected to execute non-deterministically. It is similar to the select statement of Ada [124] and to the non-deterministic ALT of Occam (when at least two of the guards evaluate to true) [57]. It models the situation where the actual branch followed does not affect the correctness of the program, and in our model is primarily used for efficiency. As with the usual meaning of non-determinism, it does not mean that a random choice is made, it only means that the system developer is allowed to choose any reasonable algorithm to make the selection, and users of the composite must not rely on the algorithm to determine the correctness of their code. Components connected by the non-deterministic selector must have the same interface.

The non-deterministic selector is shown in Figure 4.27. Because the connec-
tor internally chooses one of the two sub-components without the influence of the external processes, on the composite’s interface there are only exposed the interface elements that are common to both sub-components.

The type equations for both selector types are shown in Figure 4.28.

4.6 Active composite components

Using the composition connectors that we have defined so far, we get a passive composite whenever we compose a passive component and an active one. This composite is passive because for each incoming process it can take and respond to only one input at a time, like the passive sub-component. This means that the external process accessing the composite component can only invoke a single method at a time, because of the passive sub-components. In contrast, the active sub-component can respond to an endless sequence of inputs, as discussed in Section 4.3. To achieve an active composite component, we have defined a stateful facade in [78] that is presented below.

4.6.1 Stateful Facade

In object-oriented programming, a facade for a set of objects is a unified interface to all the methods of the objects, and the main method of the facade invokes methods in the objects. In a stateful facade [50], the behaviour of the main method depends on the current state of the facade.

In our component model, the structure of a generic facade and a typical execution are shown in Figure 4.29. A stateful facade, as with the rest of our connectors may be executed by multiple processes concurrently. A stateful facade exposes
the interfaces of its sub-components. It’s behaviour depends on it’s current state. By using a stateful facade, we can produce an active composite (from at least one active sub-component), since such a facade provides access to the methods of all the sub-components. An active composite can service an endless sequence of inputs and exhibit similar behaviour to that of Figure 4.11.

The state of the facade, possibly along with same input parameters, determines which sub-component’s methods may be called. In Figure 4.29, the passive sub-component AC1 and the active sub-component AC2 may be invoked when the facade is in state state1 and state2 respectively. For the sake of the example, we assume that when process P1 calls m1, the facade state is in state1 and therefore the call succeeds. The facade then switches into state2, and thereby when process P2 calls m2, the call does not succeed and returns immediately to its caller. The state of the facade may be altered after the thread of control returns from a successful execution of a sub-component. This is because the facade’s data reflect the design decision that the result of the invocation to a sub-component affects the next invocation. Finally, process P3 calls the facade with a sequence of names belonging to the active sub-component. Because the facade is in the appropriate state the call succeeds. In this example it is solely the state of the facade that decides which of the sub-components may execute. In general, and as shown in the example below, input parameters may be also used.

For a more practical example of using the stateful facade, we have built an authenticated flight monitoring system by applying our composition operators to active and passive components. The system is shown in Fig. 4.30(a). Its purpose is to provide authenticated access to flight information. The authentication AUTH component in Figure 4.30(b) is very similar to the AUTH1 component that has been explained in detail in Figure 4.19. However, for efficiency reasons we have used composite CC2 shown in Figure 4.24 that is built using a cobegin connector instead of the composite CC1 that is built using a sequencer. The purpose of the
AUTH component is to authenticate a user based on an integer card number that represents a user id, a password and a fingerprint. The FLIGHTS active primitive component is the one presented in Figure 4.16 and can be queried for the arrivals and departures information.

The facade exposes the interfaces of both sub-components and allows access to the FLIGHTS component only to those users that have been successfully authenticated. We can envisage that these users are the terminal users at check-in desks in an airport. Each terminal user must authenticate before accessing the flight information. Once authenticated, they may access this information directly. Therefore, composite CC provides access to both the authentication and the flight components. Users with a given id may access the flight monitoring component only if they have been authenticated first. To that end, the state of the facade connector consists of the ids of the users that have been authenticated. Therefore, when a user attempts to access the flights component, the facade checks whether the user id is part of its state, and if not it generates an error message. For distinguishing between different users, an extra integer input parameter has been added to the interface methods of FLIGHTS when they are exposed to the composite’s interface. Finally, authenticated users can execute both methods of the
active sub-component within a single call to the composite CC. For example calling “*allArvls *allDprts” indeed returns two arrays containing the (expected) arrival and departure times for all aeroplanes. This is because authenticated users may access directly the FLIGHTS component, and as already discussed, the FLIGHTS component provides that functionality.

The significance of the stateful facade is exemplified when we try to use our other connectors to produce the authenticated flight monitoring system. Such a composition would produce not only a passive composite, but also the wrong one. For instance, the composite could be composed using a pipe and a guard as shown in Figure 4.31. The result of the authentication is passed into the guard connector. The guard is an adaptation connector, that allows control to pass only if an internal condition is satisfied and is further discussed in Section 5.1.1. Only if the authentication result is true may the user access the flight information. The obvious disadvantage of this solution, is that the user needs to authenticate every time before accessing the flights information. This is not the behaviour that we want.

In Figure 4.32 the type equation of the stateful facade is shown. The interface of the stateful facade is generated from the union of the interface of the sub-components, after substracting the interface elements that are common in at

![Figure 4.31: Authenticated flight monitoring system.](image)

```
stFacComposition : ST_FACADE
    × {comps : P(COMPONENT) | #comps ≥ 2}
    ↔ COMPOSITE
```

Figure 4.32: Construction of a composite using a stateful facade in Z.

of the stateful facade is generated from the union of the interface of the sub-components, after substracting the interface elements that are common in at
least two sub-components. Otherwise, i.e. if the common interface elements were kept, the stateful facade would not be able to ‘forward’ the invocation to the appropriate sub-component. Out of these interface elements, it is possible to precede their input parameters with new input parameters that will be used by the stateful facade when making the selection. This was also the case in the example in Figure 4.30.

### 4.7 Chapter summary

In this Chapter we presented passive and active atomic components, and their composition. Concurrent processes were defined in terms of active components and composition operators. Through this discussion it became apparent that passive components need a means for controlling concurrency. This gave rise to the synchronisation constraints that were presented in detail in Section 4.2. To ease explanation, the above discussions were accompanied by an example. Specifically, we demonstrated how our model can be used to hierarchically construct an authenticated flight monitoring system. This system contains both active and passive components, and all connectors used are concurrent.

In the next Chapter 5 we discuss component adaptation and show the various ways in which an assembled system can start executing.

---

*This does not constitute a limitation, because it is always possible to adapt a component prior to composition, Ch.5.1, and thus remove any naming ambiguities.*
Chapter 5

Adapted components and component execution

In the previous chapter we defined in detail passive and active atomic components, and we demonstrated how they are composed in our model using explicit composition operators. Although composition is central to the life cycle of components, there are other operations on components that form a part of their life cycle. Particularly in the deployment phase, i.e. the phase where we assemble systems from components residing in a design-phase repository, components can be adapted in order to be made suitable for the specific system under construction. After the whole system is assembled using an assembler tool, Section 2.2.1, the system may start executing. In this chapter we present how we adapt components into our model and how an assembled system can start executing, i.e. how it can transit in the run-time phase of its life cycle.

5.1 Adapting components

Adapting a component is a very important part of a component’s life cycle. Components residing into a component repository may be used in different contexts for constructing systems with a varying degree of differentiation. In order to maximise component re-use a component model should provide means for component adaptation, i.e. making components suitable for use under some particular context. In our component model we have defined explicit adaptation operators or adaptors. We briefly explain their basic principles.

All our components, be it atomic or composite, they have a top-level connector
that defines the control flow for the component. Our adaptors do not change the behaviour of that connector, as this would have unpredictable results during composition. Instead, our adaptors operate on the interface level of components respecting the control that they encapsulate. Evidently, this reflects the fact that components are black-box entities. Consequently, whenever we adapt a component a new adapted component is created. Figures 3.2(c), 3.5 and 3.6 showed that an adaptor is an explicit connector that is applied to components externally (or exogenously) and results into a new adapted component.

In general when adapting a component we either adapt the control flow that reaches the component, or we leave the control flow unaffected and only adapt its interface. These two kinds of adaptation are discussed in Sections 5.1.1 and 5.1.2 respectively.

It is worth noting that adaptation connectors are more suitable for the deployment phase of a component’s life cycle. As already explained, this is because in the deployment phase components are deployed into a particular execution environment having a specific system in mind, whereas the purpose of the design phase is to build generic, reusable entities. However, the above does not mean that adaptors cannot be used in the design phase – on the contrary, adaptation of design phase components is an allowed operation in our model. This is because in the design phase, prior to composition, component adaptation can be used as a means for facilitating composition, especially since in our model, the interfaces of the sub-components affect the result of the composition. For example the non-deterministic selector, see Figure 4.27, can only ‘forward’ to the composite’s interface, methods with exactly the same signature.

5.1.1 Adapting the control flow of a component

In terms of control flow, we have defined unary operators that adapt the control before it reaches the sub-component. In general, there are two possible ways to perform that. First, the adaptor may block or allow the control to pass to the connected sub-component based on some boolean condition(s). Second, the adaptor may loop over the connected sub-component.

The first kind of adaptation gives rise to the guard adaptor. The second kind of adaptation gives rise to a family of loop connectors. Different kinds of loop connectors are created by varying the characteristic elements of a loop. If the connector loops sequentially a finite number of times, then we get the
control flow described by the *finite sequential loop* adaptor. If the connector splits the incoming process into a finite number of parallel processes that execute the connected sub-component in parallel, then we get the *finite parallel loop* adaptor. Finally, in the *finite sequential loop* there are two possible variations depending on whether output data of the previous invocations are used as inputs to the subsequent ones. We now present these adaptors in detail.

The types for the adaptors that adapt the control flow of a component are shown below. $ADAPT_{CTRL}$ is the set of all control adaptors.

$\begin{align*}
ADAPT_{CTRL} & \notin \mathbb{P} \text{ CONNECTOR} \\
\text{GUARD, FP\_LOOP, FS\_LOOP\_PIPE, FS\_LOOP\_SQ} & \notin \mathbb{P} \text{ CONNECTOR} \\
\langle \text{GUARD, FS\_LOOP\_SQ, FS\_LOOP\_PIPE, FP\_LOOP} \rangle & \\
\text{partition} \ ADAPT_{CTRL}
\end{align*}$

The type equations that relate each adaptor sub-type to the adapted component type, are of the form:

$adaptFunction : ADAPT_{CTRL}_{TYPE} \times \text{COMPONENT} \rightarrow ADAPTED$

Because of their simplicity, we will not repeat them below, where we elaborate on the semantics of each of the control adaptors.

**Guard**

The guard connector adapting a single component is shown in Figure 5.1. As with

![Figure 5.1: Guard adaptor.](image)

the rest of connectors and components, the guard connector supports concurrency
by allowing it to be executed by concurrent processes. In the figure, processes $P_1$ and $P_2$ concurrently invoke the adapted component. $P_1$ succeeds in invoking sub-component $C$, i.e. the conditional statement of the guard evaluates to true, whereas $P_2$ fails and immediately returns.

The interface of the adapted component consists of the interface elements of the sub-component, where the input parameters of each interface element are preceded by the same sequence of conditional variables. This is because the guard connector guards the component as a whole, and not individual methods.

An example of using the adaptor connector was presented in Figure 4.31. In the figure the interface of the adapted component $AD_1$ is created by preceding all methods of the $FLIGHTS$ sub-component with two boolean variables, the result of the authentication of the $AUTH$ sub-component. The conjunction of these variables is used by the guard connector for deciding whether control should be allowed to pass or not.

**Finite sequential loops**

A finite sequential loop, iterates over the connected sub-components a fixed number of times. This is shown in the CT-net in Figure 5.2(a), where the sub-

![Figure 5.2: Sequential loop with fixed number of iterations.](image)

component $C$ is executed $k$ times, hence the number $k$ within the loop’s decision point. The number of iterations gets fixed when the connector is attached to the sub-component. The finite sequential loop can be executed by concurrent processes, each of them iterating a number $k$ of items independently of the other processes.

When looping a finite number of times over a single component, there are two possibilities regarding the data flow of the adapted component, similar to when executing a set of components in sequence. These two possibilities give two
5.1. ADAPTING COMPONENTS

different adaptors. The first one is \texttt{FS\_LOOP\_SQ} and simply concatenates the i/o parameters of the invoked interface elements and is similar to the sequencer connector. The second one \texttt{FS\_LOOP\_PIPE} can be used for passing the output of a previous invocation to the input of a latter one and is apparently similar to the pipe connector. The interfaces of the adapted components are similar to the ones generated when the sequencer or the pipe composition operators are used from some components, Section 4.5.1. In the case of the \texttt{FS\_LOOP\_SQ}, the i/o parameters of all invoked interface elements are concatenated. When the \texttt{FS\_LOOP\_PIPE} is used, output parameters are concatenated. Input parameters are also concatenated apart from those input parameters that are evaluated based on the output of an earlier invocation.

The fact that in both cases the adapted sub-component gets executed the same number of times each time that is invoked by a process, does not mean that each iteration invokes a different interface element. The same interface element may be invoked any number of times. Additionally, and as was shown in Figure 3.3, every component has a \texttt{nilIE} which when invoked is inconsequential. Therefore, although each time a concurrent process invokes the adapted component \( k \) times, the number that ‘actual’ methods invoked, i.e. methods other than \texttt{nilIE}, can be less than that. For instance, by invoking \texttt{nilIE} \( k - 1 \) times, methods from the interface of the sub-component appear in the interface of the adapted component.\footnote{Apparently, an implementation can be more efficient and need not invoke the sub-component \( k \) times. This is how we have implemented the finite sequential loop adaptor, Chapter 7.}

The above observations apparently apply to the finite parallel loop as well, which is discussed below.

\textbf{Finite parallel loop}

Parallel loops constitute a well-known notion in parallel computing. The objective is to make the iterations of a loop independent, so that that each iteration is executed by a different process. The way iteration is made independent in our model is by fixing their number at the moment the adaptor is attached to a component, and by demanding all input data to be provided when the adapted component is invoked.

The finite parallel loop (\texttt{FP\_LOOP}) adapting a single component is shown in Figure 5.3. We use the finite parallel loop adapter when we wish during a
single call to a component, to access it using parallel, independent threads. As shown in the figure, for each incoming process, a predefined number of processes $k$ are created that execute the sub-component in parallel. When all processes terminate, they are joined, and control return to the caller.

The split sub-processes are independent in the sense that they ‘carry’ different data for execution, which is similar to the fact that the processes in a parallel loop refer to different values of the loop index. If these threads access the same data inside the adapted component $C$, this synchronisation is handled by $C$.

The interface of the adapted component is identical to the interface generated by the finite sequential loop $FS\_LOOP\_SQ$ that does not pass data among the executing processes. Both loops concatenate all i/o parameters for the invoked methods, the difference being the $FP\_LOOP$ is more efficient.

5.1.2 Adapting the interface of a component

In this section we define adaptors that do not affect the control flow reaching the adapted sub-component, in the sense that each process arriving at the adapted component will always execute the connected sub-component exactly once. As such, the CT-net describing their control flow is identical for all adaptors. These adaptors only change the interface of the sub-component.

Their types are given below:

\[
\text{REN\_ADAPT, HIDE\_ADAPT, R\_O\_ADAPT, M\_I\_ADAPT,}
\text{COND\_INV\_ADAPT, ADAPT\_IE, CONNECTOR}
\]

\[
\langle \text{REN\_ADAPT, HIDE\_ADAPT, R\_O\_ADAPT, M\_I\_ADAPT,}
\text{COND\_INV\_ADAPT}\rangle \text{ partition ADAPT\_IE}
\]
5.1. ADAPTING COMPONENTS

\textit{REN\_ADAPT} renames some of the interface elements of the sub-component and \textit{HIDE\_ADAPT} hides, or removes, some of them. \textit{RO\_ADAPT} can remove any of the output parameters of an interface element. \textit{MI\_ADAPT} merges two or more of the input parameters of an interface element into a single one. Finally, \textit{COND\_INV\_ADAPT} selects from a set of interface elements with identical i/o parameters which one will execute based on some boolean condition. \textit{ADAPT\_IE} collectively defines these types.

The type equations that relate each adaptor sub-type to the adapted component type, are of the form:

\[
\text{adaptationFunction} : \text{ADAPT\_IE\_TYPE} \times \text{COMPONENT} \mapsto \text{ADAPTED}
\]

Because of their simplicity, we will not repeat them below, where we elaborate on the semantics of each of the interface-changing adaptors.

**Renaming and hiding interface elements**

First we define the \textit{renaming adaptor}. As its name suggests, it changes the name of some, or all, of the interface elements of a component. During renaming there is the restriction, that in the resulting adapted component, there can be no two interface elements with the same name, as this would violate the semantics of our components, Figure 3.3. Next, the \textit{hiding adaptor} removes some interface elements from the sub-component, so that the created adapted component has a strictly smaller interface than its sub-component.

**Merge input and remove output parameters**

The \textit{merge input parameters} and the \textit{remove output parameters} adaptors, operate on the input and output parameters of interface elements. They are particularly useful for adapting a composite component created with a sequencer, a cobegin or a pipe composition connector. The first, tmerge input parameters adaptor, can merge input parameters of the same type. For example, consider method \textit{insert} of component \texttt{CC1\_temp} shown in Figure 5.4 that is created using a sequencer connector. Its input parameters sequence \(\langle \text{String, Pass, String, FngrPrt} \rangle\) becomes \(\langle \text{String, Pass, FngrPrt} \rangle\) for the adapted \texttt{CC1} that is formed using the \texttt{MI} connector. Similar apply to \textit{check} method. When an external process invokes the adapted component it only needs to provide a single input string. The adaptor
connector then internally replicates input parameters as necessary before invoking the sub-component CC1_temp.

The remove output parameters adaptor is used for removing some output parameters of an interface element. Figure 5.5, shows the result of composing CC1 with atomic component AC3 using a pipe composition connector. For both methods in CC1, the pipe passes the resulting string from AC3’s invocation into their first input parameter. According to the pipe semantics, the string result from AC3 appears as an output parameter of the methods of the interface of AUTH1_temp, as they might be used in subsequent invocations. When this is not needed, the R_O adaptor can be used to remove them. In Figure 5.5, R_O adaptor has been used for this purpose, as is visible in the interface of AUTH1.

It is worth noting that these two adaptation operations are very common when a sequencer, cobegin or a pipe connector is used, because in these cases
it is very likely that we get an ‘explosion’ of i/o parameters. As a result, an implementation can ‘embed’ them during the composition process when one of the aforementioned connectors is used. In such a case the temporary composite can be completely hidden and only the final, adapted component is presented. In Figures 4.18 and 4.19, composites \textsc{CC1} and \textsc{AUTH1} are actually formed this way, using a temporary composite and an adaptor connector as shown in Figure 5.4(a).

### Conditional invocation

The conditional invocation adaptor decides which method of the sub-component to invoke based on some boolean condition. It is similar to the selector connector, Section 4.5.3, but instead of choosing among methods belonging to different components, it chooses among methods belonging to the same component. In order to choose among them, these methods need to have identical i/o parameters. A single method is created in the adapted component’s interface with the same output parameters, but whose input parameters have been prefixed by an extra sequence of variables that are used for making the selection. For each invocation on the adapted component’s method, the conditional invocation connector must always select one method from the sub-component to execute. Finally, interface elements in the sub-component that do not match with any other interface element in terms of their i/o sequences can appear unaffected on the adapted component’s interface.

An example of using the conditional invocation adaptor is shown in Figure 5.6. In the figure, component \textsc{C} has five methods. \textsc{m1} and \textsc{m2} have the same i/o parameters and so do \textsc{m3} and \textsc{m4}, whereas \textsc{m5}’s i/o parameters do not match with any other interface element. As a result, conditional invocation \textsc{C_I} can be used to choose between \textsc{m1} and \textsc{m2}, or between \textsc{m3} and \textsc{m4}. This is reflected in methods \textsc{m12} and \textsc{m34} respectively, that have introduced used some extra input parameters.

**Figure 5.6: Conditional invocation adaptor.**
for the selection processes. For instance, when \texttt{m12} is invoked input parameters \texttt{conds} are evaluated, and either \texttt{m1} or \texttt{m2} gets invoked. The component developer must ensure that always one succeeds, so that control always reaches the sub-component. Finally, method \texttt{m5} appears in the interface of \texttt{AD} unaffected.

5.2 Transition into run-time phase

After the whole system is assembled in the deployment phase using an assembler tool, Figure 2.2.1, the system may start executing. In order for this to happen, the assembled system must be instantiated with appropriate resources. For example the system must be loaded into the memory of the execution environment, perhaps according to a certain instantiation policy, and the system must be scheduled for execution. The latter means that the all processes necessary for its execution need to start executing.

In our model, there are three sources of concurrent processes. First, active atomic components that were presented in Section 4.3, define a set of active computation processes that execute in parallel. Second, certain composition and adaptation connectors have been defined, that when invoked, initiate a number of concurrent sub-processes. These are the cobegin and the efficient cobegin composition operators, Section 4.5.2, as well as the parallel loop adaptation connector, Section 5.1.1. The third source of concurrent processes are the ‘external’ processes that execute a system and represent unique user requests.

For the first source of concurrency, the hierarchical nature of our component model provides an elegant way of starting all active components. Any assembled system in our model is actually a composite component. Therefore, when transiting into the run time phase, the whole composite is started. Via recursion, its active sub-components (if any) create the active computation processes and start executing. The second source of concurrency occurs at run time, i.e. while the system is executing. In that sense, it does not affect the transition of an assembly into run time, but instead relies on the implementation language and not on any facilities of the component model. For the third source of concurrent processes, we have defined a family of (possibly infinite) loops. These loops are top-level connectors, executing in their own processes. They represent user requests to the system and are actually the ‘external’ processes that execute the whole system. Top-level loops are discussed in detail in Section 5.2.1 below.
5.2. TRANSITION INTO RUN-TIME PHASE

5.2.1 Top-level loops

We have defined four kind of top-level loops that when applied in a deployed component, generate an executing system. Their types are shown in Figure 5.7, where \( TL\_LOOP \) represents the supertype for all top-level loops. All the loops represent cases where the total number of iterations is not fixed, and therefore they can be executed an infinite number of times, or until the system shuts down. The fact that these loops can only be applied to deployment phase components is captured by the \( buildSystem \) function. This function has the restriction that a loop can only be applied to a component whose data have been initialised, which in terms of our type system, forms the main distinction between design and deployment phase components, Section 3.3.

