AN INVESTIGATION INTO COMMONSENSE REASONING

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

2011

By

Fardad Jabbary Aslany

School of Computer Science

Contents

Word count: 33,297 words

List of Figures

Abstract

In this thesis, we introduce commonsense reasoning, some of its features and reasoning types. We establish the Event Calculus as a logical formalisation to handle commonsense reasoning; and introduce circumscription as a mathematical machinery to implement default reasoning.

We define a framework in which we simulate a world scenario, initiated by an idea from Shin and Davis [40]. They simulate a real world scenario in which an agent moves from a location to another and fills in some buckets with liquid. They implement this in PDDL+. We develop their idea further, represent the scenario in the Event Calculus and elaborate on their formalisations weak points. We introduce a flagging system to deal with triggered events and prevent them from repeated occurrence. We show the elaboration tolerance of the Event Calculus and discuss that carrying out modifications on an already-developed framework does not need performing surgeries on the formalisation. We compare our Event Calculus formulas with PDDL+ of Shin and Davis. We show that their formalism not only does not handle many "commonsense" aspects of their own scenario, performing small changes in their scenario requires major modifications whereas in the Event Calculus representation this is not the case due to its elaboration tolerance.

Later in the thesis, a method to transform Event Calculus formulas into propositional logic will be introduced that can be fed into a SAT solver for automated reasoning. The results can be transformed back into Event Calculus formulas by reverse mapping.

Different automated reasoners that deal with the Event Calculus are discussed and SAT solving method is explained in more detail.

Declaration

No portion of the work referred to in the 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 Statements

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, 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 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 and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://www.campus.manchester.ac.uk/medialibrary/policies/intellectual-property.pdf), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University's policy on presentation of Theses.

Acknowledgements

I would like to thank my supervisor Dr. Ian Pratt-Hartmann for his support and guidance throughout the research and directing me in the path which was unclear before to me.

I would also like to thank my family, Dr. Farzad Jabbary Aslany, Dr. Shahnaz Hariri, Dr. Farnaz Jabbary Aslany, Shahrad Jabbary Aslany and John and Mahin Goodall and Q for their support during this time.

Special thanks to Dr. Elena Martin Ávila, Jaime Martin Ávila, Carlos Martin, Concepcion Inmaculada Ávila, Ana Sánchez, La Abuela, Amin Rismanchian, Brako and Chipy. And in the memory of Tuna.

Hilverd Reker, Lily Safie and Mo Pouya Haghighi have been amazing friends and colleagues with whom this difficult world is a better place to live in.

Many thanks to Dr. Daniel Neagu whose support I will forever appreciate. I also would like to thank Dr. Keshav Dahal for his positive thinking and good spirit. Special thanks to Emmanuel Giannopoulos and Los Cochinotes: Víctor Parrilla Mesa (Cochinote Grande), Rodrigo Martin (Tito), Concepción Fernández López (Conchita), Ana Navarro (hermana de Bea), Lorraine Richard (catwoman), Susel Arzuaga (la Cubana), Benoit Lepetitcolin (enamoredo de calimocho) and Jean Plateau.

This period of life was engraved into memory with the company of Claire Tolley, Silvia Sáenz Romo, George Peltekis, Braulio Girela, Laura Rubio Moreno and Alba Ibáñez-Vivanco.

I have great gratitude towards Malcolm Harper for his great help, support and understanding. Also towards my friends and colleagues at George Kenyon Hall, specially Saurabh Sunder.

At the end, many thanks to Dr. Daniel Espino, Dr. Pablo Vivanco, Dr. Rocío Ortuño Casanova, Cheryl Williams, Dr. Cesar Lopez Camacho, Dr. Lars Nilse, Dr. Gisela Orozco, Dr. Roberto Carrasco, Dr. Neil Armitage, Gregorio Barba, Ander, Deborah Twigger and Omid Karimpour.

Chapter 1

Introduction

For decades many scientists have been interested in building machines with human-like intelligence and computational power [1]. Even though there have been numerous advances in the field since its birth in 1940s, Artificial Intelligence simulation on a machine is still difficult due to the lack of full understanding of how a human's brain works.

1.1 Commonsense Reasoning

Commonsense reasoning is the process of using implicit world knowledge for making inferences about a scenario based on the facts provided; the world knowledge we implicitly obtain by various methods such as learning from one's own experiences, others' experiences, reading about them, observation and so on. Most often, when we hear or read a sentence, we perceive some information. It is very clear, however, that not everything could be said in one sentence. So when we read a sentence and we understand it, we are using much more knowledge than could be extracted from the sentence. Therefore, the implicit knowledge which we already possess makes this understanding possible. This is exactly what commonsense reasoning is about.

For instance, when a person is watching a scene in which an object is falling from some height, they *know* (or actually *infer*) that it is not going to stop until it reaches the ground; and when it reaches the ground, it will either bounce or stay still, depending on the mass, material and other properties of the object and the ground.

During decades of research, various researchers have tried to describe and capture commonsense, starting with a call by John McCarthy [4] to use logic to build computer applications with commonsense. Following a suggestion by McCarthy, Lifschitz [10] created a list of commonsense reasoning benchmark problems which helped the researchers of the field to focus their attempts and which resulted in many new features added to the commonsense reasoning representational languages. Morgenstern keeps an up to date list of commonsense benchmark problems [11].

1.1.1 Reasoning Types

The kinds of reasoning that a human performs are not fully identified. The following, lists some of the reasoning types that have been simulated on a machine. The list is by no means exhaustive but contains the reasoning types that we are interested in commonsense reasoning simulation on a machine:

- Prediction
- Planning
- Postdiction
- Model Finding

Prediction consists of determining the state of the world after a sequence of actions. For instance in the example of a falling object, we predict that the object is going to land on the ground based on various facts, such as that we *know* when an object that is falling will continue to fall until it reaches a surface and will not magically disappear; an object that is falling is most likely to continue falling in a straight line; a falling object will be going towards the ground not upwards; and so on. Formally, we can present this as:

Initiates(Fall(object), Falling(object), t).