The \( LOOP \) connector waits for input user data. It then invokes the connected sub-component and finally it returns the result to the user. This process occurs \textit{ad infinitum}, or until the system user decides so. The loop is concurrent in that it may serve multiple users concurrently. In Figure 5.8 we show the authenticated flight system that was presented in Figure 4.30 connected to a loop connector. Concurrency is achieved because the concurrent loop actually consists of multiple loops, each one serving a single user sequentially. We can envisage that these loops are actually terminals in the check-in desks of an airport.

All our top-level loops are concurrent in a similar manner, i.e. they consist of an unbounded number of sub-loops of the same type. In the following, when we describe the behaviour of a loop connector we refer to the behaviour of one of the sub-loops. The actual loop connector will consist of an unbounded number of
CHAPTER 5. ADAPTED COMPONENTS & COMP. EXECUTION

int[] allArvls(int)
int[] allDprts(int)
[String,bool,bool] check(
[String,bool,bool] int,Pass,
int[] ARR.allArvls() int[], DEP.allDprts() int,Pass,FngrPrt)

Figure 5.8: Executing the authenticated flight system.

these sub-loops. Their actual number is defined at the moment the connector is applied at the deployed component, i.e. at the moment the deployed component transits into run-time. Depending on the application, the actual number of loops may or may not change dynamically.

Another form of looping is the infinite loop INF_LOOP. An infinite loop executes a single method of a deployed component continuously. As soon as the previous invocation returns, the result of the invocation, if any, is displayed and the loop re-invokes the same method. In the general case, the method invoked will be of the form [results] method(inputParams). During the first invocation the system developer provides the input parameters. Subsequent invocations can either use the original input parameters, the results of the previous invocation, or a combination of both. This is specified by the system developer when the assembled system transits into run-time. The infinite loop is useful in systems that need to execute continuously with minimal or no user intervention. For example, a typical control system [66] consists of sensors and actuators. Within a feedback loop, sensor readings are transferred to the actuators’ controllers. An infinite loop provides the necessary control flow for this type of systems, i.e. it continuously triggers the process of reading the sensors and then controlling the actuators.\(^2\)

The pre- and post- check loops (PRE_LOOP and POST_LOOP in Figure 5.7) correspond to the while ... do and do ... while loops in programming languages. The pre- check loop is shown in Figure 5.9(a) and the post- test loop in Figure 5.9(b). In the figure their difference is obvious, as the pre- test loop first evaluates the loop’s condition and based on that it decides whether execute that attached sub-component or not. After each iteration, the loop’s exit condition is

\(^2\)Do be precise, in control systems it also necessary to define the period of successive invocations. Currently, in our model we do not support the notion of time.
re-evaluated. In the post-check loop, the attached sub-component is executed at least once. After the first invocation, the loop’s condition is re-evaluated before deciding whether to iterate again, or terminate the loop.

In both loops we assume that component \( C \) provides a method \([\text{results}] \) \( \text{method}(\text{inputParams}) \) that is invoked continuously. In the case of the post-check loop, the evaluation of the loop’s exit condition is based on the results of the previous invocation. Additionally, the results of a previous invocation are used as input parameters for the subsequent one. In the case of the pre-check loop, during the first iteration the input parameters are used for evaluating the exit condition of the loop. In subsequent iterations, the results of the immediately previous invocation are used both as the input parameters and for evaluating the exit condition.

It can be noted that in the case of the infinite loop and of the pre- and post-check loops, the results of the previous invocation are used as the input parameters for the next one. As a result, interface elements of a component \( C \) that can be invoked using these connectors are ones, such that for each input parameter type there exists at least one output parameter type, with the same type. Also, even though an interface element may need no input parameters, it must have at least one output parameter that will be used for evaluating the loop’s exit condition. Formally, for each interface element \( ie \) that can be used by the loop, the following conditions must hold: \((\#ie.\text{output} > 0) \land (\forall \text{type} : ie.\text{ran}(\text{ie.input}) \bullet \exists \text{type}' : \text{ran}(\text{ie.output}) \bullet \text{type} = \text{type}'\).
5.3 Chapter summary

We have presented in this chapter how components in our model can be adapted and how an assembled system can start executing. Considering that components are black-box entities, adaptation occurs outside of components in two possible ways. First, adaptation may result in changing the interface of the component without affecting its control flow. Alternatively, it may alter the control (and data) flow reaching the component. Evidently, in the latter case its interface may be also affected. By respecting the black-box nature of components, every time a component is adapted, a new, adapted component is created.

In the next chapter we show how connectors in our model are composed into composite ones.
Chapter 6

Composite connectors

In our component model there are two basic entities, components and connectors. Components are passive or active units of computation and connectors are units of control. Our connectors compose components into composite components according to the idealised component life cycle, Figure 2.1. We have extended the idealised component life cycle to include the life cycle of connectors as shown in Figure 3.7. In Chapter 4 we discussed in detail components and their composition. In this chapter we discuss connector composition.

Composite connectors are formed from the composition of at least two connectors according to a composition equation. One connector has the role of the operator, and the rest are the operands of the composition equation. The operator can be one of the composition connectors defined earlier, since in our model composition connectors are used not only for component composition, but for connector composition as well. Our approach is algebraic, i.e. the result of connector composition is a (composite) connector that can be stored in the connector repository. Composite connectors can be retrieved and used in further compositions, both as operators and operands. In this way a hierarchical, or systematic construction of complicated connectors is defined.

Previous work ([77]) demonstrated that the observer design pattern [61] is a composite operator. However in that work no formal semantics were given, the association between components’ and connectors’ life cycle was not made, and consequently the implementation presented only served as a proof of concept and was not integrated into the component life cycle. Additionally, previous work was limited in scope because the connectors used did not express concurrency. In the current work we define a syntax for connector composition that conforms
to the connectors’ life cycles. The syntax forms the basis of our implementation which tightly integrates component and connector life cycle. We have additionally defined formal semantics for our connectors that capture their behaviour in terms of their control flow. Based on the syntactical form for connector composition, we have defined semantical rules, that taking into account the behaviour of the connector operator and of the connector operands of the composition, define the behaviour of the composite connector.

The syntax of connector composition actually defines the structure of composite connectors and is discussed in Section 6.1. The semantics of composition, i.e. how the behaviour of the composite connector is derived from the constituents, is discussed in Section 6.2. In Section 6.3 we show that our approach to connector composition is unique by comparing against related work. Our comparison is not limited to component models only, and spans the related literature in the broader field of software engineering. In Section 6.4 we discuss some issues related to connector composition that we were not presented elsewhere in this chapter and we conclude in Section 6.5 by providing the chapter summary.

6.1 Structural composition

Structural or syntactical composition, as its name suggests, defines the structure of a composite connector in terms of its constituents. It defines the operator and the operands of a composition, and most importantly, it defines how these are “glued” together to form a composite connector. Before presenting structural composition in detail, we need to distinguish between connector templates and connector instances.

A connector template is a generic connector stored in the connector repository. We store in the repository composition and adaptation connectors, as in sections 4.5, 4.6 and 5.1 respectively. By the term ‘generic’, we mean that the connector is not connected to any specific component or connector. Instead, it can be used both for component and connector composition (or component adaptation for the case of adaptors). The arity of composition connectors may be parametric, for example all composition connector that we have defined so far are $n$-ary, or their arity can be fixed to a specific value (adaptors are always unary). Invocation connectors, channel connectors and top-level loops are not stored in the repository for reasons that are explained later. During connector
composition, only connector templates are used.

Connector instances are the result of instantiating a connector template for component composition/adaptation. They are attached to specific components, and composition operators obviously have their arity fixed. Connector instances are different from connector templates, because during component composition a connector is used to compose specific components via their interfaces. Therefore it must contain additional information related to its constituents’ interfaces. For example, if a composite component is defined using a binary sequencer, it does not suffice for the sequencer to only know that each sub-component is invoked sequentially. The sequencer connector must additionally contain information describing which method of the composite triggers which methods of the constituents. This information is clearly not applicable during connector composition.

As mentioned earlier, invocation connectors, channel connectors and top-level loops are not stored in the repository. These are also generic since they are respectively not connected to any specific computation unit, active computation processes, or assembled system. However they are not stored in the connector repository because they cannot be used for connector composition. Instead, the invocation and the channels connectors are only used by the component builder when defining a new passive or active atomic component respectively. Top-level loops are only used when an assembled system transits into run-time.¹

During connector composition, exactly one connector template acts as the operator of the composition and one or more connector templates are the operands. In Figure 6.1 we present the different kinds of connector template that reside in a connector repository and what their roles are during composition.

1As will be shown in the implementation Chapter 7, the invocation and the channels connector are stored in the repository. This is done purely for the simplicity of our implementation and for creating an intuitive user interface.
connectors may have their arity fixed, or they may be parametric. Only composition connectors with fixed arity can act as composition operators. When the arity is parametric the connector can be only used as an operand and by specifying its arity parameters it can also act as an operator. Finally, adaptation connectors are unary and cannot be used as composition operators.

Currently, we do not allow our model to describe compositions where the arity of the connector operator is not fixed. For example we cannot define a composite connector where “all $n$ branches of the connector operator connect to some connector $C$”; $n$ has to be fixed prior to composition. Pragmatic reasons dictate this decision, namely that an $n$-ary composition operator would lead to an overly complicated syntax and implementation.\footnote{We believe that this restriction does not significantly limit our ability in expressing a wide range of ‘useful’ composite connectors. This is discussed in more detail in Section 9.3.}

We have defined a syntax in pseudo-BNF for constructing connector templates, which is given in Figure 6.2. Before presenting the syntax in detail, we discuss some conventions used. Tokens $\text{con tmpl name}_p$ and $\text{con tmpl name}_f$ are used to uniquely identify connector templates in the repository by their name. Suffixes ‘$p$’ and ‘$f$’ respectively denote connectors whose arity is parametric or fixed. When defining new connector templates, the templates must be given unique names and a similar convention is used. We use $\text{tmpl def name}_f$ to denote that the arity of the new connector is fixed, and $\text{tmpl def name}$ to denote that arity of the new connector template can be either fixed or parametric. Token $\text{arity}$ denotes a natural number, greater than, or equal to two. Finally, all the above names are considered to be strings, but for brevity of the syntax we
A new connector template $tmpl\_def$ can be defined in two ways. The first one is by fixing the arity of a parametric connector template already existing in the repository and the second one is via composition. For the first case, the expression $fixArExpr$ identifies a connector template $con\_tmpl\_name\_p$ from the repository, and a sequence of $arity$ tokens are used for specifying the new connector’s arity. As will be explained later in this section, more than one arity token may be used. For example, if $PIPE$ denotes the $n$-ary pipe in a repository, then $pipe2 = PIPE(2)$ defines the template of a binary pipe.

The result of connector composition is a new composite composition connector, as we do not form ‘composite adaptors’. The arity of the new connector template with the name $tmpl\_def\_name$ can be either fixed or parametric, but cannot yet be decided because it depends on the composition expression. The expression uses a connector template with fixed arity $con\_tmpl\_name\_f$ as the composition operator, and a sequence of arguments. The number of arguments equals the arity of the composition connector. An argument $arg$ in the composition expression can be either another expression, so that a connector template is defined recursively, or a parametric connector template name $con\_tmpl\_name\_p$, or a composition place $cp$, or a connector template with fixed arity $con\_tmpl\_name\_f$ that is parameterised with a list of composition places $cps$. If at least one parametric connector is used in the composition expression, then the new connector is also parametric, otherwise its arity is evidently fixed. Composition places are used to denote that the respective connector branch can be replaced by “any” component or connector in subsequent compositions. The role of the index $i$ is explained later in this section.

As an example of connector composition we discuss the observer pattern that was briefly presented in Section 3.4. In the observer pattern there are two roles, that of a publisher and that of the subscriber(s). After the publisher publishes a value, all subscribers get notified, ideally concurrently, about the value. For this pattern we use a pipe and a cobegin connector as shown in Figure 6.3. The syntax for the observer connector’s definition is $OBS = pipe2(CP, CB)$, where $pipe2$ is the binary pipe template defined earlier. $CB$ identifies the $n$-ary cobegin. As a convention we are using lower case letters for connector templates with fixed arity and upper case letters for connector templates whose arity is not fixed. The
CHAPTER 6. COMPOSITE CONNECTORS

The observer design pattern as a composite composition connector.

arity of the new connector OBS is \((n + 1)\), where \(n\) is the arity of the cobegin. Before OBS can be used as an operator for connector or component composition, the variable \(n\) needs to be defined. For example, by declaring \(obs = OBS(3)\) we define a connector template \(obs\) with three subscribers.

In Figure 6.3 the usage of composition place “CP” is illustrated. Whenever “CP” is used as a parameter to the composition operator it means that the respective branch of the connector operator does not connect to another connector template, but instead it remains unconnected or ‘dangling’. In order to denote two or more dangling branches “CP” is also used. For example, the equation \(conn' = conn(CP, CONN, CP)\) denotes that for new connector template \(conn'\), a ternary connector operator \(conn\) is used whose first and last branches are dangling and the second one connects to a parametric connector. By default, when “CP” is used, it means that the unconnected branches are different. In order to denote that the unconnected branches refer to the same branch, we index “CP” over the natural numbers. An example is shown in Figure 6.4. In

the figure, the quinary connector operator \(connOp\) composes two binary connectors \(c1\) and \(c2\) and an \(n\)-ary connector \(C\). Some of the branches of the connected connectors are shared. The composition equation for \(C\_CONN\) is:

\[ C\_CONN = connOp(CP, CP1, c1(CP1, CP2), c2(CP2, CP), C) .\]

Composition places can only be defined for connectors with fixed arity. The arity \(C\_CONN\_ar\) of \(C\_CONN\), and of any composite connector template in general, is an expression that can be trivially calculated as the number of unconnected branches, i.e.
the number of times that the token “CP” appears in the composition expression, plus the number of different “CP_i” occurrences, plus the arities of connector template operands. Assuming that the arity of \( C \) is \( n \), for the above example we have that \( \text{C}_\text{CONN}_{\text{ar}} = (2 + 2 + n) = 4 + n \).

We mentioned earlier that more than one arity token may be used when fixing the arity of a parametric connector template. This was also reflected in the connector template definition syntax for the token \textit{fixArExpr} in Figure 6.2. This occurs when two or more parametric connectors are used as operands of the composition. As an example, we define the composite connector \( \text{CONN} = \text{conn}(C, C') \), where the arities of \( C \) and \( C' \) are \( m \) and \( n \) respectively. The arity of the composite connector \( \text{CONN} \) can be trivially computed as \( m + n \). Consequently, when fixing the arity of this connector, the component developer needs to provide two numbers as input.

It worth noting that we have been able to compose composition connectors in an algebraic way because our connectors have the property of being \textit{structurally self-similar}. Structural self-similarity for composition connectors means that every connector consists of a single upper branch and of a set of lower branches. The single upper branch is used for receiving connections from higher-level connectors and the lower branches are used to connect to other connectors/components. All branches are also used for control and data i/o, but this forms part of their behaviour and is discussed in detail in Section 6.2.

When composing connectors according to the syntax in Figure 6.2, the result of the composition is a connector template that is structurally self-similar to the composed connectors. The new connector has a single upper branch, the one of the connector operator, and its lower branches consist of (some of) the lower branches of the participating connectors. Consequently, composite connectors can be further used in composition either as operators or as operands, and this defines our algebraic composition. Structural self-similarity and algebraic composition have the effect that starting from a connector repository containing a small set of composition (and adaptation) operators, any number of composition operators of arbitrary complexity can be defined. An evaluation of how useful algebraic composition is in practice is given in Section 9.3. Typically a connector repository is initiated with the connectors presented in sections 4.5, 4.6 and 5.1. We henceforward name these connectors ‘basic’, as opposed to ‘composite’. Connector templates obtained by fixing the arity of \( n \)-ary basic connectors, for example
the binary pipe2 that was earlier defined, are also considered as basic.

6.2 Semantical composition

Structural composition, as its name suggests, only defines the structure of the composite connector in terms of the constituent ones. Algebraic expressions have been defined for that purpose. In this section we describe the semantical composition of connectors. Semantical composition allows us to derive the control-flow behaviour of the composite connector based on the composition expression, i.e. the structure of the composition, and on the control-flow behaviours of the connectors participating in it.

To that end, Coloured Template nets (CT-nets) were introduced in Section 4.5.1 for representing the behaviour of our connectors in terms of their control flow. As was discussed in that section, CT-nets were introduced because the semantics of Coloured Petri nets (CP-nets) do not allow their composition and extension in order to form larger CP-nets. CT-nets are used for describing the behaviour of both primitive and composite connectors in the connector repository and the CT-nets for all our basic connectors are given in [80]. In this section we define compositional rules for the CT-nets representing our connectors. This means that we use the CT-nets of the connector operator and of the connector operands in order to define the CT-net representing the control flow of the composite connector. This has been possible because CT-nets are compositional.\footnote{Hierarchical Petri nets offer similar functionalities but they are unsuitable for our purposes. This is explained in detail in Section 6.4.2.}

A generic CT-net representing the control flow of any of our connectors is shown in Figure 6.5. A CT-net consists of three parts. In the upper part an input place \( \text{In} \) and an output place \( \text{Out} \) are defined. Its lower part consists of a set of composition places \( CP_1, \ldots, CP_n \), where \( n \geq 1 \) represents the arity of the connector. Obviously, when \( n = 1 \) an adaptation connector is presented. The middle part of the CT-net defines the control flow of the connector. Control flow is captured by the flow of tokens from the input place \( \text{In} \) to the output place \( \text{Out} \) as is explained below.

Tokens in CT-nets represent control and they are typed as pairs of natural numbers with a case identifier, \( \mathbb{N} \times CID \). The case identifier distinguishes among requests of different users of a connector. Users of a connector are other, higher
level connectors. However, since our model is hierarchical, all case identifiers originate from the single, top-level connector. For example, in Figure 5.8 the processes within the sub-loops $L_i$ of the parallel top-level loop, all have different case ids. The role of the natural number is explained later in Section 6.2.1.

Since tokens represent control, their flow in a CT-net represents the control flow of the connector. Input, output and composition places of a connector are typed as $\mathbb{N} \times CID$ in order to accommodate control tokens. Initially, control tokens are expected to arrive at the $In$ place via higher level connectors as explained above. Then, via the control flow of the CT-net, control tokens are expected to arrive at some (or all) of the composition places. The control flow behaviour is described by a set of arcs and transitions in the middle part of the CT-net. In the figure we only show the arcs that connect to and from the lower places of the connector ($in_1 \ldots in_n$, $out_1 \ldots out_n$) and the transitions attached to these arcs ($T_1 \ldots T_n$, $T'_1 \ldots T'_n$). As soon as all control tokens return from the composition places, the original control token is placed into the $Out$ place. This designates that execution for the given control token has finished.

So far we have not explained the exact nature of composition places. Composition places represent an entity, be it a connector or a component, and each one has precisely one incoming and one outgoing arc, representing the input and the output of the control respectively. In any CT-net they are by definition different from each other, because a composition connector composes different entities. Composition places have a significant role in connector composition and this is explained later.

For each of our basic connectors it is important to make a fundamental observation when they are treated as black-box entities. By ‘black-box’ we mean that
we are only interested in their interface behaviour. Given that we have described
our connectors as CT-nets, their black-box behaviour is captured by only looking
at the behaviour of the In and Out places. Any observation made is important
because it represents the common denominator of all our composition and adap-
tation connectors, regardless of their control flow and their arity. We describe
this observation on basic connectors as the following property:

**Property 1** The control flow of each connector guarantees that each token in
the input place will eventually flow to the output place of the connector. In the
output place there can only appear the tokens that have previously appeared in
the input place, and for each token in the input place there will be exactly one
token in the output place.

Property 1 simply states that incoming control processes do not vanish through
connector execution, and only those processes return as a result of the connect-
or executing. This property could be trivially proved for the basic connectors,
under the assumption of course that processes in the stateful facade return im-
mediately when the facade is not in the correct state. This property is important
because it defines the *semantical self-similarity* of our basic connectors, i.e. what
it means for our connectors to be similar in terms of control flow. Because of
semantical and structural (Section 6.1), self-similarity, we can use our basic com-
position connectors both as operands and as operators in connector composition.
Semantically, even n-ary composition operators can be used as the operator of the
composition, provided there is a powerful enough syntax to describe this compo-
sition. However, our syntax restricts that only composition operators with fixed
arity can be used as composition operators. Since semantical self-similarity holds
when the arity of the composition operator is parametric, then it clearly holds
when its arity has a specific value, and it therefore holds for connector templates
with fixed arity. In the following discussions we consider that the composition op-
erator can be n-ary. This has the advantage that future extensions of the syntax
will have solid semantical foundations.

During connector composition, we derive the CT-net of the composite con-
nector from the CT-nets of the operator and operand connectors used in the
composition equation. First, we describe the process when no places are shared
among the participating connectors, and in Section 6.2.1 we describe the case
where places are shared.

In Figure 6.6 we present how semantical composition is achieved for a generic
6.2. SEMANTICAL COMPOSITION

connector composition when no places are shared among the participating connectors. A $k$-ary connector $conn$ composes $k$ connectors $C_1, \ldots, C_k$. The details of this composition are presented later in this section, after Property 2 is defined, but in essence the result of the semantical composition is a CT-net that defines the control flow of the composite connector. This resulting CT-net is based on the control flows of the operator and operand connectors.

By virtue of the composition process, the control flow of the composite connectors is similar to the control flow in the basic connectors. This forms Property 2:

**Property 2** Property 1 holds for composite connectors that are formed from basic operators according to our composition syntax (see Figure 6.2), with the restriction that no places are shared during composition.

Due to this property it is guaranteed that the only control processes returned from a composite connector are the ones that are given as input to the connector. This fact, along with the structural self-similarity defined for any composition connector, implies that composition connectors can be used both as operators and as operands in further compositions, in a similar fashion that basic connectors are. We next describe in detail the process followed for composing connectors into composites.

For connector composition, our starting point is the composition expression. For example, we consider the case shown in Figure 6.6, where all $k$ composition places of connector operator $conn$ are replaced by the CT-nets of the connector.

Figure 6.6: Generic connector composition with no shared places.
operands. As the first step of the procedure, we remove each composition place on
the CT-net of the operator connector \( \text{conn} \), that is not “CP” in the composition
expression, but we retain the arcs pointing to and from the composition places
as ‘dangling’ arcs. In the figure, all composition places \( C_i \), \( 1 \leq i \leq k \), of \( \text{conn} \)
are removed, and all \( \text{in}_i/\text{out}_i \) arcs are retained. Next, based on the composition
expression, we re-direct the dangling arcs from the first step to the in/out places of
the CT-net of the connector operand at the appropriate branch of the composition
expression. Each ‘dangling’ arc \( \text{in}_i \) is redirected, so that it points to the input
place of the replacing CT-net. Similar hold for the outgoing arc \( \text{out}_i \); its source
becomes the output place of the replacing CT-net. As shown on the right-hand
side of the figure, the CT-net for the composite connector is structurally similar
to the CT-net of any other connector in the system. In order to be seamlessly
used in further composition, Property 2 must additionally hold. The replacement
procedure minimally affects the control flow of the composition connector, and it
therefore ensures Property 2 holds as is informally explained below. Although a
formal proof would be possible, this is out of scope of this thesis.