Terminates(HitSurface(object), Falling(object), t).

$\text{HoldsAt}(\text{Height}(\text{object}, h_1), t_1) \wedge h_2 = \text{Max}(0, h_1 - t_2^2) \Rightarrow$

$\text{Trajectory}(\text{Falling}(\text{object}), t_1, \text{Height}(\text{object}, h_2), t_2)$.

The above representations are in the Event Calculus (EC) format which we will fully describe later in the report. What they mean, however, is that the event of falling an object initiates falling of that object. The event of it hitting a surface will terminate the fact that object is falling. The last axiom reads that if an object starts falling at time t_1, then its height at time t_2 will be $Max(0,h_1-t_2^2)$. The above representation is not a full prediction problem represented in EC, it is only a simple representation of small facts. However it is by the combination of these little facts that a full proposition can be proved; we will show this later in the report.

Planning consists of determining what events will lead to a final state from an initial state in the world. For instance if we know there is a hungry cat in the kitchen and there is a piece of meat on the table and the cat can reach the table, then we could devise a plan in which the cat approaches and eats the meat.

Postdiction consists of determining an initial state of the world given a sequence of events and a final state. For instance, if the cat is no longer hungry, then there was some food in the kitchen for the cat to have.

Model Finding consists of generating possible models from states of the world and events that can happen in the world. For example, if Fred is at home and he potentially can turn off the light, the TV or the fan, and that each of these actions have different effects, it is possible that at a later timepoint the light is off, or the TV is off, or the fan is off, all of them are off, none of them are off or combination of some off and some on. This potentiality makes reasoning more complex and some cases impossible due to infinite number of possibilities, one leading to another. How we deal with this problem is through *default reasoning*.

1.1.2 Default Reasoning

When performing commonsense reasoning, it is rarely the case that we have complete information about the state of the world. Therefore we need to make certain assumptions and make inferences based on them to proceed. That is why commonsense reasoning requires default reasoning. Commonsense reasoning is based on actions and their effects on the state of the world. For this reason, when performing commonsense reasoning we need to have the default (commonsense) assumption that:

- Unexpected events do not occur; and
- Events do not have unexpected effects

This way we could make inferences based on incomplete information (which we are likely to have in most cases) and proceed. We will give a detailed explanation of using *circumscription* to implement default reasoning on a machine.

1.1.3 Space

Many instances of commonsense reasoning involve space. In this section we briefly describe space axiomatisation in our commonsense representation.

Space: In the commonsense world, objects stand in various relations to each other or to a base point. For instance, a pencil is in a jar, a person is in a room, two objects moving with varied velocities might collide and so on. In discussing commonsense, we need to take these relations into account. In one of the scenarios presented at a later chapter, we use the spatial theories about space in the commonsense world of Region Connection Calculus (RCC) by Randell, Cui and Cohn [12]. In their theory, the ontological primitives include physical objects, regions and sets of entities. For example, basic relations such as P(x, y) 'x is part of y', C(x, y) 'x is connected with y', PO(x, y) 'x partially overlaps y' and composite relations such as INSIDE(x, y) 'x is inside y', P-INSIDE(x, y) 'x is partially inside y' and OUTSIDE(x, y) 'x is outside y' resemble (partially) space domain of commonsense. It is easy to see that these formalisms represent concepts of commonsense. As an example, if two distinct objects which are not disjoint initially and over time they become connected, then those objects collide (the commonsense fact is clear in this theory that it is odd that two regions can be distinct but occupy the same amount of space).

1.1.4 Analogical Processing

In analogy, a given situation is understood by comparison with another similar situation. Analogy could be used to guide reasoning, to generate conjectures about an unfamiliar domain, or to generalise several experiences into an abstract schema.

Analogical Processing: It deals with novel situations that an agent might encounter and has no direct commonsense knowledge of; in such a case, the agent might be able to reason about the novel situation by matching the analogy to a familiar situation. Consider the following example: If an agent puts a stopper in place in the sink and opens the tap, the water will eventually start spilling onto the floor because it reaches and goes over the rim of the sink (example from Shanahan [48, 56 pp. 302-304]). An agent might have encountered and reasoned about the previous situation before, but a novel situation could be formed by replacing water by sand. Using analogical processing, and in particular a well-developed mechanism called Structure-Mapping Engine (SME) this comparison is possible. For details of SME please refer to [25] as we will not discuss it here. We would like, however, to present an example from [25] to show how SME can be *partially* used to solve commonsense reasoning problems:

The comparison between water flow and heat flow is represented as follows:

The base (already encountered) domain:

Causes(GreaterThan(Pressure(Beaker), Pressure(Vial)).

Flow(Beaker, Vial, Water, Pipe)).

GreaterThan(Diameter(Beaker), Diameter(Vial)).

Liquid(Water).

FlatTop(Water).

The above reads: when the pressure of the beaker is greater than the pressure of the vial it causes the flow of water from the beaker to the vial through the pipe; the diameter of the beaker is greater than that of the vial in general; water is liquid and it has a flat top.

The target (novel) domain:

GreaterThan(Temperature(Coffee), Temperature(IceCube)).

Flow(Coffee, IceCube, Heat, Bar).

Liquid(Coffee).

FlatTop(Coffee).

It reads that the temperature of coffee is greater than ice cube; heat can flow from coffee to the ice cube through the bar; coffee is liquid and it has a flat top.

The SME can then produce global mappings and choose the one with the highest score:

Beaker # Coffee.

Vial # IceCube.

Water # Heat.

Pipe # Bar.

Pressure(Beaker) # Temperature(Coffee).

Pressure(Vial) # Temperature(IceCube).

GreaterThan(Pressure(Beaker), Pressure(Vial)) #

GreaterThan(Temperature(Coffee), Temperature(IceCube)).

Flow(Beaker, Vial, Water, Pipe) # Flow(Coffee, IceCube, Heat, Bar).