In the composition, Property 1 holds for the connector operator because it
is a basic connector. The CT-net of the operator connector is similar to the
generic CT-net in Figure 6.5 and it is guaranteed that each token in place \( \text{In} \)
eventually flows to place \( \text{Out} \). During connector composition, some (or all) of its
composition places are replaced by the CT-nets of other basic connectors. We
check how this replacement affects the control flow of the operator connector and
consequently we check if it affects Property 2. In the CT-net in Figure 6.5, after
\( T_k \), \( 1 \leq k \leq n \) fires some token \( t \), \( t \) is \textit{immediately} placed in the output place
of the arc \( \text{in}_k \) and in the input place of the arc \( \text{out}_k \), namely token \( t \) is moved
into place \( \text{CP}_k \). When \( \text{CP}_k \) is replaced with the CT-net of the connector operand
according to the rules described earlier, the only difference after \( T_k \) fires some
token \( t \), is that \( t \) \textit{eventually} becomes available in the input place of \( \text{out}_k \). This
is true when the replacing connector is one of our basic connectors because of
Property 1. Therefore the only effect the replacement has on the control flow
of the operator connector, is \textit{when} \( t \) will flow through incoming transition \( T'_k \),
and not \textit{if} it will flow. Consequently, Property 1 can be extended for composite
connectors formed when the connector operator is a basic composition connector
and the operand connectors can be any of the basic ones, including adaptation
connectors. This extension forms Property 2. Inductively it can be shown that
Property 2 holds for any composite connector regardless of whether they were defined from basic or composite connectors, as long as no places were shared during composition.

### 6.2.1 Shared places

When no places are shared during composition, the possible control-flows that can be described using our connectors, are limited. For example, given two composition places \( CP \) and \( CP_1 \), it is impossible to describe a connector that executes \( CP_1, CP, CP_1 \) in that order. With sharing of composition places, a connector offering this control flow can be trivially defined as the composition \( \text{seq2}(CP_1, \text{seq2}(CP, CP_1)) \), where \( \text{seq2} \) is a binary sequencer template and the order of its parameters signifies the order in which they are executed.

In the case of sharing of composition places, some additional complexities arise, that although they do not affect structural self-similarity, they can invalidate semantical self-similarity for composite connectors as was expressed in Property 2. We go through these complexities and show how they are addressed via a generic example. As a representative example of composition with sharing of places, we use the composite connector \( C\_CONN = \text{connOp}(CP, CP_1, c1(CP_1, CP_2), c2(CP_2, CP), C) \), whose structural composition was presented in Figure 6.4. The semantics of this composition are shown in Figure 6.7. In the figure, we name all composition places according to the names in the composition expression. The replacing rules are the same as when no places are shared. The
effect is that the CT-nets of some connectors share composition places. Specifically, places \( CP_1 \) and \( CP_2 \) are shared between connectors \( connOp \) and \( c_1 \), and \( c_1 \) and \( c_2 \) respectively.

A potential problem of this composition is manifested when a token \( t \) is placed in a shared place as a result of the control flow of a connector. For example, if token \( t \) appears in place \( CP_1 \), and when our basic connectors have the generic form presented in Figure 6.5, token \( t \) will non-deterministically flow either to \( connOp \) or to \( c_1 \). If the token originated from \( connOp \), and it is transferred to \( c_1 \) directly from \( CP_1 \), then this is an error. We consider this as an error, because each of our connectors \emph{encapsulates} control. This means that when a control process is input to a connector \( connOp \) (represented as token \( t \) above), that control process cannot be transferred to another connector unless this happens through the control flow of the same connector, \( connOp \). In other words, each token in a shared composition place, must return to the same connector whence it originated from, before flowing to another connector. In the previous example, since token \( t \) is placed in place \( CP_1 \) by connector \( connOp \), it must return to the control flow of the same connector, before being transferred to the control flow of connector \( c_1 \).

In order to enforce the correct behaviour for a composite connector, it is imperative that each individual connector can guarantee the following properties:

**Property 3** If all tokens placed on the input place of a CT-net are distinct, then the same token can never appear simultaneously in more than one of the net’s input/output or composition places.

**Property 4** Given Property 3, each connector is able to distinguish in all its composition places tokens that are the result of the control flow of the connector itself.

Property 3 is an auxiliary property and is used for (informally) proving the more interesting Property 4. As is also explained in the discussion that follows, Property 4 suffices for enforcing the correct behaviour on composite connectors with shared places. We start with the informal proof of Property 3.

As mentioned earlier, tokens represent control and are typed as \( \mathbb{N} \times CID \), where \( CID \) stands for ‘case id’. Since tokens represent control, they only appear at runtime. The tokens for any executing component based system built according to our component model, are either defined when the assembled system transits
into run-time, or they are created dynamically while the system is executing. In
the first case, the tokens are either defined by the top-level loop connector of a
system, Section 5.2.1, or they are used to define the control flow of active atomic
components, Section 4.3. In both cases, all these tokens have distinct case ids.
At run-time, new tokens are created when an executing process splits into two or
more sub-processes, Figure 3.9(iii). In terms of CT-nets, this means that from a
single control token, two or more tokens are created. Therefore, when a system
first transits into run-time, all control tokens in the executing system are unique.
In order to prove Property 3, we additionally need to guarantee that when a
connector creates new tokens for executing sub-components, these new tokens
are also unique.
From our basic connectors, only the cobegin, the efficient cobegin, and the
finite parallel loop, presented in Sections 4.5.2 and 5.1.1 respectively, may create
new tokens. Their CT-nets are presented in [80]. To aid the discussion, we present
the CT-net for the cobegin in Figure 6.8. For the other two connectors, similar

\[\text{Figure 6.8: CT-net for a binary cobegin.}\]

hold w.r.t. the generation of new tokens.
In Figure 6.8, control flow starts from In place and transition SPLIT is im-
immediately enabled. After SPLIT fires, the two composed entities CP1 and CP2
execute concurrently. Transition JOIN then fires and two control tokens are
merged into one and are returned to the Out place. In the figure, there are two
special places, mem and TID. The role of the mem place is related to Property 4
and is discussed later. The role of the fusion place TID is to guarantee that
the connector creates unique control id tokens, i.e. pairs of type $N \times CID$, that
are placed to the composition places. In order to do so, for every control token
$(i, c) \in N \times CID$, that splits, we vary the first part of the pair, i.e. $i$. This is
because the value of $c$ is unique, as it is created from one of the top-level loops
of the system. Since the cobegin, the efficient cobegin and the finite parallel loop generate new tokens via a single \textit{SPLIT} transition, and given that tokens in their \textit{In} places have different \textit{CID}, it is guaranteed that the newly generated tokens are unique, by sharing among all the above connectors the fusion place \textit{TID}. \textit{TID} contains the last used value of \textit{i}. It is initialised with the value 1, and every time a process (token) is split, it gets augmented by the number of the split tokens (two in the case of the binary cobegin). As a result, and given that the control flows of the above connectors do not output the same token in any of their own composition places, all \((i, c)\) pairs in the input/output and composition places of the connectors are unique. We are not interested in the other places of connectors, because these are hidden and do not affect composition. An extended discussion on the exact role of \textit{TID} and on its implications, is provided in [80].

Since Property 3 holds, we proceed by informally proving Property 4. To that end, we equip each connector with memory places, so that the connector is aware of which of the tokens in its lower places, are a result of its own control flow. The generic control flow of our connectors, when equipped with memory places, is shown in Figure 6.9. In the figure, places \textit{mem}_i, 1 \leq i \leq n, are equipped with copies of the tokens that are placed in \textit{CP}_i as a result of \textit{T}_i firing. Consequently \textit{T}’_i only fires for ‘correct’ tokens, i.e. for tokens placed in \textit{CP}_i as part of the control flow of the connector. In other words, even if \textit{CP}_1 is shared with the control flow of another CT-net, the two control flows will be able to distinguish, and sub-sequently retrieve, their ‘own’ tokens. Because of Property 3, and given that the two control flows receive unique tokens in their input places, they cannot put the same token in any of their composition places, including the shared ones.

![Figure 6.9: Generic control flow of our connectors.](image-url)
Consequently, if we show that the two control flows can only receive unique tokens in their input places, then Property 4 is true. The two control flows only receive distinct tokens in their input places, because, and as was explained earlier, there are limited ways for producing tokens in our model, and all of them have been demonstrated to create unique tokens. The top-level loop creates unique tokens by varying $CID$, and, whenever new tokens are generated, number $i$ in each $(i, c)$ token is increased. Consequently, it is not possible to have the same token into the input place of any two connectors.

Regarding memory places, we can note that their distribution and their exact type in the basic connectors can be different from each other. This depends on the control flow expressed by the particular connector. In Figure 6.8, the memory places in the CT-net for the cobegin connector were shown. As can be seen in the figure, they are typed as $N \times N \times CID$. The first number in the triple is used for returning to the $Out$ place the correct token that was input in the $In$ place.

Since Property 4 holds, the potential problem that was described in Figure 6.7, can no longer arise. Therefore, we can now extend Property 2 for the case of composite connectors, where some places are shared among the constituent connectors. This forms Property 5:

**Property 5** The control flow of each connector guarantees that each token in the input place will eventually flow to the output place of the connector. In the output place there can only appear the tokens that have previously appeared in the input place, and for each token in the input place there will be exactly one token in the output place. This also holds for composite connectors even when the constituent connectors share some, or all, of their composition places.

This property is important because it provides a semantically sound basis to our algebraic composition. Recall that algebraic composition is a direct result of the structural and semantical self-similarity of basic and composite composition connectors.

Although the previous discussions do not constitute a formal proof of Property 5, which is out of scope of this work, they increase our confidence in that the proposed semantical composition is sound with respect to being algebraic. Algebraic composition is an immediate result of our composition connectors, either basic or composite, being self-similar. Therefore, every composite connector can be further used in composition, either as an operand or as an operator.
We have now defined structural and semantical composition. In the rest of this chapter we present related work and we then discuss some issues related to connector composition. We conclude in Section 6.5 by providing a summary of the chapter.

6.3 Related work

To the best of our knowledge, our composition of connectors is unique. In our comparison we have not limited the scope to the component models that were discussed in Section 2.3. Instead, we have taken into account work in connector composition in the broader field of software engineering. As is explained below, in related literature ‘connector composition’ usually refers to the construction of a single connector out of some primitive elements that are not connectors.

In the SOFA component model [39], connectors have been defined as a “composition of connector elements” [40]. Given a set of connector elements, these are ‘composed’ in order to form a single connector according to the connector’s architecture. Connector elements and their composition, define the functional and non-functional properties of the connector, for example encrypted and distributed communications. A connector builder is used for generating the connector’s implementation based on the connector elements, on its architecture and on the interfaces of the connected components. In [60] the work has been extended to provide support for heterogeneous components, i.e. the generated connectors can be used to connect components spanning differing component models. However, in work done in the context of the SOFA component model, connectors are not composable. They are constructed from basic elements, but they cannot be composed into larger, composite connectors. Additionally, the connector themselves are not meant to be re-used. They form a customised solution for a specific system.

In [120] connectors are adapted by applying a set of transformations on them. Examples of transformations include data transformations for changing the format of data exchanged between components, or the addition of a new participant in a connection, for instance by making a binary communication a ternary one. The new participant may for example be only observing exchanged messages. Transformations are composable, in the sense that two transformations can be composed into a larger one. However, composition of transformations is largely
an ad-hoc process, where the developer has to manually specify how the transformations can be composed. A tool is then used to generate the code for the new connector, based on the code of the constituent ones and of the composition instructions. More importantly, this work does not compose connectors into composite connectors. Instead, connectors are adapted, or transformed, to meet the requirements of specific communication patterns. Connector adaptation also involves the modification of the source code of components, since the connector code is embedded within the component code and connectors are not distinct entities from components.

The work presented in [127] focuses on building connectors out of a rich set of building blocks. Connectors consist of input and output ports, and channels that connect these ports. By varying these elements, different synchronisation policies can be defined for concurrently executing components. For example it is possible to enforce variations of synchronous/asynchronous communication. However in this work connectors cannot be composed to form larger, composite connectors.

In [86] the notion of a higher-order connector (hoc) is defined. A hoc is “a connector that takes a connector as parameter”. It has the form $bConn(pConn)$, where $bConn$ is the body connector that defines an independent aspect to be applied to instances of the parameter connector $pConn$. An independent aspect can be for instance compression, fault-tolerance, security, etc. $pConn$ provides an abstraction of the kinds of connectors that $bConn$ can be applied to. For example, if $pConn$ describes one-way communication, before the hoc $bConn(pConn)$ can be used to connect components, $pConn$ needs to be instantiated with a basic connector describing exactly how one-way communication is performed, for example synchronous or asynchronous. But in the approach presented in [86], not any connector can be a body or a parameter connector, as they describe distinct roles in the composition that are not interchangeable. This contrasts with our approach where composition connectors can be used both as operators and as operands. Most importantly, in our approach a composite connector can be further composed, while in their approach composition of higher-order connectors is undefined.

In the Reo coordination language [22, 23] a rich set of low-level connector primitives (channels) for coordinating data flow among components is defined. Control lies within components and connectors transfer passive data, i.e. data
that do not transfer control. Channels can be composed to form larger connectors of arbitrary complexity. Since connectors are formed of channels, they can be also composed into composite connectors. However, composition of connectors is performed by joining channel ends (nodes). This is different to our approach where composition is performed by explicit composition operators. Additionally, the nature of connectors in Reo is very different to that of our connectors. Reo connectors coordinate a set of (active) components by controlling the data flow among components and they cannot be used for component invocation. In our approach, connectors exercise control by explicitly invoking sub-components. Control in Reo lies in components and connectors are passive. In our approach and with the exception of active atomic components, control starts from the (top-level) connectors.

6.4 Discussion

In this section we discuss issues related to connector composition that have not been discussed earlier in the chapter. First, we show the significance of composite connectors for design phase component composition. Next, we present the related notions of hierarchical Petri-nets and we explain why they are insufficient for describing connector composition. A relevant issue to connector composition has to do with the practical applications of the formal CT-net semantics that were introduced in Section 6.2. This forms part of future work and is discussed in detail in Section 10.2.2.

6.4.1 Composite connectors and design-phase composition

Composite connectors have a significant role during design phase composition. Their significance lies in that they allow us to overcome a limitation regarding the expressiveness of design phase composition. As was discussed in Section 3.3, during design phase composition, copies of components from the component repository are obtained. Then, a composition operator is applied and a composite component is immediately formed that is stored back to the repository. The composite component is an encapsulated entity that can only be accessed through its interface and whose internal structure is completely hidden.
Given a pre-defined set of connectors, for example our basic connectors, we are limited regarding the control flow that can be applied for any given set of components. For example, in Figure 6.10(a) composite component \( CC \) is formed using composition connector \( c1 \). Directly accessing the sub-components of \( CC \) in another composition by connectors \( c2 \) and \( conn_{-}op \), would break the encapsulation of \( CC \). As a result, it would not be possible to describe that composition in the design phase. Expanding the initial set of ‘basic connectors’ offers no solution to the problem, because however large a set of basic connectors we define, there will be always some control flow not in that set.

Connector composition overcomes this problem, as long as the initial set of basic connectors and their composition is sufficiently expressive.\(^4\) By defining a composite connector \( compConn \), composition occurs in a single step as shown in Figure 6.10(b), thus preserving encapsulation. The definition of such connector is shown in Figure 6.11. This composition is straightforward, and is exactly the same shown in Figure 6.10(a), where instead of components, composition places are used. This is possible because our connectors can be used to compose both connectors and components.

\(^4\) An evaluation of the expressiveness of our connectors and connector composition is presented in Section 9.3.
6.4.2 Hierarchical Petri-nets

In [71, 73] hierarchical Petri-nets (HP-nets) have been defined. In HP-nets, substitution places and substitution transitions are introduced that can be substituted by other Petri-nets. Rules are defined that precisely describe how a substitution place/transition can be replaced by the substituting CP-net. Consequently, it could be argued that composition places in CT-nets are actually substitution places in [71, 73], on the grounds that in our approach, we replace composition places with CT-nets of other connectors.

However, purpose of HP-nets is to be used as a graphical convenience for large Petri-nets. HP-nets are actually flat Petri-nets that have been decomposed into smaller Petri-nets. A superpage defines all substitution places/transitions, which are in turn defined as (probably hierarchical) Petri-nets in separate pages. In our approach, a connector is defined in terms of its composition places and this represents the ‘whole’ connector. When composition occurs, a new CT-net is defined via structural modification of the CT-nets of the constituent connectors according to our composition rules. In other words, our approach is compositional, instead of the decompositional approach followed by HP-nets.

6.5 Chapter summary

In this chapter we presented the syntax and semantics of connector composition. The life cycle of connectors and its relationship to the component life cycle were demonstrated. By surveying related work in the broader field of software engineering, we provided evidence that our approach is unique in related literature. Finally, the importance of composite connectors during design phase component composition was discussed.

With the current chapter we have completed the presentation of all aspects of our model. In Chapter 7 we present its implementation.
Chapter 7

Implementation

We have implemented our model in Java and we have developed a graphical user interface for assisting the component developer. Our implementation integrates the component and connector life cycles that were presented in Sections 2.2.1 and 3.4 respectively. The structure of implementation is shown in Figure 7.1. The GUI layer consists of the component and connector builders, the component assembler, as well as the component and connector repositories. In the model layer, we have developed Java APIs in order to support composition and execution of both components and connectors. The repository APIs simply allow reading and writing components into the local file system. As will be explained later, components and connectors in the repository are XML files. Since our implementation is in Java and we do not currently support distribution of components/connectors, and the implementation is executed inside a single Java Virtual Machine.

The Java API for the design phase composition only captures the structure of components, whereas the deployment phase API captures the execution semantics as well. The “less than” symbol in the figure denotes that the first is derived from...
the latter and apparently it is a subset of it. It should be mentioned here, that the purpose of our implementation is only to serve as a proof of concept for the notions presented in our model.

We begin our presentation with the deployment phase API of our model in Section 7.1. This API constitutes the cornerstone of our implementation and reflects the semantics of our model as these were presented in Chapters 3 – 5. Then, in Section 7.2 we focus on the most complicated parts of our API, namely active atomic and composite components. In the following two Sections 7.3 and 7.4, we demonstrate how our tool can be used for building a simple system. The case study that we use is the authenticated flight monitoring system that was presented in Figure 4.30. We use the component builder for the construction of the authentication AUTH and flights monitoring FLIGHTS components. These are two generic components that will be stored in the component repository. We use the assembler for building the composite CC via the stateful facade connector. Then we demonstrate its execution using the concurrent loop that was presented in Section 5.2.1.

In Section 7.5 we discuss the implementation of composite connectors. We conclude in Section 7.6 by summarising this chapter and by discussing some limitations of the implementation.

7.1 Deployment phase model API

In the this section we present an overview of the Java API for the deployment phase. The API we have developed reflects the Z type system of our model given in Chapters 3–5. In fact, the API defines the syntax of our model. As was discussed in Chapter 2, we have adopted the definition for a software component model presented by Lau and Wang in [84]. According to this definition, a component model must define the syntax, the semantics and the composition of components. In the previous Chapters 3, 4 and 5 we have explained in detail the semantics and the composition of our components. The deployment phase API presented in this section implicitly defines the deployment phase syntax of our model. We use the word implicitly, because we have not yet defined a language in terms of BNF [25], and certainly we have not defined a parser for it. The design phase syntax of our model is discussed in Section 7.3.

After presenting an overview of the deployment phase API, in Section 7.2
we provide more details on the implementation of active atomic and composite components. We focus on these aspects of our implementation, because they demonstrate the combination of data- and event-based synchronisation. We do not present the implementation of synchronisation constraints because it consists of simple usage of Java synchronisation primitives and of a basic round-robin scheduler (for avoiding starvation as was explained in Section 4.2.2).

The class diagrams for components and connectors are shown in Figure 7.2. Subtypes are created by inheritance. For example, the Atomic, Composite and

\[ \text{ActiveAtomic} \]

class diagrams are extensions of the Component abstract superclass. Similarly, connectors extend the Connector interface.\(^1\)

\[^1\text{In the actual implementation the naming a slightly different to reflect that this API corresponds to the deployment phase model. For example the Component superclass is actually} \]

\[ \text{Component} \]

\[ \text{Composite} \]

\[ \text{ActiveAtomic} \]

(a) Component hierarchy.

\[ \text{Connector} \]

\[ \text{SwingWorker} \]

\[ \text{Invocation} \]

\[ \text{CompConnector} \]

\[ \text{Loop} \]

\[ \text{ChConn} \]

(b) Connector hierarchy.

Figure 7.2: Class diagrams for component and connector types.
In Figure 7.2(a), it can be seen that every component is equipped with three public methods: `getMethods`, `execute` and `start`. `getMethods` returns a component’s interface as an array of `MyMethod` objects. As shown in Figure 3.3, every component’s interface consists of a set of triples `<name, ⟨input_param_type(s)⟩, ⟨output_param_type(s)⟩>`. Each triple is an interface element and programmatically this is captured as a `MyMethod` object. Every component’s interface is generated automatically during its construction. For passive components, a name denotes a method name. For active atomic components a name denotes a channel name. For active composite components a name can denote either, depending on whether it belongs to a passive or active sub-component. The `execute` method provides the means of executing the triples in the interface. The `start` method simply marks the transition of a component into the run-time phase of its life cycle, and as shown in Figure 7.2, it is only useful for active atomic components as will be discussed later.

The hierarchy imposed by our type system is enforced by our implementation. Therefore, every component contains a reference to its top-level connector (Figure 7.2(a)), and every composition connector contains references to its lower level components (Figure 7.2(b)).

Control flow in exogenous connectors follows the connector hierarchy. In Figure 7.2, it can be seen that all our components and connectors implement the `execute` method. The `execute` method of a component calls the `execute` of the top-level connector (Figure 7.2(a)). The top-level connector then calls the `execute` method of the sub-component(s) (Figure 7.2(b)) and so on until control reaches the lowest level in our hierarchy, namely atomic components. For passive atomic components, the invocation connector then dynamically invokes the method in the computation unit using Java reflection. For active atomic components the channels connector interacts with the active computation processes. The result from the invocation of atomic components then returns to the calling connector until the top-level connector is reached.