And the following candidate inference is produced:

Causes(GreaterThan(Temperature(Coffee), Temperature(IceCube)),

Flow(Coffee, IceCube, Heat, Bar)).

SME is only a partial solution for commonsense reasoning. Other commonsense reasoning mechanisms are required to evaluate and draw inferences; SME can be used to produce *potential* inference candidates, as a complement method.

1.2 The Cyc project

Doug Lenat, founder of the Cyc project explains: "The purpose of Cyc is to provide computers with a store of formally represented 'common sense': real world knowledge that can provide a basis for additional knowledge to be gathered and interpreted automatically" [46].

Cyc project began in 1984 and has been evolving and gathering commonsense knowledge from various sources for more than 26 years to this day. Cyc states its long term goal as: "automating the process of building a consistent formalised representation of the world in Cyc knowledge bases on machine learning" [47]. Cyc, however, uses logic for commonsense reasoning [7, 17]. Cyc's knowledge base is quite reliable due to the methods that it uses before any new knowledge is added to its knowledgebase. We briefly describe the methods here.

The Cyc ontologists asserted some basic facts into Cyc in the early days in order to enable Cyc to crawl the web and gather new information based on what it already knows [7]. With the knowledge in the knowledgebase, Cyc decides to search for an "interesting" subject using the following algorithm (the topic is either picked from the knowledge base or set by a human expert and the search is run on the Internet via Google) [47]:

For a given search run, a depth of *D* is selected. *D* is the maximum number of different values that can be used for each argument of a predicate. For each binary predicate p_i in the test set *P*, the types of constraint in each of the two arguments are retrieved from the knowledge base. The *D* most fully represented values from the knowledge base are retrieved unless the type is generalised to an infinite class. The *D* fully represented values means those that appear in the most assertions and therefore about which the most is known. These are assumed to be the most "interesting" terms of that type and the ones most likely to be found by a web search. There are the types T^{i1} and T^{i2} for p_i. The *D* best represented values are $(t^{i1}_1 \ldots t^{i1}_D)$ and $(t^{i2}_1 \ldots t^{i2}_D)$.

If neither of a predicate's arguments are of values of a continuous type, there will be $2D*|P|$ queries generated (in CycL):

$(p_1\ t^{11}_1\ ?VAR) \ldots (p_1\ t^{11}_D\ ?VAR)$

$(p_1\ ?VAR\ t121\ ) \ldots (p_1\ ?VAR\ t^{12}_D)$

…

$(p_{|P|}\ t^{|P|1}_1\ ?VAR) \ldots (p_{|P|}\ t^{|P|1}_D\ ?VAR)$

$(p_{|P|}\ ?VAR\ t^{|P|2}_1\ ) \ldots (p_{|P|}\ ?VAR\ t^{|P|2}_D)$

The limit of binary predicates (*p*) is set by the ontologists. The number of *p* is currently 134 predicates.

When Cyc *decides* to learn about a new fact, it poses a query. Then the query is translated into natural language from CycL (Cyc's representational language) and is searched on the web via Google. This translation is done by 233 manually created special generation templates for the 134 predicates. Cyc knowledgebase generally contains one or two generation templates for any given predicate. For instance, Cyc might pose the queries:

“Microsoft company founder _____”

“MS company founder _____”

“Microsoft company founded by _____”

“MS company founded by _____”

Whose aims are to find out the founder of Microsoft.

The results of the query are then translated back into CycL. Since Cyc uses predefined templates for generating natural language queries, it also expects the results to be in the same predefined forms. For instance, when Cyc poses the above queries, it expects the results to be in the form of:

“Microsoft founder Bill Gates is still running the company.”. Depending on where the position of ____ was in the initial query, Cyc will assume that position to fill in the predicate argument (Bill Gates in this case).

Then Cyc checks the result for inconsistency or redundancy against the already known (and supposedly correct) knowledgebase. Cyc discards redundant or inconsistent search results; then rechecks the remaining results in Google by adding a word (from the same concept) to the query. If there are no results returned then the “fact” is discarded otherwise the fact is sent to an ontologist to review and insert into the knowledgebase if correct.

Using this method and the fact that a human expert will review the acquired fact before assertion, it is guaranteed that the asserted knowledge in the knowledgebase are (at least to a human-expert level) correct.

CycL

CycL's syntax is based on the syntax of First-order Logic and Lisp [45]. CycL handles all of FOL connectives such as *and*, *or*, *implies* and also *quantifiers*. CycL also handles default reasoning which makes a suitable language to deal with commonsense [24, Sec. 1]. It has five different truth values for statements (fluents) which are: *monotonically false*, *default false*, *unknown*, *default true* and *monotonically true*. As a regularly updated knowledgebase, the default values can be overridden. For instance a statement such as "Dogs have four legs" in standard truth-conditional logic would raise inconsistency since there are dogs who have three legs as well. The approach of different truth values taken by Cyc helps to prevent inconsistency and also makes the ontology more robust (by solving the problem that three-legged dogs, for instance, are still "dogs" which are still "mammals" and so on). Cyc also uses "microtheories" [24] which are small and dynamically generated concepts depending on implicit context of reference. For instance, the following conversation would raise inconsistency by using standard truth-conditional logic:

CHILD: Who is Dracula, Dad?

FATHER: A vampire.

CHILD: Are there really vampires?

FATHER: No, vampires don't exist.

Of course, is as discussed in [24], the father's answer to the first question is in *context* of mythology and fiction but the answer to the second question is in context of the real world. Assertions in a microtheory must be consistent with each other (*local consistency*) but do not need to be consistent with other

microtheories. This way, global consistency is assured while local consistencies can exist in different microtheories.

1.3 Aims of The Project

The principal aim of this thesis is to

1. *show and emphasise on the Event Calculus to be a robust and flexible formalisation to deal with commonsense reasoning*

This will be complemented by a series of further aims:

2. *explain the event calculus in detail*
3. *introduce circumscription to deal with default reasoning*
4. *compare the Event Calculus with PDDL+ in practice*
5. *introduce a method for converting the Event Calculus formulas into propositional logic to be reasoned over by a SAT solver*
6. *introduce automated reasoners that deal with the Event Calculus*

In this chapter we talked about commonsense reasoning in general and some of its specifications and reasoning types. We also talked about the Cyc project, a good example of a systematic approach towards commonsense reasoning using a logical formalisation. In the rest of this report:

- To achieve our second aim we will introduce the Event Calculus, a formalisation that deals with commonsense reasoning in detail. This will be complemented with examples to better emphasise the features of the Event Calculus (EC).
- For achieving the third aim, we will introduce circumscription which will be our mathematical machinery to implement default reasoning. This will

be followed by an example in EC to show the use of circumscription in the EC in practice.

- For satisfying the fourth aim, we will present a framework which we have constructed in the Event Calculus with a real world scenario. Our framework is an abstract simulation of a real world model in which we have tried to capture some commonsense facts and rules. The initial idea of this model was based on Shin and Davis formalisation of a similar (but much more abstract) world. We prove a proposition in the scenario in which an agent has a goal in mind and sets off to perform some actions to achieve the goal. We show that by using the EC formalisation and modelling of the world we could indeed prove the proposition. A flagging system is introduced which prevents repeated occurences of an action while all its conditions hold. We compare our formalism in EC with that of Shin and Davis and mention the advantages and shortcomings. Then we analyse and talk about the importance of determining the level of details we need to focus on to solve a commonsense problem.
- To achieve our fifth aim, we present an encoding method to transform EC formulas into a satisfiability problem so that we could automatically perform reasoning on a commonsense problem by sending the then-propositional problem to a SAT solver. The results will be converted back to EC formulas by using reverse mapping.
- To satisfy our sixth aim, we mention automatic reasoning methods that deal with the EC in the literature and describe how a SAT solver works in greater detail.

- We then draw conclusions on this report and the achievements of it and talk about the future work and the many great possibilities that the Event Calculus provides to handle commonsense problems. By this point, we will have achieved our principal aim which is to show and emphasise on the Event Calculus to be a robust and flexible formalisation to deal with commonsense reasoning.

One note to take into account here is that although we give many EC examples in this report in different sections, we do not perform the propositional encoding on our framework. The reason for this is simply because this is not an aim of the project. Using a SAT solver for automating the proof is a different piece of work which we did not intend to achieve (and compare performances of different solvers and reasoners). This could be a follow-up piece of work but not in the intention or scope of this project.

Chapter 2

The Event Calculus

In this chapter we introduce the event calculus; the standard representational language of representing and reasoning with commonsense.

2.1 History

The event calculus (EC) uses the syntax of First-order Logic. Mueller [14, pp. 271-289] investigates the evolution of different logical approaches in commonsense reasoning which started from the introduction of the Situation Calculus by John McCarthy [27] and McCarthy and Hayes [28].

The Event Calculus was developed by Kowalski and Sergot [50]. A lot of research and work has gone into the Event Calculus and new extensions have been appended hence it has evolved enormously over time. The extensions to the EC have brought the advantages of the other logics such as the Forced Separation of Features and Fluents introduced by Murray Shanahan [56, Ch. 16], Causal constraints of Fluent Calculus and Continuous Change to make the Event Calculus the most robust and the standard logic for commonsense reasoning.

Figure 2.1 shows a brief history of evolution of the Event Calculus. This figure does not mean Situation Calculus is out of use today – it simply shows how Event Calculus has evolved to where it is now and the properties it has borrowed from other formalisations.

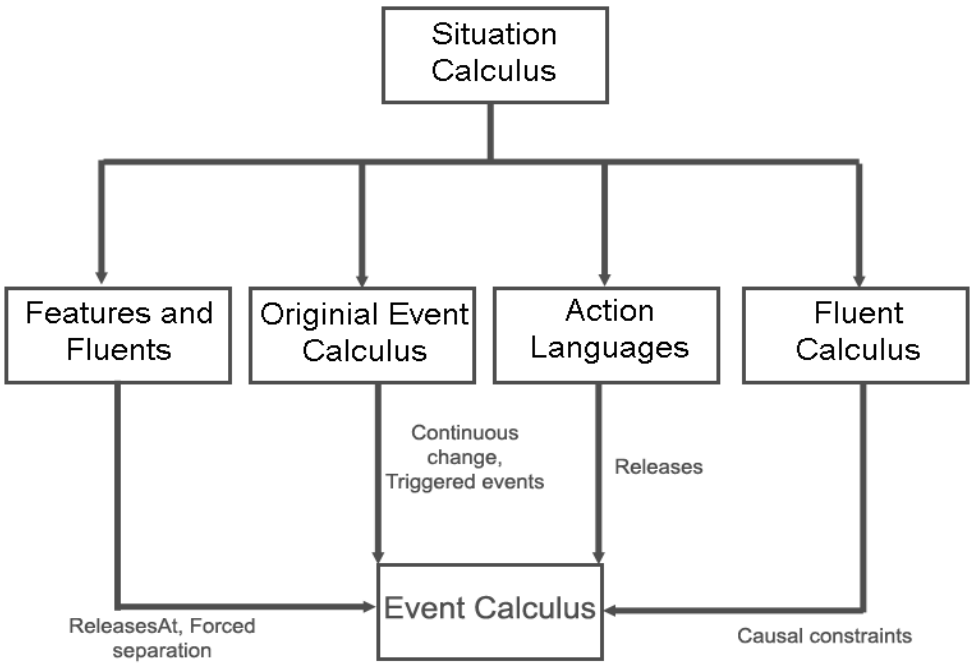

Figure 2.1: Logics for Commonsense Reasoning

2.2 Formal definition of the Event Calculus

2.2.1 Introduction

The event calculus is a narrative-based formalism for reasoning about action [5]. Shanahan [56, p155] defines a narrative as: "A distinguished course of events about which we may have incomplete information". The event calculus addresses all of the key issues of commonsense reasoning for which we will give more details later in this chapter. EC is the language of action and change and therefore has three basic notions as follows:

- Events: which represent actions or events that may occur in the world such as breaking a glass. An event may *happen* at a *timepoint*.
- Fluents: which represent time-varying properties in the world such as location of a glass. A fluent has a truth value at any given *timepoint* or a *timepoint interval*.
- Timepoints: which represent an instant of time such as 8:00 AM Greenwich Mean Time on September 20 2007.