All our connectors are concurrent in that they can accommodate an unbounded number of concurrent requests. In the implementation, this is reflected by having the `execute` method not synchronised. This means that an unbounded number of users might be accessing different “copies” of the same method. As named `DeployedComponent`. Similar hold for the rest of the components and connectors. However, to avoid unnecessary confusion we present the simpler naming scheme.

\[2\]Within the limitations of the underlying machine of course.
we mentioned in sub-section 4.2.2 our connectors control concurrency via syn-
chronisation constraint. This is shown in the abstract Connector super-class.

Finally, in Figure 7.2(b) the top-level loop connectors are shown. The other
kind of loops, namely finite sequential and finite parallel loops are adaptors and
are not shown in the figure. As was explained in Section 5.2.1, top-level loop
connectors are a source of concurrent processes in our model (the other being
active atomic components) and are responsible for executing a deployed system.
As such they execute on their own thread. Because loops are used directly by
the GUI layer of our implementation, according to Java 6 rules for concurrency
in the GUI, they need to extend the abstract SwingWorker class.

7.2 Implementation of active components

The most interesting part of our implementation is how active components are
implemented. For active atomic components there is a contribution in the im-
plementation in that it mixes both data- and event- based synchronisation in a
single construct, namely the channels connector. In the implementation of active
atomic components, all synchronisation issues are handled transparently with the
channels connector. In this section we provide more details on the implementation
of active atomic and composite components.

7.2.1 Active atomic components

An active component provides on its interface a set of channels connected to
(a set of) CSP processes. The user can input data values to the channels via
the channels connector, and get back the results of executing the processes with
these input values, also via the channels connector. Similar to the methods of
passive components, each channel is defined by a name, the types of its input pa-
rameters and the types of its output parameters: \langle name, \langle input\_param\_type(s)\rangle, 
\langle output\_param\_type(s)\rangle \rangle. Thus the input data to the channels connector contain
the same data as that to a passive component, i.e. \langle (channel) name, \langle input\_param(s)\rangle, \langle placeholders\_for\_output\_param(s)\rangle \rangle. As was also shown in Fig-
ure 7.2(b), the channels connector is equipped with an execute method. There-
fore, the above input data are given as input parameters to the channels connec-
tor’s execute method.
The main implementation issue for active components is how to define and implement the interaction between the channels connector and the active computation processes. Our approach is shown in Figure 7.3. We define this interaction by special active agents that we call channel environments. These environments (chEnv0 and chEnv1 in Figure 7.3) are predetermined interaction points between the channels connector and the CSP processes. We define one channel environment for each channel connected to the CSP processes.

The channels connector is passive; it waits for external control threads and it then forwards the control signal along with any data to the chosen channel environment. CSP processes are active, and communicate via channels, and every communication on a channel is considered to be an event. Therefore, in order to communicate with a CSP channel, a channel environment must be able to generate and accept events. This implies that, channel environments need to combine both event-based and data-based synchronisation mechanisms, i.e. channels and semaphores, in order to communicate with the CSP processes and the channels connector respectively.

We implement the CSP processes in JCSP [2] (a Java implementation of CSP). In JCSP, a CSP process is defined as a Java class (CSProcess) that executes in its own thread. A JCSP process, contains JCSP channels, which are in turn defined as Java classes, and are responsible for implementing the synchronous behaviour of CSP channels. Therefore, a CSP process is implemented in Java using purely the API provided by JCSP. The actual behaviour of the CSP process, i.e. the order in which it reads from/writes to its channels, possible alternating reads on channels, etc., is written by the process developer. In essence, the process developer subclasses CSProcess, and appropriately overrides the related methods. Atomicity and other related properties, are handled by the JCSP implementation.

Given that CSP processes are implemented in Java, it follows naturally that
the channels connector is also implemented in Java. For a channel environment, the channels part is implemented in JCSP, using the relevant API classes. The semaphores part of the channel environment, is implemented using standard Java semaphores. Our implementation allows the user to choose any channel from the CSP processes to interact with (through channel environments). It is worth noting that channel environments are necessary to build as intermediate entities between CSP channels and the channels connector. The alternative to that would be for the channels connector to create appropriate channel connections dynamically for each user request. However, CSP does not define the notion of dynamic channel connections and neither does JCSP. Consequently it is not possible to directly implement our semantics in JCSP.

As already discussed, a channels connector takes as input, via its `execute` method, a triple `<(channel) name, (input_param(s)), (placeholders_for_output_param(s))>`. This is the method that performs the run-time functionalities of the channels connector and is outlined in Figure 7.4. When this `execute` method is called, the channels connector interacts with the chosen channel environment. The incoming thread of the channels connector is used to set the input values `inVals` (see Figure 7.3) of the channel environment to the input parameters of the call to execute. The channel environment is then signalled using semaphore `inSem` to communicate with the chosen (connected) channel. The `execute` method then waits on semaphore `outSem` for the signal from the channel environment that indicates that channel communication has finished. Output values `outVals` are then returned to the caller.

Channel environments are active and they operate inside a non-terminating loop, implemented as the method `run` in Figure 7.5. Their run-time behaviour is

```java
public void execute(String name, Vector<Object> inParams, Vector<Object> outParams) {
    ChannelEnvironment env = ((ChannelEnvironment) envs.get(name));
    // Set input values
    env.setInputParams(inParams);
    // Signal the ch.env.
    env.inSem.release();
    // Wait for signal that output is ready
    env.outSem.acquireUninterruptibly();
    // Get output values
    outParams = env.getOutParams();
}
```

Figure 7.4: Method `execute` of the channels connector.
Figure 7.5: Method `run` of the channels environment.

complementary to the `execute` method of the channels connector. They actively wait to be signalled by the channels connector that the input values have been set. Then these values are communicated to the connected CSP channel, and finally they signal the channels connector that communication has finished. The actual implementation of the `communicate` method depends on the type of the connected channel, for example an integer channel, a channel with no input/output values, and so on.

Our implementation ensures that the CSP semantics are preserved and this is reflected in the behaviour of the active component. The `communicate` call in the `run` method is not guaranteed to return immediately. This is because the `communicate` method contains a call to the channel part of the channels environment, which is responsible for communicating with the related channel of the CSP process. Because the communication between the channel part and the CSP channel is synchronous, it will not return until the CSP channel engages in the communication. For example, it is possible that the caller gets blocked until some values are input in another channel.

The above implementation of the channels connector is integrated into our tool. In Sections 7.3 and 7.4, a case study will be presented in building, deploying and executing the `FLIGHTS` active component.

### 7.2.2 Active composite components

We now discuss the implementation of active composite components. For this presentation we are using as an example the authenticated flight monitoring composite (CC) in Figure 4.30(a). Constructing an active composite comprises of several steps. Firstly, the system developer must define the state (data) of the facade. We define these data as a set of triples of the form \( \langle \text{name}, \text{type}, \text{value} \rangle \).
For the authenticated flight monitoring system, the connector’s data need to keep track of which users have been authenticated. Therefore, initially the data are empty. As soon as a user authenticates by using the check method of the AUTH component, via the composite CC, their card number is stored to the facade’s data. For example, for the user with card number 100, the data added will be: (100, Boolean, Boolean, TRUE).

Although for the authenticated flight monitoring system no initial data are needed, this is not always the case. For this reason every stateful facade contains a DataImporter object as shown in Figure 7.6. The DataImporter object contains a defineData method which is used by the constructor of the stateful facade for defining its data. As can be seen in the figure, the data of the facade are implemented as a Java class named Data that provides read/write access to individual data elements (Datum). The roles of the other classes are explained below.

As the second step for defining an active composite component, the system developer must define a condition that when true will enable each sub-component to execute. Each sub-component therefore gets associated with a boolean expression: if the expression is true, then the component executes. For the authenticated flight controller, the authentication component can be executed at any time, therefore no checks need to be made. For the flights component on the other hand, before it can be accessed, the user’s card number must belong to the facade’s data. The boolean expression therefore can be written as cardNo ∈ facade.data. Since in general the checking expression can be arbitrarily complex, we use a Checker object as is shown in Figure 7.6. For the authenticated flight monitoring system, the code for its canExecute method is shown below:

Figure 7.6: Simplified class diagram for the Stateful Facade.
CHAPTER 7. IMPLEMENTATION

```java
public boolean canExecute(String mName, Vector<Object> inParams) {
    // if the active sub-component is invoked
    if(mName.equals("ARR.allArvls") || ...) {
        String cardNo = inParams.get(0).toString();
        // if the card number exists in facade’s data then the
        // user has already authenticated
        if(facade.readDatum(cardNo) != null)
            return true;
        else // else not
            return false;
    }
}
```

As the third step for defining an active composite component, the system developer needs to define how the composite’s state gets updated after a call to a sub-component returns. This is because the facade’s data reflect the facade’s definition that the result of an invocation to a sub-component affects the next invocation. In order to update its own state, every facade connector contains an Updater object that has an update method, see Figure 7.6. That method is called when the execution of one of the sub-components terminates, and it may change the data of the facade. For the authenticated flight monitoring system, the overridden update method is:

```java
public void update(String name, Vector<Object> inParams,
                    Vector<Object> outParams)
{
    String userCardNo = outParams.get(0).toString();
    // if user is authenticated, add user to facade’s data
    if(name.equals("check") && outParams.get(1).equals(Boolean.TRUE)
        && outParams.get(2).equals(Boolean.TRUE)) {
        facade.addDatum(userCardNo, Boolean.class, Boolean.TRUE);
    }
    // else remove the user from facade’s data
    else if(name.equals("check")
        &&(outParams.get(1).equals(Boolean.FALSE)
            || outParams.get(2).equals(Boolean.FALSE))) {
        facade.removeDatum(userCardNo);
    }
}
```

Below we outline the code for the stateful facade’s execute method, i.e. the method that executes when we call the active composite component:
```java
public void execute(String name, Vector<Object> inParams, Vector<Object> outParams) throws Exception{
    synchronized(this){
        if(!this.checker.canExecute(name, inParams))
            // throw exception
    }
    // Locate the sub-component
    Component comp = this.locateSubComponent(name);
    // and execute it
    comp.execute(name, inParams, outParams);
    // Update the composite's state
    synchronized(this) {
        this.updater.update(names, inParams, outParams);
    }
}
```

First, it is checked if it is possible to execute based on the given method name and input parameters. If not, an exception is thrown. Otherwise, the sub-component to which parameter name belongs to is identified and executed. Finally, the internal state of the facade gets updated. The `synchronized` keyword is used to ensure that the facade's data checking and updating are mutually exclusive actions.

The above discussion concludes our presentation of active components. Next we present our tool for constructing and executing the authenticated flights monitoring system.

### 7.3 Component builder

We have developed our implementation as a module for the NetBeans platform [11]. The reason we have chosen NetBeans is because it offers some useful functionalities for generating the graphical user interface. For example, it offers implementation libraries for dragging and dropping visual elements, which is a functionality that we use when dragging components out of the repository into the builder.

In Figure 7.7 we present an overview of our tool. The main window of the tool contains tabs for the builder and assembler windows and currently the builder window is on focus. On the right hand side, there are the component and connector repositories. The component repository is a folder in the local file system.
containing some already defined components. Each component is a `.component` file containing its XML definition. The reason we have chosen XML is because it is extensible and hierarchical. Extensible means that we can easily define the mark-up elements relevant to our model. Being hierarchical means that it is straightforward to reflect the hierarchical nature of our model using the XML syntax. It is worth mentioning that as for the deployment phase API implicitly defining the deployment phase syntax of our model (see Section 7.1), so the XML schemata for atomic and composite components implicitly define the design phase syntax of our model. In the rest of this section we will present some examples of design phase syntax.

The connector repository is initialised with the predefined composition and adaptation connectors defined in Chapters 4 and 5. Although the invocation and channels connector shown as `INV` and `CH_CONN` in the figure respectively, should not be part of the repository, see discussion in Section 6.1, they appear in the repository for convenience when defining atomic components. These connectors are not composable. The repository may store composite connectors according to the semantics provided in Chapter 6. This is discussed in Section 7.5.

Atomic passive and active components are created by dragging and dropping the invocation (`INV`) and the channels connector (`CH_CONN`) in the builder’s
window. For passive atomic components, a wizard appears for defining the interface of the component as shown in Figure 7.8. The interface of the component is defined as a set of methods. These represent the interface elements discussed in Section 3.2.1. Then the component is stored in the appropriate format and is loaded into the component repository. The generated XML file for the `Fn-grChecker` component is shown in Figure 7.9. As can be seen in the figure, the XML definition of the passive atomic component declares the name of the component and the methods of its interface.

The password checker atomic component is developed in a similar way. Next, we compose the two components with the cobegin connector. In order to do so, the two atomic components are dragged from the repository and dropped into the builder. This creates copies of these components. Next, the cobegin connector is dropped into the builder and the appropriate wizard appears as
shown in Figure 7.10. The role of the first step (whose details are not shown in the figure) is to reorder the components prior to composition. Although this is unimportant with the cobegin, with some other connectors, for example with the pipe, it is important because it determines the order that these sub-components are invoked at run-time. In the second step, the developer chooses which methods of the two sub-components should be invoked concurrently when a method of the composite is invoked. Method check has already been defined and we show how method insert is defined by matching insertPwd and insertFngrPrt of the two sub-components.

Since both these methods take as input the user name, these two input parameters can be merged as shown in Figure 7.11(a). In Figure 7.11(b) the created interface for the composite is shown that consists of two methods. The created composite can be subsequently deposited into the repository.

In Figure 7.12 we present (part of) the XML definition for the composite component. As can be seen in the figure, the definition includes the name of the composite component, its defining connector type (number 4 stands for cobegin) and copies of the constituent atomic components.

Subsequently, the fingerprint-password checker is composed with a card reader
as was shown in Figure 4.30(b). A pipe is used for that composition. Using our tool, a wizard is presented that, as in the previous cases, guides the component developer in building the composite which is then stored in the repository.

Composition with the facade connector that was shown in Figure 4.30(a) will be presented using deployment phase composition in Section 7.4. Next we show how to construct an atomic active component using the builder tool. Namely, we construct the FLIGHTS components whose details were shown in Figure 4.16. For reasons that we explain below, that view of the FLIGHTS component was imprecise. In the following discussion we present the accurate, implemented, FLIGHTS component.

The first step when creating an active atomic component is to define it formally using CSP. The formal definition is then checked against common concurrency errors, namely deadlock and livelock freedom using the model checker tool FDR [5]. However, when modelling the active atomic component we cannot use
CHAPTER 7. IMPLEMENTATION

(a) Merging input parameters.  (b) Created interface.

Figure 7.11: Additional steps in the wizard for the cobegin composition.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<CompositeComponent>
  <ConnectorType>4</ConnectorType>
  <Name>FngPwdChecker</Name>
  <Method>
    <name>check</name>
    ...
  </Method>
  ...
  <AtomicComponent>
    <Name>AC1</Name>
    <Method>
      <name>checkPwd</name>
      ...
    </Method>
    ...
  </AtomicComponent>
  ...
</CompositeComponent>
```

Figure 7.12: XML definition of composite FngPwdChecker.component.

the full expressive power of CSP. This is because the Java implementation of CSP that we are using is not that expressive. Our implementation is based on JCSP [2], a Java library for CSP. Consequently, the active computation processes that we can model and implement depend on JCSP. The implementation details of how active computation processes and the channels connector interact are given in Section 7.2.1. In the case of the FLIGHTS component, there are two limitations. The first is that in JCSP it is impossible to define the ARR process so that it is always ready to either accept data from input channel addArr or to output data to channel allArvls. Although in CSP this is modelled using the external choice operator between the input channel addArr and the output channel allArvls, in JCSP it is impossible to define a process that alternates between an input and an output channel. In order to circumvent this problem, we precede the allArvls channel by the guard channel that performs no i/o. In JCSP alternation between
an input (\textit{addArr}) and a barrier channel (\textit{guard}) is allowed. After the guard has been invoked, all arrivals information may be obtained. Similar remarks of course hold for the channels of process \textit{DEP}. This is shown in Figure 7.13. Therefore,

![Diagram](image)

\textbf{Figure 7.13: Accurate definition of the FLIGHTS component.}

in order to obtain the arrivals or departures information, the appropriate guard method needs first to be invoked before the \texttt{*ARR.allArvls} or \texttt{*DEP.allDprts} method.

In the figure the second limitation is also shown as is explained below. This limitation stems from our implementation, namely that currently active computation processes can only perform i/o of single integer values and not of an array of integers. Consequently, the \textit{CTRL}, \textit{ARR} and \textit{DEP} active computation processes must encode the arrays containing the planes and times information into a single integer which are then decoded by the receiving party. This is the role of the \texttt{enc} function in the figure. The encoding/decoding functions are defined at the source code level. The fact that process channels can only provide a single integer output, apparently affects the interface of the \textit{FLIGHTS} component as shown in the figure as explained below: the single returned integer encodes the arrival and departure information for both planes.

The updated definition of the \textit{FLIGHTS} component, i.e. the one taking into account the limitations of JCSP is shown in Figure C.1. The interested reader is referred to Appendix C for a more detailed discussion, as well as for some comments regarding the formal modelling of active atomic components.

After the active atomic component has been model-checked, the component developer needs to specify in the builder tool the constituent processes and how
these connect with each other. For simplicity, we only use two planes. A wizard facilitates this procedure. The first step is simply to define the component’s name. In the second step of the wizard, shown in Figure 7.14, the processes of the active atomic components are defined by declaring the channels that they consist of. In the bottom of the figure the two processes representing the planes have been already defined. In the top of the figure, where the channels of the \textit{CTRL} process are defined, the channel types currently supported by our implementation are shown. Channel types \texttt{CHANNEL\_INPUT\_INT} and \texttt{CHANNEL\_OUTPUT\_INT} describe integer i/o. Channel \texttt{ALTING\_CHANNEL\_INPUT\_INT} describes an integer input channel, where the receiving process may alternate. This type is used for channels \texttt{arr}_i and \texttt{dep}_i, \(i \in 0, 1\), and denotes that while process \textit{CTRL} is waiting for input on these channels, the first one received will trigger subsequent actions of the \textit{CTRL}. There are also the input channels \texttt{addArr} and \texttt{addDep} of processes \textit{ARR} and \textit{DEP} respectively. Channels \texttt{BARRIER} and \texttt{ALTING\_BARRIER} provided no i/o. The \textit{guard} channels of the \textit{ARR} and \textit{DEP} processes are of type \texttt{ALTING\_BARRIER}. This choice of channel types is affected by JCSP [2] and this is
discussed in more detail in Section 7.2.1.

After the channels of all processes have been defined, the next step is to define how these processes connect with each other. This is shown in Figure 7.15. In the figure it can be seen that the component developer first selects the channels that should be linked, which automatically populates the Selection text box. Then the type of the link for the selection is defined. In the figure we have used \texttt{ONE\_2\_ONE\_CHANNEL\_INT} which, as its name suggests, is an integer channel connecting exactly two processes. For the \texttt{FLIGHTS} component all links used are of this type. The Current Links text box defines all the links that have been formed so far. Other possible link types are \texttt{DELTA\_CHANNEL\_INT}, \texttt{BARRIER\_HUB} and \texttt{ALTING\_BARRIER\_HUB}. The first connects an output integer channel to multiple input integer channels, where all of the input channels must synchronise for consuming the value. The barrier links are applicable for multi-party synchronisation where no values are communicated and are related to the barrier channel types mentioned earlier.

Next, the XML definition of the component is automatically created and is deposited in the repository. Part of it is shown in Figure 7.16. The interface of the component is automatically generated and consists of the four methods *\texttt{ARR.allArvls}, *\texttt{DEP.allDprts}, *\texttt{ARR.guard} and *\texttt{DEP.guard}, corresponding to \texttt{allArvls}, \texttt{allDprts} and the \texttt{guard} channels of the connected processes. As
explained earlier, the output parameter of \texttt{*ARR.allArvls} and \texttt{*DEP.allDprts} is a single, encoded, integer value, representing a sequence of integers. Other than the interface, the XML definition contains all the information captured by the wizard, namely the processes of the component, their channels and how they connect with each other.

Via the procedure presented above, we created the XML definitions for active and passive components that are stored in the component repository. During this procedure, the interfaces of composite components and of active atomic components are defined, based on information provided by the component developer. It should be noted that the definition of each component complies with the Z formalisation that we provided earlier. For example, the interface of the composite \texttt{FngrPrtChecker} shown in Figure 7.12 complies with the formal definition provided in Figure 4.20. This means that the interface of the composite can be ‘no more’ and ‘no less’ than the allowed interface of the Z definition. The Z definitions have actually guided us in providing this implementation.

The builder only guides us in building the XML definition for active and
passive components. It does not provide a programming environment for actually
developing the computation units and the active computation processes. These
must be manually developed by the component developer and are then packaged
into a jar file. When the component assembler is used, Section 7.4, both the XML
definition and the jar files are needed. In the current implementation, these two
developed independently, which constitutes a limitation. This is discussed in
more detail in Section 7.6. In the following two sub-sections we provide a brief
overview of the implementation of computation units and of active computation
processes.

7.3.1 Developing computation units

Developing the source code of a computation unit is straightforward. It is a
plain Java class that contains public methods that have identical signature to the
methods of the atomic component. If necessary, the class can use other auxiliary
classes. In Figure 7.17 we give as an example (part of) the source code for the

```java
package cbsd.examples.compUnits;

public class PwdChecker {
    ...
    public void insertPwd(String usrName, String pass) { ... }
    public Boolean checkPwd(String usrName, String pass) { ... }
    ...
}
```

Figure 7.17: Source code for the PwdChk computation unit.

password checker computation unit.

After defining the computation unit class, the next step is to define the syn-
chronisation constraints (if any) and then pack them, along with all auxiliary
classes into a jar file. The manifest of the file contains the necessary informa-
tion for locating the class of the computation unit as well as the synchronisation
constraints. For the case of the password checker component, synchronisation
constraints are needed, see also Figure 4.15. This is reflected in the following
entries in the manifest file of pwdChk.jar:

```
Comp-Unit: cbsd.examples.compUnits.PwdChecker
Sync-Constraints: PwdCheckerConstraints.txt
```
The above entries are required for any computation unit. If no synchronisation constraints are needed, the relevant entry must be set to null.

### 7.3.2 Developing active computation processes

Similar to the computation unit, the active computation processes are defined separately and are then packed into a jar file. The only entry needed for the jar file is to define the active computation processes. In the case of the FLIGHTS active component, the entry would be:

\[
\text{CSPprocesses: } \text{processes.flights.CTRL processes.flights.PLANE processes.flights.BUFF}
\]

A single PLANE class is needed for both planes. This is because the two planes have the same behaviour. Therefore a single class is used to define this behaviour. At run-time, two distinct instances (objects) of this class will represent both planes. Similarly, a single BUFF class is needed for the ARR and DEP processes.

The implementation of each active computation process is done using the rules in JCSP and their complete presentation is out of scope of this thesis. The rules can be found in [2]. When developing active computation processes we have additionally added some conventions for enabling automatic instantiation by the assembler tool. We briefly discuss the development of active computation processes via the example of the PLANE process shown in Figure 7.18. As can be seen in the figure, we have declared a PLANE process that will be used for implementing processes PL0 and PL1 in the FLIGHTS component. As expected, the process declares integer output channels for reporting plane departure and arrival times. In order to automate the construction of active atomic components, we have adopted the convention that the declared channels are the only input parameters of the sole public constructor of the active computation process. This convention and the limitations it poses is discussed in more detail in Section 7.6. Finally, method run is used for defining in actual behaviour of the process. For the case of the PLANE process, the run method is implemented so that it outputs at random intervals the departure or the arrival time for the plane.

### 7.4 Component assembly and execution

In this Section we discuss component assembly and execution. Since component assembly occurs for deployed components, we first present how our tool supports
7.4. COMPONENT ASSEMBLY AND EXECUTION

```java
package processes.flights;

public class PLANE implements jcs.p.lang.CSPProcess {
    /* Channel declarations */
    private final ChannelOutputInt arr;
    private final ChannelOutputInt dep;
    /* Single constructor whose only input params are channels ordered identically to their declarations */
    public PLANE(ChannelOutputInt arr, ChannelOutputInt dep) {
        this.arr = arr; this.dep = dep;
    }
    /* Actual process behaviour - executes on its own thread */
    public void run() {
        while (true) {
            ...
        }
    }
    ...
}
```

Figure 7.18: Source code for the Plane active computation process.

deployment of active and passive components with the help of two representative examples. The examples that we use are the AUTH and FLIGHTS that were built previously in Section 7.3. Then, the deployed components are assembled using the facade connector and the system starts executing as was shown in Figure 5.8. These examples however only provide an overview of the tool. Deployment phase composition is supported by a Java implementation of the model that reflects the formal type system as was discussed in Section 7.1.

In the assembler, components from the repository are deployed. Deployment means that we provide the components with all the necessary resources needed for their execution. Currently, we only consider data as a resource. For computation units the data needed are the ones reflected by their constructors. For active computation processes we consider that they require no data. As was explained earlier, this is because we use the convention that they only have a single constructor whose input parameters are channels. Composite components are recursively parsed and the data they require, are those of the constituent atomic components.

In Figure 7.19 the deployment wizard for the AUTH component is shown. Since AUTH is a composite component, the wizard recursively locates the atomic components that it consists of and the appropriate wizard panel is shown. In the figure, the wizard has located sub-component AC3 which corresponds to the
Figure 7.19: Deployment wizard for the 
AUTH component.

CardReader component, see also Figure 4.19. The component developer specifies the jar file that contains the computation unit and selects the constructor that will be used for creating an instance of it at run-time. The constructor takes as input a string which identifies a file to be used for read card data from.

After all computation units have been deployed, AUTH component shown in Figure 4.30(a) is deployed and it appears in the assembler tool as shown in Figure 7.20. AUTH component is now executable. However, we need first to deploy the active atomic component FLIGHTS and then compose the two components using the facade connector.

Similar to the deployment of passive atomic components, when deploying an active atomic component the user needs to specify the jar file containing the active computation processes. The next step, shown in Figure 7.21, is to map the processes of the designed component to their actual implementations in the jar file. In the figure, there are five active computation processes in the designed component, PL0, PL1, CTRL, ARR and DEP, that are mapped to three implemented processes BUFF, PLANE and CTRL. As was explained earlier, the reason for having fewer implemented processes is that the same implementation can be used for different processes. For example, the BUFF implementation represents a single-slot buffer that can clearly be used for both ARR and DEP active computation processes.

In Figure 7.21, process ARR is mapped with BUFF implementation. In the
7.4. COMPONENT ASSEMBLY AND EXECUTION

Figure 7.20: Deployed AUTH component.

Figure 7.21: Deploying FLIGHTS component.
lower part of the wizard figure, the user maps the channels of the two processes. After all processes have been mapped, the active atomic component AUTH is placed in the assembler window, as shown in Figure 7.22. Both components are now ready to execute. We will show how to run the system after composing them with the stateful facade.

We compose the AUTH and FLIGHTS components with the stateful facade as shown in Figure 7.23. A general observation that can be made about the figure, is that whenever a composite component is formed in the assembler, the sub-components remain visible. This is in contrast to the composition using the builder, where the sub-components are well encapsulated within the composite. This is because the builder and assembler are used to realise the design and deployment phase compositions respectively, and as explained in Section 3.3, in the deployment phase, using the same sub-components in multiple compositions is allowed.

Specific to the assembly of composite CC, the developer needs to provide a DataImporter, a Checker and an Updater object, see also Section 7.2.2. Doing so via a GUI would require an overly complicated implementation. Instead, we currently define the three classes inside an external jar file. The role of the assembler is to locate this file and create appropriate instances of these classes, which are then passed to the facade’s constructor.
Figure 7.23: Executing the authenticated flight monitoring system.

After the stateful facade and the composite CC have been constructed, the composite can be executed as shown Figure 7.23. A sample execution is shown in the same figure, where we consider two users executing the composite CC in parallel via the windows “User 0” and “User 1”. This the visualisation of the top level loop connector that was shown in Figure 5.8. We posit that the card number of the first user is 100, and that of the second user is 200. We have externally enumerated the execution trace for ease of explanation. First, user 0 was successfully authenticated. Then, all channels of the FLIGHTS component were invoked as shown in the figure. The results of the invocation can be seen in both the execution trace and in the user window. In the latter, the results are 0 and 2855735. As was explained in Section 7.3, these two numbers represented the encoded information of the arrival and departure times of the two planes. The decoded information is shown as number 2 of the execution trace. The results mean that neither plane arrives, but both of them depart at the specified times. Next, as shown in user 0’s window, the user intends to retrieve the arrival and departure information again. All four channels need to be executed and since each one requires the user card number at the CC level, the number 100 is entered four
times as an input parameter.

Part 3 of the execution trace shows that user 1 with card number 200 tried to execute the AUTH component, but failed because they had not authenticated. Next, in user 1’s window it is shown that the user tries to authenticate using their card number, password and fingerprint.

This discussion concludes our presentation on the implementation of component construction and execution. Next we discuss the implementation of composite connectors.

### 7.5 Composite connectors

Having defined a syntax for composite connector template definition, we can use this syntax for writing down an equation that describes a composite connector. Furthermore, we want to be able to conveniently store the composite connector template definition in the repository and automatically process the definition to construct the connector instance for composing components into composite ones.

To that end, we use XML to capture the connector template definition. The reasons for using XML are, first that XML is structured and hierarchical, which naturally supports our structural and hierarchical composition. Second there are existing efficient parsers such as JDOM, that ease the process of syntax parsing and can conveniently integrate as the rest of our implementation which is in Java. Consequently, a composite connector template definition is an XML file which is stored in the connector repository. For simplicity of uniqueness validation and lookup, a connector template definition file has the same name with the connector template name.

The grammar of such a connector template XML file is shown in Figure 7.24. In the figure, it can be seen that there is a root element called `connectorTemplate`. The root then consists of a declaration element identified by the `declarations` tag and a template definition element identified by the `definition` tag. The declaration element in turn consists of a number of declarations for fixed arity connector templates using the `fat` tag. The definition element must include a single element fixed arity connector which in turn consists of a number of composition places, parametric arity templates and (possibly) other fixed arity template elements identified by `cp`, `pat` and `fat` tags respectively.

As an example, in Figure 7.25 the composite connector template for Observer...
7.5. COMPOSITE CONNECTORS

is shown. It is composed using a binary pipe as an operator and an n-ary cobegin as the operand. More precisely, by declaring the arity of the PIPE connector as binary, a fixed arity type named pipe2 is defined. Thus the template definition named Observer is being constructed by using a pipe2 connector template as the operand to compose a composition place with the CB connector template as the second operand. The role of the composition place is to define an empty placeholder for the publisher.

Our tool allows a user to define a composite connector as shown in Figure 7.26. In the figure, the XML description is manually defined by the user. The ‘compact’

Figure 7.24: XML grammar for connector composition.

Figure 7.25: XML definition of the Observer composite connector.
and ‘detailed’ views of the Observer connector are shown in Figure 7.27. Using these views, the XML definition is parsed and validated by our implementation. The connector can then be stored in the connector repository as shown in the first figure. The composite connector can be used by the builder or assembler tools for component composition. When this happens, the user needs to define the arity of the cobegin connector, and then, composition proceeds bottom-up. The composite connector can also be used in another connector composition by following the procedure described in this section. Because it is stored into the connector repository, our tool can locate it and parse it into subsequent compositions.

### 7.6 Chapter summary

In this chapter we have presented the implementation of our model. We saw that the design and deployment phase implementations respectively define the design and deployment phase syntax of our model. During this presentation we focused on the most complicated parts of our implementation, namely active atomic and composite components. By means of a case study, the authenticated flights monitoring system, we demonstrated the design and deployment phase compositions, and also run-time execution.

Through the case study a limitation of our implementation was highlighted,
which is related to the idealised component life cycle, Section 2.2.1. The limitation is that the designed components are XML files that only describe the structure of the component. Ideally, a designed component should also contain its source or binary code (probably in more than one languages), instead of that being packed into a separate file.

This chapter effectively concludes the presentation of our component model. In the next chapter we present an experiment that we conducted with the purpose of evaluating the expressiveness of our composition operators.
Chapter 8

Expressiveness of composition

Having presented our model and its implementation in detail, we need to evaluate whether our model is sufficiently expressively for the modelling and construction of a wide range of component-based systems. A simplified version of our model, i.e. our model without support for concurrency, and without composite composition operators, has been used for the development and verification of an experimental missile guidance system, with about 30,000 lines of code [85].

The above case study gives us confidence about the expressiveness of our model. However, concurrency and connector composition, are out of its scope. In this chapter we conduct an experiment in order to evaluate the expressiveness of our composition operators in terms of the control flow that they can define. This experiment includes concurrency and composite connectors.

We choose to compare the control that is expressed by our connectors against the control that can be expressed by Workflow Control-flow Patterns (WCPs). WCPs [14, 114] are patterns of control for workflow systems [129]. Two reasons dictate our choice to WCPs. The first reason is that our connectors represent patterns of control for component based systems and in fact there is a very close relationship between the role of WCPs for workflow systems and the role of our connectors for component-based systems. In other words, it is semantically meaningful to compare WCPs to our connectors. This claim is further explained in Section 8.1. But a comparison cannot be practically useful if WCPs describe trivial control flow. This forms the second reason for choosing WCPs. WCPs describe a rich set of complicated forms of control for workflow systems. This is witnessed in reviews performed on the support of WCPs in current workflow managements systems and in business modelling languages [14, 114, 34, 75]. In fact,
8.1 Workflow Control-flow Patterns

Workflow control-flow patterns (WCPs) are patterns of control for workflow systems. In this section we introduce WCPs and demonstrate their relevance to our connectors and their composition.

Patterns originate from the work of Alexander [17] on architectural patterns. This work has since been inspiring to software engineers, particularly during the last two decades [61, 42, 112]. For a pattern definition we adopt the one given in [112]: “A pattern is the abstraction from a concrete form which keeps recurring in specific nonarbitrary contexts”.

Aalst et al introduced in [14] the first twenty workflow control-flow patterns, which were later extended to forty three [114]. WCPs are patterns of control for workflow systems. Therefore WCPs describe recurring forms of control in the context of workflows. A workflow is “the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules”, where a business process is “set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships” [129]. Therefore a WCP describes a recurring form of control among a set of activities that constitute a business process. These activities are logically related to each other, in terms of their contribution towards the realisation of the business
objective.

In the following, we show that there is a correspondence between the role of WCPs for workflows, and the role of our connectors for component-based software systems. Consequently, we consider it meaningful to compare the control expressed by WCPs against the control expressed by our connectors.

8.1.1 WCPs are connectors

We mentioned earlier that the forty three patterns in [114] describe complicated forms of control for a set of workflow activities that constitute a business process. Workflow activities need to be ‘invoked’ by the workflow in order to execute and an activity cannot directly trigger another activity – this has to be done through the workflow. Similarly, in our model, connectors describe patterns of control among software components in a system. Components are invoked by our connectors and components do not directly communicate with each other. At this abstract level, WCPs serve the same purpose for business process activities as the purpose that our connectors serve for component-based software components. Consequently, it is meaningful to say that WCPs are connectors, with the difference that WCPs are applied to workflow activities instead of software components. Furthermore, it is also meaningful to analyse how specific WCPs are matched to specific connectors.

At a more concrete level, we use the formal Coloured Petri net (CP-net) semantics that were introduced in [114] for defining the exact meaning of WCPs. More precisely, we use the CP-net formalisation of the WCPs to analyse and match them to our connectors. In the CP-nets provided in [114], transitions in general represent the activities, or tasks, of the workflow. Places represent the state that the workflow is in before and after firing a transition, and they therefore indicate which activities may fire next. In addition to these, there are auxiliary places and transitions that assist in realising the workflow and which are different to the aforementioned ones. For instance there are places that represent the input and output of a workflow. Therefore in the CP-net of a workflow, the control-flow is represented with the flow of tokens from the input to the output places of the pattern, through the special transitions that represent activities. For example, in the sequence pattern, two or more activities execute sequentially. As in our work, so in [114], tokens represent control.

In the CT-net formalisation of our work, we use a set of composition places
to represent an abstraction of the components/connectors that our connectors compose, as was described in Section 6.2. Each connector has a single input and a single output place. A set of transitions and places describe the control flow of the connector, in terms of the flow of the tokens from the input place to the output place through some of the connected components. Since our tokens represent the control in the system, their flow defines the control-flow of the connector.

When we map WCPs into our connectors, we match the control flow among workflow activities to the control flow among components. Some of the patterns described in [114] are matched to our basic connectors. However, most of them describe complicated patterns of control, which we can only represent via composite connectors. Some WCPs cannot be modelled using our connectors. This is either because they do not make sense in the context of our model or because our model is not expressive enough to model them. Before presenting a summary of the mapping process, we present some context assumptions that are associated with WCPs.

8.2 Context assumptions

CT-nets in [114] are associated with specific context assumptions. These assumptions are used for unambiguously defining the meaning of WCPs. One such assumption is that all CT-nets are safe unless explicitly said that this is not the case. A safe Petri net, is a net where each place in the network, at any marking, can contain at most one token. This means that each activity in a workflow cannot execute concurrently with itself. In our component model, we do not have this restriction, since in general, every component/connector supports concurrency in the form of concurrent processes as was described Section 3.5. Additionally, whenever we require that a connector is executed by a single processes, it is straightforward to enforce this, by defining a mutual exclusion (mutex) synchronisation constraint for the connector. The mutex synchronisation constraint for atomic components has been already presented in Figure 4.9. It essentially means that although many tokens can be in the input place of a component, they are forced to access it sequentially.

This notion can be applied to any of our generic composition connectors by

---

1The CT-nets for all our basic connectors are given in [80].
converting their control flow, presented in Figure 6.5, into the one shown in Figure 8.1. The fusion place MUTEX is initially marked with a single token, the natural number 0. In order for transition $T_{in}$ to fire, there must be single token in the MUTEX place and at least one token in place In. After $T_{in}$ fires for the first time, it cannot fire again until $T_{out}$ places the token 0 in the MUTEX place. In other words, there can never be two processes executing a single connector concurrently. For symbolising the MUTEX synchronisation constraint to a connector, we apply an line above the connector’s name, such as $\overline{CONN}$.

The above discussion highlights a fundamental difference between workflows and our model regarding concurrency control. The difference is that in a workflow, the decision whether an activity can be executed concurrently or not, is made not by the activity itself, but instead is made by the workflow. In contrast, in our model this decision is made by a component/connector itself.

The assumption of using safe CT-nets, has forced the authors in [114] to usually define two WCPs for what is actually a single control pattern: one WCP is defined where the CT-net is safe and one where it is not. In our model, we offer the same connector in both cases. In the first case, we assume that the connected connectors/components are defined according to the mutual exclusion synchronisation constraint. Nevertheless, whenever this is the case we mention it explicitly.

\section*{8.3 WCPs that can be mapped}

In this section we start by presenting a summary of the patterns that can be mapped into our connectors. Representative examples of the mapping process
are also presented. A detailed analysis of the mapping process for all WCPs is given in [80].

Out of a total of forty three patterns, twenty four patterns describe control flow that can be expressed in our model. Some of the patterns are mapped “in isolation” to a connector in our model. This means that a single pattern has been mapped directly to a connector. These patterns are shown under the relative column in Table 8.1. However, most of the patterns have been mapped “in pairs” to a connector in our model. This means that we have combined two patterns into a larger one, which is then mapped to one of our primitive or composite connectors. The ‘Implicit’ column denotes that the control described in the relevant pattern (WCP-11) is implicit in our component model and no connector is needed for that purpose. In the ‘Partial’ column we include patterns such that either our model only offers an approximation for the pattern (WCP-21, 39), or it requires an excessively complicated solution from us, to express the same control flow with the pattern (WCP-17, 40). The implications of this table and whether implicit, partial and in-pairs mapping constitute a significant disadvantage of our approach in the field of CBSD, is discussed in Section 9.4.

It is worth noting that during the mapping process, we have used both primitive and composite connectors. We next provide more details of these categories, by discussing a representative example from each one. For simplicity, we only present examples where WCPs are mapped to basic connectors. An exhaustive presentation can be found in [80].

As an example of a pattern that maps directly to one of our connectors, we use the sequence pattern (WCP-1), which evidently maps to the sequencer connector. This is shown in Figure 8.2. Because WCP-1 has the context assumption of a safe Petri net, this is achieved in our model by using the mutual exclusion synchronisation constraint. We use this figure to present some conventions that we follow for relating the CP-nets WCPs to the CT-nets of our connectors. Specifically, we name the transitions in the WCPs that present workflow activities as $A_i, i \geq 1$. These activities map to our components, which are the composition

<table>
<thead>
<tr>
<th>WCPs</th>
<th>In isolation</th>
<th>In pairs</th>
<th>Implicit</th>
<th>Partial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1, 13, 18</td>
<td>2, 3, 4, 5, 6, 7, 8, 9, 12</td>
<td>11</td>
<td>17, 21, 39, 40</td>
</tr>
<tr>
<td></td>
<td>16, 28, 39, 31, 33, 37, 41</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
places $CP_i, i \geq 1$. Naming these composition places as $CP_i$, instead of $CP$ as used so far, is done to emphasise the correspondence between WCPs and our connectors. Finally, input and output places are marked as $In$ and $Out$, and auxiliary transitions and places always have different names.

The second category of the mapping scheme describes the case when a pair of patterns is mapped to a basic or composite connector. This occurs in order to describe symmetric control flow and is illustrated in Figure 8.3. WCP-2 is the parallel split pattern and WCP-3 is the synchronisation pattern. In the parallel split, a single branch is split into two or more parallel branches, which execute concurrently. In the synchronisation pattern, two or more branches converge into a single branch, which only executes when the preceding ones have finished. Since we map the two patterns into the cobegin connector, and the second pattern has the context requirement that it is safe, the cobegin must conform to the mutual exclusion synchronisation constraint. These patterns individually cannot give rise to a connector because they do not describe symmetric control flow. For instance, if we had a single connector to represent the parallel split pattern, then the resulting connector would either have two (or more) output places, or a single output place, but for each incoming token it would return two. In either case, Property 5 that describes the semantical self-similarity of connectors would be broken and such a connector could not have been used in composition.

Next, there is a pattern that cannot be explicitly expressed by a connector in our model, because it does not describe explicit control-flow. This is the
8.3. WCPS THAT CAN BE MAPPED

Figure 8.3: WCP-2&3/ CB.

implicit termination pattern (WCP-11) and describes that a process instance should terminate when there are no remaining (workflow) tasks to be executed either now or at any time in the future. In other words, it should be possible to infer when a process instance has terminated its execution. This pattern is supported implicitly in our model as is explained below. According to the component life cycle, when an assembled system is transferred into run-time there are two sources of executing processes, either active atomic components, Section 4.3, or one of the top-level connectors, Section 5.2.1. The purpose of active atomic components is to never terminate, as is the case for the infinite loop. Therefore, in order to model the implicit termination pattern, we only need to identify when termination occurs in the case of the top-level loops. These are the concurrent loop and the pre- and post-check loops. In the case of the concurrent loop, after a complete iteration, the user decides if the same process should be used again, and either the process is used for another iteration or it terminates. In the case of pre- and post-check loops, as soon that the loop’s exit condition is met, the loop and the process executing it terminate. In either case, it is clear when a process has finished its execution. Consequently, in our component model implicit termination is supported by definition of the model’s run-time semantics.
Finally, the fourth category in Table 8.1 contains patterns that we can either only offer an approximation of their behaviour to, or they require an excessively complicated solution from us in order to express the same control flow. For example, partial support is provided for the structured loop pattern (WCP-21). The pattern defines that an activity, or a set of activities, may be executed repeatedly inside a loop. The loop terminates based on a condition that is evaluated either at the beginning, or at the end of the loop. In either case the looping structure has a single entry and exit point. This pattern can be mapped into the pre- and post-check loops which were discussed in Section 5.2.1 depending on whether the loop’s exit condition is evaluated at the beginning or at the end of the loop respectively. However we classify our support for this pattern as partial, because our loop connectors can only be applied as top-level connectors in the system due to termination issues.

An example of a pattern that requires an excessively complication implementation from us is WCP-40, the interleaved routing pattern. WCP-40 describes that there is a set of activities where each must be executed exactly once. The order in which the activities execute is arbitrary. However no two activities may execute in parallel. In other words, from a set of activities any permutation is acceptable. In our model, given a set of components and our basic connectors, either the components in the set execute in sequence or in parallel, when a sequencer or a cobegin connector is used. In order to describe the equivalent control behaviour we need to explicitly enumerate all possible permutations of components in the set. Then, for each permutation a sequencer is used and all sequencers are composed via a non-deterministic selector. The case for two components is shown in Figure 8.4. Evidently, for three components, six sequencers are needed, for four, twenty four, and so on. Therefore, although it is possible to describe identical control flow to WCP-40 for any given set of components, the way this is achieved, is overly complicated.

![Figure 8.4: Composite connector wcp40-2.](image-url)
8.4 WCPS that cannot be mapped

Table 8.2 summarises the workflow control-flow patterns that cannot be expressed in our model. There are three basic reasons explaining why we cannot model one of these patterns in our model. The first reason is related to the inability of our model to support a component with an interface whose length of input/output parameters can only be defined at run-time. This is the category under the heading “Interface generation” in the table. For example, WCP-14 is the multiple instances pattern with \textit{a priori} run-time knowledge. This pattern allows the creation of multiple instances of an activity that can execute in parallel. This pattern could be mapped to our finite parallel loop adaptor shown in Figure 5.3, but in WCP-14 the number of concurrent instances is determined when the activity is invoked. This is accomplished by having the instances number as an input parameter to the pattern. Therefore, the length of input/output data cannot be known until the activity is invoked at run-time, and consequently that component cannot have a fixed interface. The rest of the patterns in this category are all variations of WCP-14. Building a connector with such a property, i.e. a connector that defines a component whose interface is undefined until run-time, would violate the semantics of our component model, because that component would be impossible to use in further compositions. However, this behaviour might be acceptable when the connector is used as a top-level connector, i.e. when this connector is used, a system is defined, no further composition is possible, and the run-time phase follows. These patterns could therefore be potentially mapped in our model, but this is still an open issue.

The second reason regarding why we cannot model a pattern using our component model, has to do with the lack of cancellation semantics for component execution. The patterns in the relevant column describes scenarios where an activity, or a set of activities, have their execution cancelled. For example, WCP-19 is the cancel activity pattern and describes that an activity can be cancelled while executing. Our model does not currently support the necessary semantics

<table>
<thead>
<tr>
<th>WCPs</th>
<th>Interface generation</th>
<th>Cancellation Semantics</th>
<th>Semantical Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>14, 15, 34, 35, 36</td>
<td>19, 20, 25, 26, 27, 29</td>
<td>10, 22, 23, 24, 38, 43</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>32, 35</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.2: Summary of WCPS that cannot be expressed in our model.
for that, but this is part of future research, see also Section 10.2.3.

The first two reasons describe difficulties that may be overcome with a subsequent version of our component model. However, the third reason states that some patterns cannot be expressed in our model because doing so would violate the semantics of our model in a way that cannot be remedied with a subsequent version of our component model, as was the case with the first two reasons. In other words, applying these patterns would lead to violation of fundamental properties of our component model, like encapsulation of control flow and self-similarity. For example, WCP-10 is the arbitrary cycles pattern. It describes the case where there can be arbitrary cycles in the model, i.e. workflow activities may be executed an arbitrary number of times and an activity is allowed to be followed by any other activity in the process. Applying this pattern into our model would break one of its fundamental properties, namely that connectors encapsulate control. In other words, arbitrary cycles in our model would mean that the control flow can be arbitrary, which violates our semantics. Evaluation of the fact that it is not possible to model certain WCPs in our model is discussed in Section 9.4.

8.5 Analysis

Based on the tables 8.1 and 8.2 some observations can be made. The first observation is that in the current version of our model, we can express approximately half of the WCPs. However, this is not a big disadvantage per se, because as was previously explained, only a small proportion of WCPs are such that they can never be expressed in our model, i.e. even in subsequent versions of our model. These are the six patterns that if applied, they would lead to the violation of our semantics.

A larger concern regarding the expressiveness of our component model, is that we have mostly mapped pairs of patterns into connectors. As has also been explained earlier, this occurs because our connectors describe symmetric control patterns, with a single input and output place, and each control process in the input place is eventually forwarded to the output place. In essence, this shows that WCPs express more fine-grained control flow and our connectors cannot capture this elementary control flow. For instance, we do not have a connector for expressing the split of a process into two, or the joining of a number of processes
into one, as is captured by WCPs 2 and 3 respectively. Instead, we can only model them by combining them in a specific way, as we did for instance with the cobegin connector. Arbitrary combinations of these primitive control-flow patterns may lead to perfectly valid forms of control, however modelling this control in our model would be impossible, because it would violate our fundamental principle of semantical self-similarity described as Property 5 in Section 6.2.

Another remark regarding expressiveness is that our connectors express more generic forms of control. This holds because our connectors are parameterisable. The fact that our connectors are more general does not contradict the fact that we cannot express control-flow that can be described in WCPs. Our connectors are generic regarding certain forms of control flow and on the contrary they do not express elementary forms of control flow in isolation. For example, the efficient cobegin can be used for WCPs 9, 28, 30 and 31 by varying either the number of joined threads, or the application of the mutex synchronisation constraint, or both (see [80]). On the other hand, we do not actually model these patterns in isolation, but only when we explicitly pair them with WCP-2.

The fact that our connectors are generic it means that they can be specialised into different control forms, as the previous example with the efficient cobegin demonstrated. An immediate effect that is ubiquitous to all our connectors, as well as to our components, is that by definition we do not need to use safe Petri nets since our connectors have the necessary semantics for concurrent execution. Therefore and contrary to WCPs, when we want to restrict their execution to a single process, we apply the mutex synchronisation constraint. The generic nature of our connectors is also evidenced by their structure being more complicated for expressing the same control flow compared with their WCPs counterparts. The fact that our connectors are generic and parameterisable, is certainly an advantage of our approach. This gives us the opportunity for greater reuse, not only at the modelling level, but also at the source code level, by using the same connector implementation in different systems.

Regarding composition, we have previously shown that it is possible to combine patterns to form a larger pattern. For example the parallel split and the synchronisation patterns can be combined to form the cobegin connector. We claim that this is combination of patterns rather than composition. It is an ad-hoc process and it involves merging places (and/or transitions) into a single place (and/or transition). Additionally, auxiliary places and transitions may be
required. This is because their patterns do not have a strong notion of self-similarity. Although each pattern has both input and output places, their arity is not fixed. Also, for each incoming thread more than one thread may be output on the same output place. This accounts for a very loose notion of self-similarity and therefore it is not possible to automatically compose any two patterns into composites.

An additional reason for WCPs to not be able to be automatically composed, is because they depend on context assumptions. With the exceptions of patterns 10, 11 and 39, the other patterns are associated with some context assumptions for their precise definition. For example, the context assumptions for some WCPs dictate the use of safe Petri nets, while for others this is not the case. As a result, it is not possible to combine arbitrarily a safe with a non-safe Petri net. The system developer must ensure, usually through additional transitions/places, that the more restrictive context assumptions are satisfied. For the case of safe Petri nets, the composer must enforce that in the resulting, ‘composite’ CP-net, the safe sub-net remains safe.

The above discussion can be summarised in the following two points. First, WCPs express elementary forms of control that it is not possible to express using our connectors. Our connectors express more coarse grained control and despite this fact, they are more generic than WCPs. This is because our connectors are parameterisable. The second conclusion is that “composition” in WCPs is actually ad hoc combination. This is in contrast to our approach where the composition of connectors is algebraic and follows pre-defined rules. Equally important is the fact that it is not the case that any two patterns may be combined. Our model on the other hand allows any two connectors to be composed using a composition operator, since we have defined algebraic semantics where the connectors are both operators and operands. An evaluation of these points taking into account the broader context of CBSD, where we place this work, is provided in Section 9.4.

8.6 Chapter summary

In this chapter we conducted an experiment to evaluate the expressiveness of the composition and adaptation connectors of our component model. To that end, we introduced Workflow Control-Flow patterns (WCPs). WCPs describe a rich
set of complicated control structures for workflow systems. Given that there are certain analogies in the role of WCPs for workflow systems, and in the role of our connectors for component-based systems, we considered it meaningful to compare WCPs to our connectors.

The result of this experiment was that the control flow in approximately half WCPs can be expressed by either basic or composite connectors of our model. For the other patterns we explained that most of them can be potentially expressed by appropriately enriching our component model. Only a small proportion cannot be expressed by our connectors because doing so would lead to the violation of fundamental properties of our component model, namely encapsulation of control in connectors and self-similarity. During our analysis we highlighted, first, the differences in the granularity level of control described in the two approaches, and second, we highlighted the differences in composition.

In the next chapter we provide a holistic evaluation of our model. In that discussion, the results of the current experiment are taken into consideration and are evaluated in the light of wider CBSD desiderata.
In this thesis we presented a software component model with concurrency. In the
current chapter we evaluate our work against related component models that were
presented in Section 2.3. This evaluation is performed with specific regard to three
issues: software reuse, support for concurrency and expressiveness of composition.
As was discussed in Chapters 1 and 2, reuse is of paramount importance in CBSE
and it constitutes one of its main targets. In fact, component reuse has motivated
and significantly affected the definition of the idealised component life cycle that
was presented in Section 2.2.1. Therefore, when we compare component models
in terms of reuse, we use the idealised component life cycle as the basic criterion.
But software reuse is also achieved via connector reuse. These two directions of
comparison are presented in Section 9.1.

Concurrency is an important aspect of software systems. We argued in Sec-
tion 2.2 that in the context of software component models, both active and passive
components should be modelled. It is also desirable that this distinction is made
explicit at the modelling level and not at the implementation level. In general, all
concurrency should be defined and handled at the modelling level, where formal
checks are possible. Support for concurrency in our model w.r.t. related work is
examined in detail in Section 9.2.

It is worth mentioning that our arguments in the evaluation of software reuse
and concurrency are based on qualitative criteria. We introduce some quantitative
criteria when evaluating the expressiveness of our approach to building software
systems. Since control flow in our model is defined by connectors, we try to
answer the question of whether our connectors are sufficiently rich for describing
a wide range of systems. We recognise that a definite answer to that question
can only be provided after the complete formalisation of our work. However, we
have drawn some interesting conclusions by performing a thorough comparison
against Workflow Control-flow Patterns (WCPs), that were meticulously analysed
in Chapter 8. The results of this analysis are discussed in Section 9.3.

Next, in Section 9.4 we provide a holistic evaluation of our model in the light
of software reuse, support for concurrency and expressiveness of composition. We
conclude in Section 9.5 with the chapter summary.

9.1 Software reuse

In Section 2.2 we presented the desiderata for a software component model. For
convenience, we briefly repeat them below. First, components should be pre-
existing software units, that can be reused by being deployed in different ap-
plications by different developers. Second, component development and compo-
nent use should be separate activities and consequently the stakeholders in these
two activities should differ. This helps ensure that components developed are
well defined and whenever possible they are independent from other components.
Third, in order to be used multiple times, components should be capable of being
copied and deployed in different application environments. Fourth, components
should be composable into composite components which in turn can be further
composed into other (composite) components. Composition promotes reuse and
offers a systematic approach to system construction.

The above desiderata are reflected in the idealised component life cycle that
was explained in Section 2.2.1. Recapitulating, the life cycle of components con-
sists of three consecutive stages. In the design phase, component developers
construct the components as reusable entities. In the deployment phase, compo-
nents are deployed into an execution environment where they become ready for
execution. The run-time phase follows, where components start executing. In
order to maximise reuse of components, component composition should occur in
both the design and deployment phases of their life cycle. This is discussed in
Section 9.1.1. In the same section we discuss a limitation of our model regarding
run-time composition and reconfiguration.

As we have shown in Chapter 6, reuse is not only limited to components. Every
time a connector is used in a composition it is actually being reused. More
importantly, connectors in our model can be composed into composite connectors.
This provides reuse both at the design (conceptual) and at the implementation level. Connector reuse is discussed in detail in Section 9.1.2.

9.1.1 Reuse of components

As was explained in Sections 3.3 and 4.5, in our model we support composition of components in both the design and deployment phases of the component life cycle via explicit composition operators. Component composition is facilitated by a repository where components can be stored and retrieved for latter compositions. This is supported by an implementation that allows for design and deployment phase compositions, and also for executing a component-based system. Details of the implementation are given in Chapter 7.

Related component models, as was explained in Section 2.3 and summarised in Figure 2.15, offer a varying degree of support for the idealised component life cycle. Some components models, for instance software architecture description languages, do not even support the notion of a component repository. The categorisation of component models in the figure also revealed that the most advanced component model is SOFA. SOFA offers a repository where components can be deposited, retrieved and used in further compositions. In SOFA, as well as in our model, hierarchical, or composite, components can be defined which are then deposited back into the repository. However in SOFA, although component deployment is defined, no deployment phase composition is possible. Our component model therefore extends the state of the art by being the only component model that supports the idealised component life cycle.

One limitation of our model has to do with run-time composition and re-configuration. Although at run-time “it is hard to define meaningful composition operators other than glue code” [84], run-time composition and re-configuration is a useful asset for a component model, as witnessed by the number of component models that support, in some extent, dynamic reconfiguration: SOFA, Fractal, OSGi, COM, .NET, and EJB. For example, and as was explained in the discussion of the observer connector in Figure 3.8, it is not possible in our model to dynamically vary the number subscribers. In general, once a component is deployed, it is not possible to change any of its internal connections.