After an event occurs (or in EC terms: *happens*), the truth values of fluents may change. We have commonsense background knowledge about the effects of events on fluents such as dropping an object results in the object falling.

We represent the basic notions in the Event Calculus as follows:

HoldsAt(f, t): means fluent 'f' holds at timepoint 't'.

Happens(e, t): means event 'e' occurs at timepoint 't'.

Initiates(e, f, t): represents that if event 'e' happens at timepoint 't', then fluent 'f' will be true after 't'.

Terminates(e, f, t): represents that if event ‘e’ happens at timepoint ‘t’, then fluent ‘f’ will be false after ‘t’.

For instance, the following is an EC representation of if somebody wakes up then they will be awake and not sleeping anymore. Also that John is initially not awake but then he wakes up:

Our EC axioms (commonsense or background knowledge):

Ex1.1: Initiates(WakeUp(a), Awake(a), t)

Ex1.2: Terminates(WakeUp(a), Asleep(a), t)

Initial situation (observations):

Ex1.3: ¬HoldsAt(Awake(John), 0)

Event occurrences (narrative):

Ex1.4: Happens(WakeUp(John),1)

In the above example, from Ex1.4 and Ex1.1 we can conclude that John is awake after 0 or in other words *HoldsAt(Awake(John), 1)*. Since as humans we have background knowledge, we can deduce this conclusion easily. But formally, we cannot make this conclusion yet; we are missing some information.

In the event calculus, there are several basic axioms and definitions which make reasoning on EC formulas possible. These axioms and definitions are available in Appendix A for the Event Calculus (EC) and Appendix B for Discrete Event Calculus (DEC - which we will discuss in section 2.2.3). We have borrowed the axioms of the event calculus from Mueller [14, pp. 24-29] who represents them in

a neat way (he has originally taken them from Miller and Shanahan [51, 52]). Discrete Event Calculus axioms, however, are from Mueller [14].

Using the EC and DEC axioms, we can prove our example. We need to assert the following axiom in order to be able to draw conclusions:

Ex1.5: ¬ReleasedAt(f, t)

Axiom Ex1.5 tells us that no fluent 'f' is released from the commonsense law of Inertia at any timepoint. Although this axiom has not been defined in the report yet, we will discuss it in section 2.2.2. Let us accept it for the sake of argument, for now.

Now with the conjunction of axioms Ex1.1 to Ex1.5 and DEC axioms (in Appendix B) we can systematically draw the following conclusions:

From Ex1.4 and Ex1.1 and DEC9 we have *HoldsAt(Awake(John), 2)*. This says that John is awake at timepoint 2 (as the result of waking up at timepoint 1).

From Ex1.4 and Ex1.2 and DEC10 we can conclude ¬HoldsAt(Asleep(John), 2) which says that John is not asleep at timepoint 2.

However, we can also represent the opposite of these axioms: if somebody sleeps, then they will be asleep and not awake anymore. To represent this, we need to insert the following axioms:

Ex1.6: Initiates(Sleep(a), Asleep(a), t)

Ex1.7: Terminates(Sleep(a), Awake(a), t)

And suppose we have the following narrative:

Ex1.8: Happens(Sleep(John), 3)

Similarly, we can show that from Ex1.6, Ex1.8 and DEC9 we have HoldsAt(Asleep(John), 4) and from Ex1.7, Ex1.8 and DEX10 we have ¬HoldAt(Awake(John), 4).

This works. However, there seems to be redundancy between axioms Ex1.1, Ex1.2, Ex1.6 and Ex1.7: There is no indication that these actions or fluents are exactly the opposite of each other. This must be represented in our formalism to simulate the commonsense knowledge that if something is on, then it is not off or if someone is awake, then they are not asleep. The event calculus is a very strong and flexible representational language. We can show this by using State Constraints:

HoldsAt(Asleep(a), t) $\Leftrightarrow \neg$HoldsAt(Awake(a), t)

The above axiom says that somebody is asleep if and only if they are not awake. We shall discuss State Constraints and some other features of the event calculus more in detail in the following section.

2.2.2 Features

We briefly discuss some of these features of the event calculus as follows:

Elaboration Tolerance: The event calculus is elaboration tolerant. That means it allows for an axiomatisation to be extended through the addition of new axioms rather than performing surgery on existing axioms. This is particularly important for a representation of commonsense since new information is constantly gathered through time. Elaboration tolerance increases the efficiency and reduces inconsistency in a knowledge base. For instance if we have the following axiom:

Initiates(MoveInside(agent), AlarmOn(alarm), t).

If an agent moves inside, then the alarm will be triggered. If we later find out that the agent we are referring to must be either one of Fred, John or Richard, we can add an axiom to the formula without changing the existing ones:

Happens(MoveInside(a), t) $\Rightarrow$ HoldsAt(At(Fred, location), t) $\vee$ HoldsAt(At(John, location), t) $\vee$ HoldsAt(At(Richard, location), t)

Reification: McCarthy [54, p 1034] defines reification as "making objects out of sentences and other entities". He introduces reification [55] in order to reason about knowledge and belief in First-order Logic. He introduces [p. 129] terms to represent concepts such as "Mike's telephone number" in the sentence "Pat knows Mike's telephone number". Using this technique we would be able to represent this sentence as follows: *Mike* represents *Mike* in the sentence, Telephone(Mike) represent's *Mike's telephone number*, and Know(Pat, Telephone(Mike)) represents the whole sentence. The event calculus is based on reification.