The facts that first, we have defined components that cannot initiate calls to other components or connectors, and second that all control and data flow in our model is initiated by connectors, see also Section 3.1, facilitates reuse.
This is because the above characteristics result in loosely coupled components. Actually, components in our model are unaware of each other\(^1\) and can therefore be developed and deployed as independent entities. Apparently, independent components are more re-usable than components with strong dependencies on each other. The latter require that whenever they are used, all their dependencies are satisfied, which in some cases it is not trivial, whereas the former do not face the same obstacle.

Related component models do not share the same degree of independence. The reason is that in related models, components communicate with each other either via direct or indirect message passing. The following discussion summarises the points made in Section 3.1. Direct message passing is the way components communicate in Darwin, SOFA, Fractal, EJB, OSGi, COM and .NET.\(^2\) Indirect message passing is used in C2, Wright, ACME and Javabeans, thus reducing the level of coupling between components. The merits of exogenous communication and control have been recognised by the community of coordination languages and by the community that combines the CSP and B-machines formalisms. Our work is the sole component model that applies exogenous control for component composition and interaction, which results in minimal component coupling.

### 9.1.2 Reuse of connectors

Connectors in our model are composition and adaptation operators that encapsulate the control flow of a system. Every time component composition and adaptation occurs, reuse of connector design, in terms of control flow logic, and of connector implementation, is achieved. More importantly, connectors can be composed with each other and form composite connectors. The fact that composite connectors, along with primitive ones, are stored to a connector repository so that they are used in further connector or component compositions, significantly reduces the engineering effort required in building software systems. Instead of consecutively applying primitive connectors for building a system, pre-built composite connectors can be used instead. The significance of this claim can be recognised when we consider that (composite) connectors are actually patterns, as was discussed in Sections 3.4, 8.1, and in Chapter 6.

---

\(^1\)Obviously we do not refer to the relationship of a composite component with its sub-components.

\(^2\)EJB, OSGi, SOFA, COM and .NET alleviate components’ tight coupling by supporting dynamic loading, connecting (and un-connecting) of components.
As was discussed in Section 6.3, our approach is unique not only w.r.t. to other component models, but also in the broader field of software engineering. This is because in related works, the focus is on the construction of connectors out of primitive elements that are not connectors. In contrast, our approach defines an algebraic approach to connector composition where both the operators and the operands are connectors. Our component model has therefore introduced a new perspective in the design and development of software systems. This perspective is particularly useful and applicable to component-based development as a means of further reuse.

9.2 Concurrency

We have defined a software component model that defines and supports concurrency at the modelling level. Starting from primitive components, we have defined both active and passive atomic components. Active atomic components, see Section 4.3, define active computation processes that execute concurrently and interact with the rest of the system via explicit composition operators. Active computation processes are defined in a process algebra (CSP) and can therefore be model-checked using the appropriate tools (FDR). Passive atomic components, see Section 4.1, support concurrent processes in terms of concurrent invocations. More importantly, we have defined a compositional approach for active and passive components. In our approach both kinds of components can be seamlessly composed into composite ones where all synchronisation issues are transparently handled by our connectors.

Because passive atomic components (and all our components in general) contain data, we have introduced synchronisation constraints as a means for controlling concurrency, Section 4.2. Synchronisation constraints are read and enforced by our connectors. All our composition and adaptation connectors are concurrent because they support the execution of concurrent processes. Finally, we have defined connectors that explicitly branch and join processes, as was briefly explained in Section 3.5.

In summary, concurrency is defined and handled at the modelling level. We believe that this is important because it avoids the pitfalls of defining and handling concurrency at the implementation level. The latter is usually an ad hoc procedure and common concurrency errors are difficult to detect and debug. In our
model, the most complicated concurrent behaviour occurs within active atomic components whose abstract definition can be model checked. In the rest of the model, all concurrency and synchronisation is captured by explicit constructs. In essence, no concurrent behaviour can be introduced unless it is also defined by the model.\(^3\)

Regarding concurrency, the work most closely related to ours are architectural description languages (ADLs), see Section 2.3.1, that use process algebras to capture the behaviour of components and connectors. Wright and Darwin can model concurrent systems by defining the behaviour of their components and connectors in terms of concurrent processes. Wright is formalised in CSP, and Darwin in Finite State Processes (FSP) and $\pi$-calculus, see also Section 2.3.1. However, in these models all components are active entities and passive components are not defined. In Wright where explicit connectors are used, connectors are also active. In the other two ADLs presented in Section 2.3.1, C2 and Acme, all components are active and concurrency is explicitly supported. As we explained in Sections 2.2 and 2.4, it is important that a model defines both active and passive components, because they capture different requirements in terms modelled systems.

Regarding active and passive components, other than our model, only Pecos, see Section 2.3.2, explicitly distinguishes between active and passive components. However, in Pecos no formal checks are possible and synchronisation is performed at the implementation level.

With respect to synchronisation, in our model synchronisation constraints are used that are described in a declarative notation. Synchronisation constraints are more fine grained than the high-level synchronisation facilities, (i.e. those given without using implementation primitives), offered by EJB, COM and .NET. In EJB all components are passive. In COM the apartments model is used where a component can be declared as thread safe or not, and finer grained synchronisation is not defined. In .NET, by defining synchronisation domains, locks can be used for an object or for a set of objects that reside in the IL code of an assembly.

In the remaining component models surveyed in Section 2.3, namely SOFA, Fractal, OSGi and Javabeans concurrency is defined and handled at the implementation level. That is although in the first two, behaviour protocols have been

\(^3\)Our current implementation does not protect against ‘malicious’ implementations of computation units. For example if a computation unit spawned new processes, we would not have been able to detect that. Future implementations will detect and forbid such behaviour.
used as a means for checking interface conformance among composed components.

From the above discussion we can conclude that our approach to concurrency lies between the completely formal approaches of Wright and Darwin ADLs, and the informal ones followed by most component models. Although the whole system has not been formalised, process algebraic semantics are used for active atomic components that in our model encapsulate the most complex, and hence most error-prone, concurrent behaviour. Passive and active components are explicitly distinguished which is not the case in most component models. Synchronisation is also completely handled at the modelling level which is more fine grained than what most other component models provide.

### 9.3 Expressiveness of composition

Our composition and adaptation connectors exercise exogenous control, i.e. from outside, to components. As was explained in Chapter 6, our connectors are structurally and semantically self-similar. This means that after composition (or adaptation) a composite (or adapted) component is formed that can only be accessed from its top-level connector. Connectors describe symmetric control flow, i.e. for each executing process control starts from the ‘top’ of the connector and after executing one more more components at the ‘bottom’, according to the connector’s control-flow semantics, control returns to its caller from the top of the connector again. This property of connectors allows for hierarchical construction of composite connectors.

However, the strictly symmetrical control flow of our connectors raises the question of whether it limits the expressiveness of our model. In other words, is our model sufficiently expressive to be used for the modelling and construction of a wide range of systems? For answering this question, we have conducted an experiment against Workflow Control-flow Patterns (WCPs). In Chapter 8, WCPs were introduced, their relevance to our work was explained and a summary of mapping our connectors to WCPs was analysed. We briefly present these points below.

WCPs are patterns of control for workflow systems. This work is relevant to ours because we claim that our connectors actually represent patterns of control for component based systems. Actually, our connectors have the same role for component-based systems to that of WCPs for workflows, namely they define
control and data flow for a set of entities. These entities are components and workflow activities respectively. We evaluated our connectors and their composition against WCPs by mapping WCPs to our primitive and composite connectors. In essence, in the current version of our component model, we can describe the control flow of approximately half of the WCPs. Extensions to the model that would support cancellation semantics for components and where some extra loop connectors would be introduced, would increase the coverage of WCPs to thirty seven out of forty three. The remaining six patterns cannot be modelled, because doing so would lead to the violation of some fundamental properties of our component model, for example violation of the self-similarity of connectors.

However, a larger concern regarding the mapping procedure, is that we mostly map pairs of WCPs to our connectors. For example whereas in WCPs branching and joining of processes are separate patterns, in our case they are part of the same (cobegin) connector. As was explained in Section 8.5, this is because our connectors express coarser grained forms of control. On the other hand, our connectors are more generic because they are parameterisable, and a single connector can be used to model many (pairs of) WCPs. We next evaluate these findings within the context of CBSD.

9.4 Discussion

In our model we have achieved algebraic compositionality for both components and connectors. Compositionality is achieved at the expense of restricting the control flow that can be expressed by our connectors into strictly symmetric forms of control. Symmetric control flow is necessary in order to achieve self-similarity which then leads to compositionality. Without some form of self-similarity in control flow structures, compositionality of connectors would not have been possible. For instance, this is the case with WCPs.

Our thesis is that compositionality of components and connectors is a very important aspect in software engineering in general, and in component-based software development in particular. In CBSD, systems are composed out of pre-built components. Compositionality of components means that the composition of two or more components leads to a composite component that can be further reused in another system. Compositionality of connectors means that we can
derive complicated control structures out of simple ones. In essence, compositionality gives the opportunity for greater reuse which is one of the main targets of CBSD. Composition of independent components in both the design and deployment phases of the idealised component life cycle is certainly a step in this direction. Generic, parameterisable control structures in the form of connectors is the next step. Structured, algebraic composition of connectors constitute a further advancement.

But compositionality is not everything in CBSD. A CBSD method, as determined by the component model, should delineate mechanisms for defining and handling concurrency. Modelling of both active and passive components is important and in fact, all of the concurrency offered by a component model should be defined at the modelling level. Ideally, a formalism should be used for checking against concurrency errors. Our model lies between completely formal approaches and informal ones. In that respect, we advance the state of the art by proposing a new way for bridging this gap, by introducing active atomic components formalised in CSP. We also contribute by offering a compositional approach to active and passive components.

Finally we provided an implementation that complies with the idealised component life cycle. Both active and passive component are implemented. The constructs that were introduced for defining and handling concurrency at the modelling level are supported by their appropriate implementation counterparts.

We therefore believe that the above strengths of our model outweigh the weaknesses that symmetric control flow entails. In Sections 8.4, 8.5 and 9.3 it was explained that six patterns cannot be described in our model due to a semantical violation. From a detailed examination of these patterns in [80], we have concluded that only WCP-10 and WCP-22 describe general control flow patterns, namely arbitrary cycles and recursion. The other four patterns describe rather specialised forms of control and the reader is referred to [80] for more details. We believe that recursion per se is not a limitation, since in our model we can have structured loops as top-level connectors in a system. The effect of disallowing arbitrary cycles cannot be objectively measured unless there is a complete formalisation of our work. An informal argument against the importance of arbitrary cycles in our model, is that it does not significantly effect the computation that a system built using our connectors can perform. This is because the computation units inside passive atomic components, as well as the active computation
processes inside active atomic components, can indeed include arbitrary cycles.

Regarding the limitation of our component model with respect to dynamic, or run-time, composition and reconfiguration, it is obvious that this constitutes a disadvantage of our model. However, this is a deficiency that can be corrected in sub-sequent versions of our model. Dynamic reconfiguration does not violate any of the fundamental principles of our model, and therefore it is feature that can be later on added to it.

To summarise, we believe that we offer a model that is sufficiently expressive to build a large range of software systems while promoting reuse. The property of compositionality is very important as it facilitates reuse for both components and connectors. Certainly, in order to evaluate how useful our approach is in practice, we would need to implement numerous case studies. The initial choice of atomic components that would be deposited into the repository is crucial and for that domain knowledge would be needed [59]. However this lies outside the scope of this thesis. In this thesis we have laid out the principles for larger reuse in the context of software component models.

9.5 Chapter summary

In this chapter we evaluated the work in this thesis. Naturally, it has its strengths and its weaknesses. The main strength of our model is that it provides a unified modelling framework for component and connector composition according to the idealised component and connector life cycles. Explicit support for concurrency is provided at the modelling level. By contrast, the main weakness of our approach is that the control patterns that are described by our composition operators are limited. Notably, recursion and arbitrary cycles cannot be defined by our connectors. As we explained, we believe that in the context of CBSD, the strengths outweigh the weaknesses.

In the next chapter, we conclude the thesis and discuss future research directions.
Chapter 10

Conclusions and future work

In this thesis we set-out to advance the state of the art in CBSE by defining a new software component model. In this chapter we conclude our efforts and we indicate future research directions.

10.1 Conclusions

We have defined a software component model with concurrency and compositionality. Compositionality constitutes the linchpin of our component model. Compositionality of components means that the composition of two (or more) components, always yields a composite component which can be further reused through composition. Compositionality of connectors means that the composition of two (or more) connectors, always yields a composite connector which be used for further component or connector composition. This form of compositionality has been made possible because first, our connectors encapsulate (symmetric) control flow, and second, in our model there is a clear separation of computation, which may be concurrent, from control which may be also concurrent.

So does our component model advance the state of the art? As we have already explained, compositionality has a central role in CBSE and it actually defines the idealised component life cycle. We therefore believe that indeed, our work does advance the state of the art as it is the sole model supporting the idealised life cycle. As with all new works however, so in ours, there are certain weaknesses, the most important of them deriving from the very nature of our connectors. Our connectors express symmetric control flow, which inevitably means that there are some control flow patterns, apparently the non-symmetric ones, that cannot be
expressed in our model. But as we have explained in our evaluation, this does not constitute a significant drawback of our work because inside atomic components, both passive and active, arbitrary control-flow forms may exist.

10.2 Future work

Research is open-ended, and at the moment some research results have been obtained, new research questions and opportunities emerge. In the context of a PhD study and due to the time constraints that are imposed, the amount of work that can be performed is always limited. The work that we presented in this thesis forms a solid foundation on which some interesting extensions can be made. These will further enhance the state of the art in component based software engineering and are explained below.

10.2.1 Compositional reasoning

_Compositional reasoning_ or _compositional verification_ [53, 131] allows the system developer to infer overall system properties from its components’ properties. It has been originally developed as a technique for reasoning about program correctness, and hence the term ‘component’ refers to ‘sub-program’. We will first present the original definition for compositional reasoning discussed in [53] and from there we extrapolate to CBSE.

The property required for compositional reasoning of programs is _compositionality_ and is defined as follows:

“That a program meets its specification should be verified on the basis of specification of its constituent components only, without additional need for information about the interior construction of those components.”

As explained above, the term component refers to (sub-) program. In order to be able to reason compositionally about systems, a _compositional proof method_ is needed that consists of:

---

1 Apparently we have used the term _compositionality_ throughout the thesis for a different purpose. Whenever this term is used hereafter, it will be made clear from the context which of the two usages is meant.
1. **Basic techniques** for proving that a program $P$, which is not decomposed any further, satisfies $\phi$.

2. **Compositional proof techniques** to handle the case that $P$ is composed of parts $P_1, \ldots, P_n$, i.e., $P = \text{op}^{\text{lang}}(P_1, \ldots, P_n), n \geq 1$, with $\text{op}^{\text{lang}}$ some operator of the programming language.

For the second case *compositional proof rules* are developed, i.e. rules of the form:

“From $P_1$ satisfies $\phi_1$ and $\ldots$ $P_n$ satisfies $\phi_n$ infer $P$ satisfies $\phi$.” [131]

Compositional reasoning therefore refers to proving that some program $P$ satisfies some property $\phi$ based on the properties satisfied by the sub-programs that it consists of and on the language operator used. The sub-programs can be concurrent and their properties are specified using their “observable behaviour, only” [53]. If a program cannot be decomposed any further, then basic techniques are required for proving that it satisfies its specifications. An additional requirement for the observable behaviour of sub-programs, is that the observable behaviour of each one is completely independent of the observable behaviour of the rest of the sub-programs.

From the above discussion it is clear that compositional reasoning constitutes a great asset for reducing the complexity of concurrency verification. The links between compositional reasoning and CBSE are also straightforward to establish. The fact that this technique strictly only requires the observable behaviour of components, complies with the black-box nature of components [123, 24, 33]. Given components $C_1, \ldots, C_n$ satisfying properties $\phi_1, \ldots, \phi_n$ and a composition operator $\text{op}^{\text{comp}}$ we want to infer that $C = \text{op}^{\text{comp}}(C_1, \ldots, C_n), n \geq 1$ satisfies $\phi$. As aptly pointed out in [126], the restriction that the sub-components must be completely independent of each other may be too strong a requirement for some component models. However, in our component model, at least during design-phase composition, this requirement holds by definition.\(^2\) Independent components also means that in our model we have a clear separation of component behaviour (defined as computation inside atomic components) from component interaction (defined by connectors). The following example illustrates these points.

\(^2\)In the deployment phase where we relax our strict composition rules by allowing shared sub-components, the above requirement is not always satisfied.
10.2. FUTURE WORK

A motivating example

To illustrate the compositional reasoning aspect of our model and the importance of defining completely independent components and of separating component behaviour from component interaction, we present an alternative way to construct the Single-Lane Bridge Problem, which was originally presented in [89] and revisited in [127]. The problem states that there is a narrow bridge, and that on its two sides there are blue and red cars waiting to cross. Because the bridge is narrow, cars can only pass in one direction concurrently and no overtaking is allowed. In [89], blue and red cars are defined using FSP, and so is the bridge controller (Figure 10.1(a)). The specification is model checked, and when errors are found, successive corrections of the model (i.e. redefinition of both cars and the bridge controller) produce the final specification. The fact that both the cars and the bridge controller need to be redefined in order to get the correct solution is not surprising, because in FSP behaviour is not separated from interaction. Therefore for the interactions request, enter and exit of the bridge both components need to synchronise. As a result, the way blue and red cars interact does not only depend on the bridge controller, it also depends on the cars themselves. As we show later, the way blue and red cars are defined should not affect their interaction.

In [127], the authors define a rich set of synchronisation connectors (effectively different channel and port types) that enforce variations of synchronous/asynchronous communication between concurrently executing components. In this work, channel ports semantically belong to connectors, and components may only exchange messages via attached connector ports. In the proposed solution, components are the blue cars, the red cars, a blue controller and a red controller.

---

3We briefly discussed the problem and our proposed solution, in [79].
When these components are connected with appropriate connectors, they get the correct behaviour of their system. In this work, there is a clear separation between component behaviour and interaction. However, as can be seen in the figure, there is not a consistent treatment of components. Components can be used either to specify behaviour, or to specify interactions between other components. This inconsistency means that some of the components are ad hoc entities created to specify interactions for a specific system. This contrasts with the view in CBSE, where components are predefined entities residing in a repository. Additionally, their approach is not compositional, i.e. there is no composition operator for composing components into composite ones.

The single-lane car-bridge problem can be solved using our model. For that purpose, we need two active atomic components representing the blue and the red cars respectively, and one controller that decides when the cars pass the bridge. As shown in Figure 10.2, different composition connectors give different overall behaviour to our composed system. For example, when the cobegin is used (Figure 10.2(a)), both blue and red cars enter the bridge at the same time, leading to a collision. This solution is therefore unsafe. When a non-deterministic selector is used (Figure 10.2(b)), cars of a certain colour may dominate the bridge, since the decision of which cars pass the bridge is made non-deterministically. This solution is therefore unfair. Finally, when a sequencer is used (Figure 10.2(c)), both blue and red cars are forced to alternate on the bridge. This solution is the correct one to our problem and is both safe and fair.

As expected, in our solution the two active atomic components are derived from the same formal specification. For completeness, this specification is provided in Appendix C.4. Here we provide a brief overview. Every car first requests to enter the bridge, then waits before entering, and finally it exits: \( CAR(i) = request!i \rightarrow enter!i \rightarrow exit!i \rightarrow CAR(i) \), where \( i \) is the id of each car. The rest of the processes in the active atomic component accumulate these
requests and when invoked by the channels connector, they allow all cars that have requested to enter the bridge to actually enter it. After all cars pass the bridge, the channels connector needs to be invoked again for returning the sequence of the ids of the cars that crossed the bridge. The interface of the generated component therefore consists of two methods, \(*\text{passAllRequested}()\) that triggers all cars that have requested to enter the bridge, and \(*\text{passed}(): \text{int}[]\) that returns the ids of the cars that crossed the bridge. Of course, since this is an active component, both methods can be invoked within a single call to the component.

Related works

As it is clear from the above, by completely formalising our component model, we will be able to reason compositionally about the properties of interest of composite components. This is something new in CBSE and it is not achieved by the component models surveyed Section 2.3. Some ADLs (for example Darwin, Wright) can indeed perform formal verification but this is not achieved compositionally, as component behaviour and component interactions are mixed. Therefore the fundamental requirement of compositional reasoning for independent observable behaviour of components is not satisfied.

The idea of combining component-based modelling techniques with compositional reasoning has been exercised in [67] and in [130]. However, in these works the *engineering* aspect is missing. In both approaches, ‘components’ and ‘connectors’ are abstract, mathematical constructs. In [67], the sole composition operator is the parallel composition operator (∥) whose inputs are the constituents (sub-) components and extra parameters defining how these components should interact as shown in Figure 10.3. In the figure we arbitrarily chose, for the sake of the

![Figure 10.3: The cars-bridge example.](image-url)

example, to first compose all red cars, then all blue cars, and then compose the
two kinds of cars together. However, in this approach connectors are different entities from the composition operator; connectors simply define how the composed components may interact and a different composition operator actually composes them. This is in contrast to our approach, where our composition operators explicitly define all interactions among components. More importantly, and as illustrated in the Figure, whenever composition occurs, the result is flattened into two levels, the interaction and the behaviour levels. In our model the result of the composition is a separate, reusable entity that encapsulates all the interactions between the sub-components and abstracts these interactions into its interface.

In [130], component interfaces are sets of input and output messages types, and composition is achieved by directly mapping compatible input and output component messages types. As a result, this model does not separate behaviour from interaction, and changes in the requirements usually means reformulation of both components and their interface mappings. In other words, there are no reusable interaction constructs than can be interchanged.

Concluding, formalising our work in order to achieve compositional verification of properties of interest, constitutes the most important extension of this work. The first steps have been already taken by using CP-nets, Z, and CSP, albeit for different purposes covering distinct parts of our model. Formalisation of our work will also have additional benefits, other than compositional reasoning. For example, we could formally check how constraining symmetrical control flow turns out to be, as we have discussed in Section 9.3. Another important use of formal semantics is to be able to check behavioural equivalence of composite connectors as we discuss in the following section.

10.2.2 Behavioural equivalence of composite connectors

We have defined in Section 6.2 formal semantics for our connectors. Specifically, we have extended the CP-net semantics and we defined connector template nets (CT-nets) for describing the behaviour of connectors. So far, we have only used CT-nets for unambiguously defining the behaviour of connectors and their composition. However, as the following example demonstrates, CT-nets could be used for defining behavioural equivalence among connectors.

Let us assume that we want to build the composite connector shown in Figure 10.4(a). Although it is possible to define that connector in one step, we notice that this connector defines a repetitive pattern of a binary cobegin connecting
to two binary sequencers that share a branch. Hence we define this pattern as a composite connector $conn$ shown in Figure 10.4(b). The defining expression is: $conn = cb2(seq2(CP, CP1), seq2(CP1, CP))$. Having defined connector $conn$, it is straightforward to define an equivalent composition to that of Figure 10.4(a), using the expression: $sel2(conn(CP, CP1, CP2), conn(CP1, CP2, CP))$. This is shown in Figure 10.4(c).

We used the above example to demonstrate that it is possible to build a composite connector out of other composite ones. The constituent composite sub-connectors may be part of a top-down decomposition of the final connector one wishes to build. The two approaches to connector construction, i.e. the compositions in figures 10.4(a) and (b) have to be proved to be equivalent. However, currently we can only informally argue that this is the case. Formal semantics allows us to formally prove that the two connectors are equivalent based on a notion of behavioural equivalence. Specifically, we could use the CT-net semantics and define the behaviour of a connector in terms of the traces that it can generate on its composition places. For instance the traces generated for each execution of connector $conn$ are $(CP_L \rightarrow CP1)\parallel (CP1 \rightarrow CP_R)$, where indices $L$ and $R$ differentiate the left and right $CP$s, $\rightarrow$ is read as then and $\parallel$ stands for parallel. However, in current work this remains an open issue.

It is worth mentioning that behavioural equivalence would additionally allow to compress a large connector’s architecture into a smaller, more compact one,
which is formally proved to be equivalent to the original one. This is important because large connector hierarchies yield performance penalties that might not be acceptable in certain applications. Compacting connector hierarchies is only meaningful if the final connector exhibits the same behaviour as the original one.

10.2.3 Other future work

In Section 8.4 we explained that there are three categories of workflow control-flow patterns that cannot be mapped into our model. First, there are patterns that cannot be mapped because they require the generation of a component with a non-fixed interface, i.e. the length of input/output parameters is different for each invocation. Second there are patterns that in order to be mapped into our model require the ability to cancel the execution of our components. In the third, category there are patterns whose implementation would violate fundamental semantics of our model, for example that control must flow through connectors and cannot be arbitrary. Future work could investigate ways of implementing the first two categories of WCPs to make our component model more expressive. One possible solution for the first category is the introduction of specialised top-level connectors. In the second category, the hierarchical nature of our model may prove instrumental in introducing cancellation semantics. For example, cancelling the execution of a composite component would recursively cancel the execution of the sub-components.

We have explained in Chapter 6 how connectors can be composed in order to form composite connectors. Related works focus instead on building connectors out of primitive construction elements that are not connectors. We consider this as an interesting extension to our work. For example, our connectors cannot be used in distributed systems without re-implemented them. Instead, we could modularise our connectors so that their connection points to other components are parameterisable. By varying these points we could add on demand distribution without having to re-implement them. Other services could be also included, for example encryption.
Appendix A

The Z notation

In this appendix we briefly present the Z [121] notation. Z is a formal specification language based on axiomatic set theory, lambda calculus and first-order predicate logic. All expressions in the Z notation are typed. A type of an expression denotes the “kind” of the expression. Types in Z can be either given set names, or compound types built from simpler types using specific mathematical constructs.

A.1 Z types

In every Z specification, given sets are the simplest “kinds” of expressions, that have no internal structure. For example, [BASIC_TYPES] defines the set of all basic types in a system. Z is equipped with the basic types $\mathbb{Z}$ and $\mathbb{N}$, the sets of integers and natural numbers respectively. Sets may be also enumerated, or they may defined using set comprehension constructs. For example:

$$\{4, 5, 6, 7\}$$
$$\{x : \mathbb{N} \mid x > 10\}$$

Both these sets are of type $\mathbb{PZ}$. $\mathbb{P}$ is the symbol for powerset, and denotes the set of all subsets of a set.

The cartesian product $\times$, defines compound types. For example, the ordered pair $(3,4)$ is of type $\mathbb{Z} \times \mathbb{Z}$. In general, if objects $x_1, \ldots, x_n$ are of types $t_1, \ldots, t_n$ respectively, then the tuple $(x_1, \ldots, x_n)$, is of type $t_1 \times \ldots t_n$.

Schemata are used for describing the state space of a system. Consider for example the schema type $Operation$. Like most schemata, it consists of two parts.
The one above the dividing line describes the variables of the schema and their
types, and the one below describes the relationship between the variables and
forms the invariant of the schema.

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>name : STRING</td>
</tr>
<tr>
<td>inputParams : seq(BASIC_TYPES)</td>
</tr>
<tr>
<td>outputParams : ( \mathbb{P} ) BASIC_TYPES</td>
</tr>
<tr>
<td>#outputParams \leq 1</td>
</tr>
</tbody>
</table>

Every operation consists of a name, which is of type \( \text{STRING} \), a possibly
empty sequence of input parameters, and a set of output parameters. The invari-
ant describes that an operation may have at most one output parameter.

For an operation that sums two integers and returns their result, there is
a binding \( \text{sumOp} = \langle \text{name} \Rightarrow \text{sum}, \text{inputParams} \Rightarrow (\mathbb{Z}, \mathbb{Z}), \text{outputParams} \Rightarrow \{\mathbb{Z}\} \rangle \). This binding has the schema type \( \langle \mid \text{name} : \text{STRING}; \ \text{inputParams} : \text{seq}(\text{BASIC\_TYPES}); \ \text{outputParams} : \mathbb{P} \text{BASIC\_TYPES} \mid \rangle \).

Finally, compound types may be also constructed by using relations and func-
tions. A binary relation over two sets \( X \) and \( Y \) defines all the pairs of \( x : X \) and \( y : Y \) such that this relation holds. Notation \( X \leftrightarrow Y \) means the set of all
binary relations between \( X \) and \( Y \) and is a synonym for \( \mathbb{P}(X \times Y) \). Functions
are a special type of relations, where each member is \( X \) is mapped to at most
one member of \( Y \). As a result, functions and relations have the same type.

### A.2 Mathematical constructs

The Z notation comes with a set of mathematical definitions (mathematical tool-
kit) whose purpose is to compactly and simply describe structures frequently used
in program specifications. In this appendix, we describe the parts of the tool-kit
used in this thesis.

#### A.2.1 Sets

As already discussed, \( \mathbb{P} S \) is the set of all subsets of some \( S \). We also define
\( \mathbb{P}_1 S = \mathbb{P} S \setminus \emptyset \) for any set \( S \), where \( \emptyset \) is the empty set. For every non-empty set
of integer numbers, the minimum member and maximum member of the set may be obtained using the \textit{min} and \textit{max} functions. The symbol $\mathbb{N}_1$ denotes the set of strictly positive numbers. A set may be partitioned into different subsets, eg. $\langle S_1, S_2, S_3 \rangle$ \textbf{partition} $S$, and it means that the union of the sub-sets $S_i$ return $S$, while the sub-sets are pairwise disjoint.

\section*{A.2.2 Relations-Functions}

Functions \textit{first} and \textit{second} are projection functions for ordered pairs, giving the first and second coordinates of an ordered pair respectively. For a relation $R \equiv X \leftrightarrow Y$, domain of $R$ (\textit{dom} $R$) is the set of all members of $X$, which are related to at least one member of $Y$ by $R$. The range of $R$ (\textit{ran} $R$) is the set of all members of $Y$, which are related to at least one member of $X$ by $R$.

A functions is a special kind of relation, that relates each element in their domain, with exactly one element in their range. For functions, the following notations are used:

- $X \rightarrow Y$ - Partial functions from $X$ to $Y$. These are functions whose domain need not be the whole of $X$.
- $X \rightarrow\rightarrow Y$ - Partial injections from $X$ to $Y$. These are partial functions that are injective, i.e. they map different elements of their domain to different elements of their range.

Finally, surjections from $X$ to $Y$ are the functions whose whole range is the whole of $Y$.

\section*{A.2.3 Sequences}

Sequences are finite functions from positive natural numbers to some set $X$. We write $\langle x_1, \ldots, x_n \rangle$ as a shorthand for $\{1 \mapsto x_1, \ldots, n \mapsto x_n\}$, where $x_i \in X$, $1 \leq i \leq n$, for some natural number $n$. Symbol $\mapsto$ is the ‘maplet’ notation, a graphic way for expressing ordered pairs.

For sequences, we have used the following notations:

- $\text{seq } X$ - The set of finite sequences over $X$.
- $\text{seq}_1 X$ - The set of non-empty finite sequences over $X$.
- $\text{iseq } X$ - The set of injective finite sequences over $X$, ie. the sequences over $X$ that contain no repetitions.
front $s$ - Function that returns a sequence containing all elements of $s$, except for the last.

last $s$ - Function that returns a sequence containing all elements of $s$, except for the first.

head $s$ - Function that returns the first element of sequence $s$.

tail $s$ - Function that returns the last element of sequence $s$.

$U \upharpoonright s$ - Function that given a set of indexes $U$ and a sequence $s$, returns a sequence that contains exactly those elements of $s$ that appear at an index in $U$, in the same order as in $s$. 
Appendix B

Coloured Petri nets

In this appendix we provide a brief and informal introduction to Coloured Petri Nets (CP-nets). Purpose of this presentation is to provide sufficient material for the reader to be able to understand the CP-nets presented in the thesis. Information about the history and development of Petri nets may be found in [102]. A detailed presentation of CP-nets with an introduction to related formal analysis methods may be found in [73].

We first introduce the readers to ordinary Petri nets in Section B.1. We then present in Section B.2, Coloured Petri nets, a high-level form of Petri-nets.

B.1 Place-Transition Petri nets

Petri nets or Place-Transition (PT) nets are a special kind of bipartite, directed graphs. They consist of two kind of nodes, called places and transitions, and directed arcs with weights from places to transitions, or from transitions to places. Graphically, places are drawn as circles and transitions as small rectangles. Each arc is labelled with non-negative natural number, which represents its weight. Unary weights are usually omitted. Places in Petri nets can contain a non-negative number of tokens. The assignment of tokens in the places of a Petri net designate the state, or the marking of the Petri net. The initially marking of a Petri net is symbolised with $M_0$. Tokens are graphically represented as black dots.

Places and transitions in Petri nets are collectively called nodes. Nodes can be input or output nodes. A node $x$ is an input node of another node $y$, if and only if (iff) there exist an arc from $x$ to $y$. In this case $y$ is an output node of
Therefore in a Petri net we can talk about *input places*, *output places*, *input transitions* and *output transitions*. The arcs directing towards a node are called *input arcs* of the node, while the arcs originating from a node are called *output arcs* of the node.

The behaviour of Petri nets is captured in terms of their states (markings) and the transitions from one state into another. Starting from the initial state, transitions may move tokens from their input places into their output places. To facilitate the rest of the discussion, let \( w(i) \) be the weight of the \( i \) arc from the input place to the transition and \( w(j) \) the corresponding weight of the \( j \) output arc. A transition is *enabled* iff at each input place there are at least \( w(i) \) tokens, otherwise the transition is *disabled*. An enabled transition may or may not *fire* or *occur*. After the transition occurs, all \( w(i) \) tokens are removed from each input place, and \( w(j) \) tokens are placed in each output place. The Petri net then reaches a new marking.

As an example, we consider the creation of two molecules of water, out of hydrogen and oxygen based on the example originally presented in [102]. In Figure B.1 the well-known chemical reaction \( 2H_2 + O_2 \rightarrow 2H_2O \) is presented. In

![Diagram of Petri Net](image-url)

Figure B.1: Forming of water molecules and hydrolysis using CP-net.
the figure tokens are used to represent molecules of hydrogen, oxygen or water, depending on the place they are in. We also use rectangles with thicker borders to emphasise the transitions that are enabled.

In the initial marking in Figure B.1(a), only the JOIN transition is enabled because it is the sole transition with sufficient tokens in its input places. After JOIN fires, two tokens are placed in the $H_2O$ place, and, two and one tokens are removed from $H_2$ and $O_2$ places respectively. This is due to the weights of the respective arcs. Now the Petri net is in the state shown is Figure B.1(b) and only transition SPLIT is enabled that corresponds to hydrolysis. After it fires, the Petri-net transits to the state shown in Figure B.1, which is the same as the initial one.

### B.1.1 Concurrency

In Petri nets two or more transitions may be *concurrently enabled*. These means that there are sufficient tokens in the input places of these transition so that all of them may fire. Let us consider for example the control flow of the `cobegin` construct of concurrent programming [20] inside a while-true loop:

```java
while(true){
    cobegin S1 || S2 coend
}
```

$S_1$ and $S_2$ represent statements that execute in parallel.

The Petri-net in Figure B.2 represents the control-flow of the previous code snippet. Now tokens represent threads of control. In the initial marking only the SPLIT transition is enabled, but after it fires, transitions $S_1$ and $S_2$ are
concurrently enabled. This means that both may fire concurrently, or either of them, as is the case with the cobegin statement.

In Petri nets it is also possible for a transition to be concurrent with itself. This is shown in Figure B.3(a), where transition $T$ may fire once, or it may fire twice concurrently. In Figure B.3 we show the state of the Petri net after $T$ fires twice concurrently.

![Initial marking of the PN](image1)
![After concurrent occurrences of T](image2)

Figure B.3: A concurrent transition to itself.

B.1.2 Non-determinism, source and sink transitions

Petri nets can be used to express non-determinism. In general non-determinism occurs when two (or more) transitions are enabled, but they are not concurrently enabled, therefore the firing of one transition disables the other transitions. Figure B.4(a) presents such a case for transitions $T1$ and $T2$, and as such, only one of the transitions may fire. This situation is called conflict. Specifically, two nodes (places or transitions) are in conflict if and only if there are two paths to them starting at different transitions, and these transitions share at least one of their input places. Therefore in Figure B.4(a) places $P1$ and $P2$ are in conflict. Figure B.4(b) demonstrates that a node (place $P1$) may be in conflict with itself.

![Conflict](image3)
![Self-conflict](image4)
![Confusion](image5)

Figure B.4: Non-determinism in Petri nets.
B.2. COLOURED PETRI NETS

In Figure B.4(c) a situation termed *confusion* is presented. It is conflict with concurrency. \( T_1 \) conflict with \( T_2 \) and \( T_3 \) conflicts with \( T_2 \). Therefore, either \( T_1 \) and \( T_3 \) may fire or \( T_2 \), but not all three of them. Once a transition fires, then the Petri net becomes deterministic.

In Figure B.4(c) all transitions are *sink* transitions. A sink transition is a transition with no output arcs and consumes tokens. A transition with no input arcs is called a *source* transition. Source transitions are unconditionally enabled and produce tokens.

B.2 Coloured Petri nets

Coloured Petri Nets (CP-nets) are considered high-level Petri nets. The word ‘coloured’ is used for historical reasons, [72]. As with other forms of high-level Petri nets, CP-nets do not add to the expressiveness of Petri nets. Instead, they represent a way of giving a more compact description of a Petri net. In CP-nets tokens are *coloured*, where a colour represents an arbitrarily complicated data type. Therefore tokens represent values of the data type. In every CP-net, all tokens in each place must be of the same type. This type is called the *colour set* of the place.

We introduce the basic notions of CP-nets through a generic example shown in Figure B.5. In Figure B.5(a) the type declarations are presented. The syntax used for declarations is the one used by the tool [49]. The tool uses the language

Figure B.5: Declarations and initial marking of generic example of a CP-net.
CP-net ML which is based on the functional programming language Standard ML [101]. We have declared three types, a type $N$ for natural numbers\footnote{The exact declaration is \texttt{colset N=int with 0..Option.valOf(Int.maxInt);}}, a case id type $CID$ and the cartesian product of two integer types $N \times CID$. A set of variables with their respective types are also declared.

In Figure B.5(b) the initial marking is shown and some conventions are followed. Every place has a colour set, which in the figure is shown in an italicised font. For example the colour set of $P1$ is $N\times CID$ while the colourset of $P5$ is $N$. The marking of each place is a multi-set over its colour set. A multi-set allow the occurrence of the same element multiple times. The initial marking is determined by the initialisation expressions, i.e. the underlined expressions next to the places. In Figure B.5, only place $P1$ has initialisation expressions, namely two pairs $(1, 100)$ and the single pair $(2, 500)$. In ordinary Petri nets the current marking was represented by placing black dots in the respective places. In CP-nets, the current marking is denoted by a small circle containing a number that declares the total number of tokens in the place. Next to the circle, a string describes the multi-set of tokens in the place. For instance, in Figure B.5(b) that represents the initial marking of the CP-net, place $P1$ has three tokens which correspond to the initialisation expression of $P1$. In the initial marking, only $T1$ is enabled.

Figure B.6(a) shows the current marking after transition $T1$ fires three times. The output arcs of $T1$ have conditional expressions based on the incoming tokens that describe what tokens will be output to the respective output places. For example the arc from $T1$ in $P3$ will place the token $(i + 1, c)$ to $P3$ for every input token $(i, c)$ to $T1$, iff $i$ is less than or equal to one. Similar hold for the tokens from $T1$ to $P2$. In the new marking, only transition $T2$ is enabled.

Figure B.6(b) shows the result of transition $T2$ firing. In order for $T2$ to fire, its input arc expressions are evaluated. Two pairs are required whose first members are identical, while the second members may vary. The pair placed in the output place $P3$ retains the first member of the incoming tokens, but sums the second members. In general, an arbitrary function may be used instead of the summation one, as long as it is defined for any possible input values.

At current marking in Figure B.6(b), only transition $T3$ is enabled. Transition $T3$ demonstrates the use of a guard. A guard contains a boolean expression and its purpose is to define additional conditions for the transition to become enabled.
**B.2. COLOURED PETRI NETS**

In the figure, transition $T3$ becomes enabled iff the value of $i$ in the input arc is greater than one. Since with the current marking this is true, transition $T3$ fires and the value of $i$ is passed to $P5$, thus transiting to the final marking in Figure B.6(c).

**Fusion sets**

Fusion sets allow the system modeller to conceptually ‘fuse’ a set of nodes into a single node, without having to graphically represent them as a single node. We proceed with an example of fusion places in Figure B.7(a). In the figure we have omitted the initial marking, the arc expressions and the place colours, because the purpose of this figure is to introduce the notion of fusion places. In the figure there is the fusion set $\text{FusA}$ containing two members, $P2$ and $P6$. To graphically identify the members of the set, the $\text{FP}$ tag is used along with the fusion set name. The members of the fusion set are considered to be identical. Figure B.7(b) shows the equivalent CP-net after the two places in the fusion set

---

**Figure B.6:** Subsequent markings of generic example of a CP-net.
 Besides fusion places it is also possible to define fusion transitions. Fusion transitions are defined in a similar fashion, however since these are not used in this thesis we do not elaborate further. Finally, fusion sets are related to hierarchical Petri nets [71, 73], where *global fusion sets*, *page fusion sets* and *instance fusion sets* have been defined. In a hierarchical Petri net, a sub-page may be used multiple times. Each time a sub-page is referred, an instance of it is created. Page fusion sets are shared among all instances, whereas instance fusion sets are for that instance only.

Since in our work we do not use hierarchical Petri nets and the CP-net of a connector/component is presented within a single figure, the aforementioned distinctions are irrelevant in this discussion. In essence, using fusion places merely represents a graphical convenience and all places that belong to a fusion set actually represent the same place.

**Conflicts**

CP-nets are high level Petri nets, and therefore each CP-net can be translated (or unfolded) to an equivalent Petri nets and vice versa. This translation is unique when transforming a CP-net into a PT-net, but there is more than one way for transforming a PT-net into a CP-net. Because of this equivalence relation, in CP-nets a conflict case may arise that was not presented in Section B.4. This is shown in Figure B.8.

In Figure B.8(a) we only show the current marking and we omit the initial
marking and all declarations, because our focus is to define a new form of self-conflict that is applicable to CP-nets. In the figure, \( x \) and \( y \) are variables of the same type with \( a \), \( b \) and \( c \). Transition \( T \) is in conflict with itself, because although it is enabled for both tokens \( a \) and \( b \), it can fire for only one of them. The conflict becomes obvious when we unfold this CP-net to the equivalent PT net in Figure B.8(b). In the figure there is conflict between the distinct transitions \( T_1 \) and \( T_2 \).

### B.2.1 Additional remarks

Finally, we can make some generic remarks regarding Coloured Petri nets. According to [73], empty CP-nets, i.e. CP-nets that only have a set of declarations, are valid. Additionally, valid CP-nets can have isolated nodes and multiple arcs between nodes. Isolated nodes can be thought of as being connected through arcs that transfer the empty multi-set token. Multiple arcs can be consolidated into a single arc. These allowances are illegal in PT nets, but are used in CP-nets for pragmatic reasons, namely convenience for the development and use of CP-net tools.
Appendix C

Formal definitions of active atomic components

In this Appendix we present the formal definitions of the active CSP components presented in the main text of the thesis. For that purpose, we first provide a brief introduction to the CSP language in Section C.1. In subsequent sections, we choose to present the machine-readable definitions that we have used with FDR [5]. An interested reader can then use this specification directly with the tool.

C.1 The CSP language

We summarise the parts of the CSP language used in this presentation in the following pseudo-BNF rule:

\[
P ::= a \rightarrow P \mid c?x : T \rightarrow P \mid c!x \rightarrow P \mid P_1 \boxplus P_2 \mid P_1 \cap P_2 \mid \text{Stop} \mid \text{Skip} \\
\mid P \setminus C \mid P_1 \parallel P_2 \mid P_1 || P_2 \mid \text{if } b \text{ then } P_1 \text{ else } P_2
\]

The recursive definition \( P = a \rightarrow P \) describes a process that engages in event \( a \) and then behaves as \( P \). Event \( a \) must belong to the alphabet \( \alpha P \) of \( P \). Definitions \( P = c?x : T \rightarrow P \) and \( P = c!x \rightarrow P \) denote input of type \( T \) on channel \( c \) and output on channel \( c \) respectively. \( P = P_1 \boxplus P_2 \) is the external choice operator and is contrasted to the internal choice operator \( \cap \). In the first case the choice between \( P_1 \) and \( P_2 \) is deterministic and is controlled by the process environment. In the
second case the choice is non-deterministic and the process internally decides which process will execute next. Process Stop is the deadlocked process that can engage in no events. The Skip process participates only in the special event \( \checkmark \) to indicate successful termination. In \( P \setminus C \), all events in \( C \) are concealed and become internal transitions \( \tau \) of \( P \), i.e. \( \alpha(P \setminus C) = \alpha P - C \). With \( \Sigma \) the set of all possible events are denoted, apart from the termination and internal events. We write \( \Sigma^{\checkmark, \tau} = \Sigma \cup \{ \checkmark, \tau \} \) to include them.

The alphabetised parallel composition operator \( P_1 || [A | B] || P_2 \) denotes that \( P_1 \) and \( P_2 \) must synchronise on every event \( a \in A \cap B \). As a shorthand we write \( P_1 || P_2 \) to denote the parallel composition of \( P_1 \) and \( P_2 \) where the two processes must synchronise upon all events that are common in their default alphabets, i.e. events in their process definitions that are in \( \Sigma \). Therefore, \( P_1 || P_2 = P_1 || [\alpha P_1 | \alpha P_2] || P_2 \). On events outside their alphabets processes may interleave. \( P_1 ||| P_2 \) means that the two processes interleave all their events, even in the common ones. Process \( P = if \ b \ then \ P_1 \ else \ P_2 \), behaves as \( P_1 \) if the boolean expression \( b \) is true and as \( P_2 \) otherwise.

C.2 FDR model checking

When checking a CSP composition in FDR, among other checks, it is possible to check for deadlock and livelock freedom. Deadlock describes the situation where a process reaches a state that can perform no further action. Livelock or divergence [113] in CSP describes the situation where a process may enter an infinite unbroken sequence of internal, hidden actions. Livelock is considered as catastrophic a situation as deadlock, because the livelocked, or divergent process may never engage in external communications. Since by definition in our active atomic components all internal communications are hidden, Section 4.3.2, there is increased risk for livelock. In fact, almost all active atomic components we have defined would exhibit livelock, because all of their internal interactions are hidden, and it is very likely that some of these interactions form a loop. In order to overcome this problem, we apply the following solution.

We start by making a general observation that applies to all active atomic components. In active atomic components there are some processes that have, informally, the role of event sources. For example this is typically the case for processes modelling hardware sensors. These processes communicate via output
channels only, and in the case where no channels are involved (i.e. no data are communicated), a process may be considered the source of event triggers. In the above cases, these are actually communication events in real-time systems that occur according to a pre-determined period. As in the reasoning provided in [44], these events cannot be the source of divergence, as long as their period is sufficiently long to allow for other events to occur in between. These events incur loops of intended internal communications and are guarded by time intervals. The generated internal communications cannot be therefore considered as livelock because between consecutive event triggers, communication via other events is possible. However, since in our analysis and definitions we are using the untimed model of CSP, we cannot capture this information. Instead, we consider these events non-hidden for the purpose of livelock analysis. The above points become more clear in the examples that follow.

C.3 The FLIGHTS component

We now present the formal definition of the FLIGHTS component that was presented in Figure 4.16. Its formal definition in machine readable CSP is given in Figure C.1. It can be noted that for simplicity reasons we have used only two planes, PL0 and PL1. Also, in order to be able to efficiently model check the specifications, the possible times that planes can arrive or depart have been restricted to the range $(-1) \ldots 100$, instead of the more accurate $(-1) \ldots 2359$. The role of the override function is to allow placing a specific value in the sequence of the arrival and departure buffers. Finally, as discussed earlier, for the purposes of livelock freedom, the events generated for the two planes remain visible. This is shown in sub-systems zero and one.

C.4 The CARS component

In this section we present the formal definition of the CARS active component that was informally discussed in Section 10.2.1. The machine-readable specification is shown in Figure C.2. Process $CAR(i)$ represents the car with id $i$. It first asks to enter the bridge, then enters, and then exits. Process $P(s)$ is a FIFO buffer that accumulates the ids of the cars that have requested to enter the bridge. The process outputs all the car ids that have requested to enter the bridge on channel
C.4. THE CARS COMPONENT

--- Representing possible times (abstracting over the more accurate
--- (-1)\ldots2359 range). (-1) denotes unavailable.
nametype time = \{(-1)\ldots100\}

--- A plane non-deterministically chooses when to depart and when to
--- arrive. For simplicity we use two planes.
channel arr0, arr1, dep0, dep1 : time
PL0 = \{
  \{x : time \mid \text{arr0}(x) \rightarrow \text{PL0}\}
  \}\{x : time \mid \text{dep0}(x) \rightarrow \text{PL0}\}\}
PL1 = \{
  \{x : time \mid \text{arr1}(x) \rightarrow \text{PL1}\}
  \}\{x : time \mid \text{dep1}(x) \rightarrow \text{PL1}\}\}

--- Controller definition
channel addArr, addDep : \{0,1\}.time
CTRL = dep0 ? t \rightarrow addDep!0!t \rightarrow addArr!0!(-1) \rightarrow CTRL
       | arr0 ? t \rightarrow addArr!0!t \rightarrow addDep!0!(-1) \rightarrow CTRL
       | dep1 ? t \rightarrow addDep!1!t \rightarrow addArr!1!(-1) \rightarrow CTRL
       | arr1 ? t \rightarrow addArr!1!t \rightarrow addDep!1!(-1) \rightarrow CTRL

--- Departures buffer. Query the buffer for all departures, or add a
--- specific departure time for a plane.
channel allDppts, allArvls : Seq(time)
channel allDpptsGuard, allArvlsGuard
DEP(s) = allDpptsGuard \rightarrow allDppts!s \rightarrow DEP(s)
       | addDep?i?x \rightarrow DEP(\text{override}(s,i,x))
ARR(s) = allArvlsGuard \rightarrow allArvls!s \rightarrow ARR(s)
       | addArr?i?x \rightarrow ARR(\text{override}(s,i,x))

--- The events from the two planes are not hidden
SUBSYSTEM0 = PL0 \[\{arr0, dep0\}\]\ CTRL
SUBSYSTEM1 = SUBSYSTEM0 \[\{arr1, dep1\}\]\ PL1
SUBSYSTEM2 = (SUBSYSTEM1 \[\{addArr\}\]\ ARR(\text{override}(\langle-1\rangle,\langle-1\rangle))\} \{addArr\}\]
FLIGHTS = (SUBSYSTEM2 \[\{addDep\}\]\ DEP(\text{override}(\langle-1\rangle,\langle-1\rangle))\} \{addDep\}\]

--- It overrides a specific place in the sequence. It is another way
--- for writing s[i]=x
override(s,i,x)=
  if i==0 then concat(<<x>,tail(s)>)
  else concat(<<head(s)>, override(tail(s),i-1,x)>)

Figure C.1: CSP definition of FLIGHTS component.
−− assume only 2 cars per side

\textbf{nametype} \texttt{shortInt} = \{0..1\}

−− Define each car first

\textbf{channel} \texttt{request, enter, exit : shortInt}
\texttt{CAR(i) = request!i \rightarrow enter!i \rightarrow exit!i \rightarrow CAR(i)}

−− Channels to pass car sequences

\textbf{nametype} \texttt{carSequences} = \{<>\!,<0\!,<1\!,<0,1\!,<1,0\!\}
\textbf{channel} \texttt{getAllRequested, passed, toEnter, toExit : carSequences}
\textbf{channel} \texttt{passAllRequested, exited}

−− Simple buffer for accumulating requests
\texttt{P(s) =}
\hspace{1cm} \textbf{if} \texttt{null(s) then}
\hspace{1cm} \texttt{request?i \rightarrow P(<i>)}
\hspace{1cm} \textbf{else}
\hspace{1cm} \texttt{request?i: \texttt{diff(shortInt, set(s)) \rightarrow P(s<\!i>) [ ] \texttt{getAllRequested}s \rightarrow P(<>)}}

−− ENTER and EXIT processes are used as intermediates between
−− the INTERFACE and the buffer \texttt{P}
\texttt{ENTER(s) =}
\hspace{1cm} \textbf{if} \texttt{null(s) then}
\hspace{1cm} \texttt{toEnter?x \rightarrow ENTER(x)}
\hspace{1cm} \textbf{else}
\hspace{1cm} \texttt{enter.head(s) \rightarrow ENTER(tail(s))}

\texttt{EXIT(s) =}
\hspace{1cm} \textbf{if} \texttt{null(s) then}
\hspace{1cm} \texttt{toExit?x \rightarrow EXIT(x)}
\hspace{1cm} \textbf{else}
\hspace{1cm} \texttt{exit.head(s) \rightarrow}
\hspace{1cm} \textbf{if} \texttt{null(tail(s)) then}
\hspace{1.5cm} \texttt{exited \rightarrow EXIT(tail(s))}
\hspace{1cm} \textbf{else} \texttt{EXIT(tail(s))}

−− Defines the interface of the component
\texttt{INTERFACE =}
\hspace{1cm} \texttt{passAllRequested \rightarrow getAllRequested?x}
\hspace{2cm} \texttt{\rightarrow toEnter!x \rightarrow toExit!x \rightarrow exited \rightarrow passed!x \rightarrow INTERFACE}

−− Gradually build the system
\texttt{S1 = (INTERFACE[| {{getAllRequested}} | | P(<>)) \setminus {{getAllRequested}}}\}
\texttt{S2 = (S1[| {{toEnter | } | ENTER(<>)) \setminus {{toEnter}}}\}
\texttt{S3 = (S2[ | union({{toExit}}, {exited}) | | EXIT(<>))
\hspace{3cm} \setminus union({{toExit}}, {exited}}\)
\texttt{S4 = S3[ | request.0, enter.0, exit.0 | ]| CAR(0)
\texttt{SYSTEM = S4[ | request.1, enter.1, exit.1 | ]| CAR(1)}

Figure C.2: CSP definition of CARS component.
getAllRequested. The INTERFACE process can pass all cars that have requested
to cross the bridge. ENTER and EXIT processes are used to ensure that cars
enter and exit the bridge in a FIFO manner. Additionally, EXIT signals when all
processes have exited the bridge with the exited signal, so that the INTERFACE
process can then output on channel pass the id(s) of the car(s) that crossed the
bridge during the last request.

All events used for inter-process communication are hidden. For reasons ex-
plained earlier, we exclude the request, enter and exit events of the individual
cars. Notation \{ | x | \} is used to denote all events on channel x. The specification
in Figure C.2 has been model-checked for deadlock and livelock freedom, and it is
worth noting that if the above events were not hidden, the SYSTEM composite
process would appear to livelock.
Bibliography