Default Reasoning: Default reasoning is handled in the event calculus by using circumscription as technical machinery. A detailed description of circumscription is presented in the next section. As described in Chapter 1, default reasoning is necessary when reasoning about commonsense to cope with the incomplete information we have about our scenario and state of the world. Default reasoning makes it possible to only consider the events that we know do happen and ignore all the other possibilities; and that the state of the world does not change unless some event happens to change it. We also take it that the events only have their intended effects. For instance, when Fred turns on the light, it does not cause the door to open, or when an object is left on the table, it will not suddenly disappear and will be there until something happens to it and moves it. Or if our scenario is

that Fred goes to the kitchen we should not assume that the TV is turned off. Circumscription does not allow this to happen.

Effect Axioms: Represent the effect of events on fluents. The effect axioms, namely *Initiates* and *Terminates* have already been discussed in the previous section.

Preconditions: In the Event Calculus, it is possible to make certain actions or fluents conditional on some other actions or fluents. There are two types of preconditions in the event calculus, Fluent Precondition and Action Precondition.

- **Fluent Precondition** must be satisfied for an event to have an effect. If a fluent precondition is not satisfied, then the event *may occur* but it will not have the intended effects. For instance, if a device is not broken, then it can be turned on:

  $$\neg \text{HoldsAt}(\text{Broken}(d), t) \Rightarrow \text{Initiates}(\text{TurnOn}(a, d), \text{On}(d), t)$$

  In this example, if *d* is broken, the event *TurnOn* might happen, but since *Initiates* is conditioned on *d* not being broken, *On(d)* will not change because of this event.

- **Action Precondition** must be satisfied for an event to occur. If an action precondition is not satisfied then the event *cannot occur*. For example, to pick up a book one must be near the book.

  $$\text{Happens}(\text{PickUp}(a, \text{Book}), t) \Rightarrow \text{HoldsAt}(\text{Near}(a, \text{Book}), t)$$

  (By contraposition, $\text{Happens}(\alpha, \tau) \Rightarrow \gamma$ is equivalent to $\neg\gamma \Rightarrow$

  $\neg\text{Happens}(\alpha, \tau)$)

Usage of preconditions is a partial solution to the *Qualification Problem* which will be discussed shortly.

State Constraints: Some properties of the commonsense world work in a law-like fashion. For instance the fact that an object cannot be on top of itself (*¬HoldsAt(On(o,o), t)*) or as already seen in the previous section someone cannot be awake and asleep at the same time (*HoldsAt(Asleep(a), t)* $\Rightarrow$ *¬HoldsAt(Awake(a), t)*) or a device could not be on and off at the same time (*HoldsAt(On(d), t)* $\Rightarrow$ *¬HoldsAt(Off(d), t)*). State constraints are important axioms describing some of our world and domain-specific knowledge.

Qualification Problem [8]: A condition that prevents an event from having its intended effects or prevents the event from occurring is called a *qualification* and the problem of representing and reasoning about a *qualification* is called the qualification problem. With the help of *preconditions*, *state constraints* and *default reasoning* the qualification problem can be solved. For instance, the example of a broken device that does not turn on or a "not broken" device can be turned on is a qualification problem:

¬HoldsAt(Broken(d), t) $\Rightarrow$ Initiates(TurnOn(a, d), On(d), t)

And since the event calculus is elaboration tolerant, we could assert the following axiom at a later time:

HoldsAt(CutWire(d), t) $\Rightarrow$ HoldsAt(Broken(d), t)

HoldsAt(Broken(b), t) $\Rightarrow$ HoldsAt(Broken(d), t)

(*b* stands for the button of a device).

Trigger Axiom: It is possible to assert *events* or *fluents* in the EC which are triggered by other axioms. This feature is a basic and fundamental concept in commonsense reasoning. For instance, if an agent puts a stopper in the drain of the sink and opens the tap (in case there is water flowing from the tap; the sink has no cracks or holes; and so on, i.e. qualification problem is dealt with), the water will eventually start spilling onto the floor because it reaches and goes over the rim of the sink (example from Shanahan [48, 56 pp. 302-304]). Another example is an alarm clock. The clock will start beeping once the present moment is the set alarm time (example from [14 pp. 75-78] which contains axioms and proof of the proposition).

Indirect Effects of Events: The event calculus can be used to represent and reason with indirect effect of events or *ramification*. An example of indirect effects of actions is moving from one room to another while holding an object (which results in the change of location of the object, we will discuss this example in more detail toward the end of the chapter). A detailed discussion of ramification is presented in [14 pp. 101-130].

Commonsense Law of Inertia: As the law states, objects tend to stay in the same state unless affected by events. The event calculus makes use of the *Releases(e, f, t)* and *ReleasedAt(f, t)* predicates to indicate, respectively: an action releases a fluent from this law at a time; a fluent is released from this law at a time. *Initiates* and *Terminates* predicates, if used, will restore the law for a fluent. A practical example of this law is the Yale Shooting Scenario of Hanks and McDermott [59,

60, 61 pp. 387-390]. EC works in timepoints and each timepoint is separate and different from the others. Existing fluents are only transferred from a timepoint to the next *if and only if* they are not released. For instance, given we have some other axioms which describe the height of the object decreasing over time, with *HoldsAt(At(Object, Table), 1)*, if nothing happens to terminate or release this fluent, by DEC5 we can conclude that *HoldsAt(At(Object, Table), 2)*. However, if we have that *Releases(e, At(Object, Table), 2)* or *Terminates(e, At(Object, Table), 1)* then we cannot make this conclusion anymore.

ReleasedAt fluents are very important and useful specially when a fluent has an integer value and is changing over time e.g.: the height of a falling object. We need EC to discard the previous height of the object at each timepoint so that an object will only have one height at a time. For instance consider the following example:

(*The object falls at timepoint 1.*)

Happens(Fall(Object), 1)

(*The object has the height 50 at timepoint 2.*)

HoldsAt(Height(Object, 50), 2)

(*The event of Fall initiates the fluent Falling.*)

Initiates(Fall(Object), Falling(Object), t)

(*The event DecreaseHeight initiates the fluent Height with the value x-1.*)

Initiates(DecreaseHeight(Object, x), Height(Object, x-1), t)

(*If the object is falling, then decrease its height.*)

HoldsAt(Falling(Object), t) $\wedge$ HoldsAt(Height(Object, x) $\Rightarrow$

Happens(DecreaseHeight(Object, x), t)

From *Happens(Fall(Object), 1)* and *Initiates(Fall(Object), Falling(Object), t)* and DEC9 we have:

HoldsAt(Falling(Object), 2)

From this, *HoldsAt(Height(Object, 50), 2)* and *HoldsAt(Falling(Object), t)* $\wedge$ *HoldsAt(Height(Object, x)* $\Rightarrow$ *Happens(DecreaseHeight(Object, x), t)* we have:

Happens(DecreaseHeight(Object, 50), 2)

From this, *Initiates(DecreaseHeight(Object, x), Height(Object, x-1), t)* and DEC9 we have:

HoldsAt(Height(Object, 49), 3)

Here is where a problem potentially lies: not enforcing the commonsense law of Inertia – nothing happens to terminate *HoldsAt(Height(Object, 50), 3)* which means the height of the object is both 49 and 50 at the timepoint 3. And as timepoints increase, so does the presence of the previous values of the height of the object. We need to have a rule enforcing Intertia law to avoid having multiple values for the same fluent at the same timepoint. We need to have either of these:

- A *Terminates* fluent (which by DEC12 enforces this law) such as *Terminates(DecreaseHeight(Object, x), Height(Object, x), t).*
- A fluent such as *ReleasedAt(Height(Object, x), t)* that says the height of the object is submissive towards the Inertia law – using this predicate the

previous values of the height of the object do not hang on at the timepoints that follow. This is because DEC5 and DEC6 need a fluent not to be released [¬*ReleasedAt(f, t)*] to maintain its value at the next timepoint.

Therefore if we do not have one of the above remedies, using DEC5 we can conclude that:

Timepoint 3:

HoldsAt(Height(Object, 50), 3).

HoldsAt(Height(Object, 49), 3).

Timepoint 4:

HoldsAt(Height(Object, 50), 4).

HoldsAt(Height(Object, 49), 4).

HoldsAt(Height(Object, 48), 4).

Timepoint 5:

HoldsAt(Height(Object, 50), 5).

HoldsAt(Height(Object, 49), 5).

HoldsAt(Height(Object, 48), 5).

HoldsAt(Height(Object, 47), 5).

And so on. Commonsense law of Inertia (through *Initiates*, *Terminates*, *Releases* and *ReleasedAt* predicates) prevent this from happening.

Nondeterministic Effects: An event has nondeterministic effects if the event can have more than one alternative effect. For instance, flipping a coin could result in the coin landing with head or tail. The event calculus deals with nondeterminism by allowing event occurrences to give rise to several classes of models using *determining fluents* [56 pp. 294-297, 72 pp. 419-420] or *disjunctive event axioms*

[56 pp. 297-298, 342-345 and 359-361]. Disjunctive event axioms are represented in the Event Calculus as:

$$\text{Happens}(\alpha, \tau) \Rightarrow \text{Happens}(\alpha_1, t) \vee \ldots \vee \text{Happens}(\alpha_n, t)$$

2.2.3 Discrete Event Calculus

The Discrete Event Calculus is the same as the Event Calculus with the only difference that timepoint sort in DEC is restricted to positive integers. DEC was developed by Erik E. Mueller [14 p. 27] and as Mueller has proven [14, Ch. B] if time is restricted to integers, then DEC is equivalent to EC axiomatisation of Miller and Shanahan [52]. We have presented the basic DEC axioms in Appendix B.

We have not used *Trajectory* or *AntiTrajectory* axioms in this report. These axioms, however, are defined in Appendices A and B. They deal with gradual change of fluents over time. Their usage is no more complex than the other axioms of EC and the reason we have not mentioned them is due to the space constraint of this report. The only occurrence of these axioms is in an example in Chapter 1 regarding gradual change of height of a falling object.

2.3 Formal definition of circumscription

The main reason for using circumscription (and non-monotonic reasoning in general) comes from the theory of knowledge representation. Axiomatic theory of classical logic cannot directly represent defaults. And since default reasoning is

an essential key factor of commonsense reasoning, circumscription is used in commonsense representation and reasoning. Circumscription, introduced by John McCarthy [42], is the technical device to implement default reasoning.

Before we formally define representation, we need to explain some basic notions. We start introducing circumscription by an example from Lifschitz (90). Suppose we have a default rule that:

CE1.1 "Normally, a block is on the table".

And we have the assertion:

CE1.2 "B1 is not on the table".

Since nothing is mentioned about *B2* (which is also a block), we want to conclude that *B2* therefore must be on the table:

CE1.3 "B2 is on the table".

Assertion CE1.1 and CE1.2 can be mathematically represented as:

CE1.4 $\text{Block}(x) \wedge \neg \text{Ab}(x) \Rightarrow \text{OnTable}(x)$

CE1.5 $\neg \text{OnTable}(B1)$

(Predicate $\neg Ab(x)$ means that *x* is not abnormal. So CE1.4 means: "if *x* is a block and *x* is not abnormal, then *x* is on the table".)

We also have:

CE1.6 $\text{Block}(B1) \wedge \text{Block}(B2) \wedge B1 \neq B2$

Our goal would be to conclude that *B2* is on the table (formalisation of CE1.3):

CE1.7 $\text{OnTable}(B2)$

We cannot draw this conclusion by classical logic only, since CE1.7 is not a consequence of axioms CE1.4, CE1.5 and CE1.6. Axioms CE1- CE1.6 say too little about the abnormality predicate *Ab*. So, for instance, we could find models

in which the universe of *M* consists of two objects, represented by constants *B1* and *B2*. The predicates *Block* and *Ab* are true for *B1* and *B2* and *OnTable* is false for *B1* and *B2*. In this model, CE1.4- CE1.6 are true but CE1.7 is false. The problem is, axiom CE1.4 is the only axiom that mentions *Ab* and yet it does not say whether there are few or many abnormal objects. Therefore there exists an identical model M' to *M* with the only difference that the extension of *Ab* is *{B1}* rather than *{B1, B2}*, which was the case for model *M*. Therefore the extension of *Ab* in M' is a proper subset of the extension of *Ab* in *M*. And since *M* and M' only differ on the extension of *Ab*, we therefore can say that M' is minimal to *M* with respect to the predicate *Ab*:

We now formally introduce the concept of minimality. But before that, we need to know what it means for a model to be as small as another model.

Formal definition of *as small as*: If M' and *M* are interpretations, then M' is as small as *M* with respect to a predicate *P* (written as $M' \sqsubseteq_P M$) if:

- M' and *M* agree on the interpretation of everything except possibly *P* and
- The extension of *P* in M' is a subset of its extension in *M*.

Formal definition of minimality: A model M' of a formula ϕ is minimal with respect to $\sqsubseteq_P$ if there is no model *M* of ϕ such that $M \sqsubseteq_P M'$ and not $M' \sqsubseteq_P M$.

This is very similar to our concept of circumscription which we will explain in the next section. We therefore need to verify that (**Theorem 0**): A model M' of a formula ϕ is a model of the circumscription of ϕ minimising the predicate *P* if and only if M' is minimal with respect to $\sqsubseteq_P$.

Proof: Since the theorem uses if and only if, we use two half-proofs for each if (*if* and *only if*). First, the *if* part: We use contradiction. Suppose M' is a minimal model of ϕ with respect to $\sqsubseteq_P$ but is not a model of circumscription. Then there must be some q such that $\phi(q) \wedge q < P$ is satisfied in M' (extension of q in M' is smaller than the extension of P in M'). We can then construct a model M which is identical to M' except that the interpretation of P in M is the same as the interpretation of q in M' ($M[P] = M'[q]$). Clearly $M \sqsubseteq_P M'$ however it is not the case that $M' \sqsubseteq_P M$. Therefore M' is not minimal which is a contradiction.

Now the *only if* part: We use contradiction again. Suppose M' is a model of circumscription of ϕ minimising P (written as $CIRC[\phi; P]$) but M' is not minimal with respect to $\sqsubseteq_P$. Then there must be a model M of ϕ such that $M[P] \subset M'[P]$. In this case, since it is possible to let $M'[q] = M[P]$, M' does not satisfy $\neg\exists q[\phi(q) \wedge q < P]$ and therefore is not a model of the circumscription which is a contradiction. ($\neg\exists q[\phi(q) \wedge q < P]$ is the formal definition of circumscription, we will introduce it later).

Back to our example, since M' is the minimal model with respect to $\sqsubseteq_{Ab}$, we have the following condition:

CE1.8 $\quad Ab(x) \Leftrightarrow x = B1$

Formal Definition and Implementation of Circumscription

Circumscription is concerned with the extension of predicates in models. We need to introduce some notations in order to actually formally implement circumscription:

Let P and Q be n-ary predicate symbols and v_1 to v_n distinct variables of appropriate sort. We have the following relations:

$P = Q$ is an abbreviation for $\forall v_1, \ldots, v_n\, P(v_1, \ldots, v_n) \Leftrightarrow Q(v_1, \ldots, v_n)$.

$P \leq Q$ is an abbreviation for $\forall v_1, \ldots, v_n\, P(v_1, \ldots, v_n) \Rightarrow Q(v_1, \ldots, v_n)$.

$P < Q$ is an abbreviation for $(P \leq Q) \wedge \neg(P = Q)$.

Formal definition of circumscription: If Φ is a formula containing the predicate symbol ρ, then the circumscription of Φ minimising ρ, written as $CIRC[\Phi; \rho]$, is the formula of second-order logic:

$\Phi \wedge \neg\exists q[\Phi(q) \wedge q<\rho]$

where q is a predicate variable with the same arity and argument sorts as ρ, and $\Phi(q)$ is the formula obtained from Φ by replacing each occurrence of ρ with q.

This says in every model of $CIRC[\Phi; \rho]$, the extension of ρ complies with Φ and there is no proper subset of the extension of ρ that complies with Φ. That is, the extension of ρ is minimal given Φ.

An example: Suppose $\Phi = P(A)$. Then $CIRC[\Phi; P]$ is given by the second-order formula:

$P(A) \wedge \neg\exists q[q(A) \wedge q < P)$

This means for every model M of $CIRC[\Phi; P]$, if A is the object named by A in M, then $A \in P^M$ and there is no proper subset Q of P^M such that $A \in Q$, from which

we conclude that $P^M = \{A\}$. Therefore, $CIRC[\Phi; P]$ is equivalent to the first-order formula

$\forall x(P(x) \Leftrightarrow x = A)$

Non-monotonicity of circumscription: Classical logic is monotonic. This means if, for instance, a sentence q follows from a collection of A sentences and $A \subset B$, then q also follows from B $(A \vdash q \wedge A \subset B \Rightarrow B \vdash q)$. A proof from the premises A is a sequence of sentences each of which is either a premise, an axiom or follows from a subset of sentences occurring earlier in the proof by one of the rules of inference. Therefore a proof from A can also serve as a proof from B. We know that the semantic notation of entailment in classical logic is monotonic. For instance A entails q $(A \models q)$ if q is true in all models of A. And if $A \models q$ and $A \subset B$, then all the models of A are also models of B which means $B \models q$ too. In other words, we can show monotonicity of classical logic with the notation:

$A \vdash q \Leftrightarrow Conjunction[A] \models q$

in which $Conjunction[A]$ is the conjunction of all sentences in G.

The formal definition of circumscription states that the circumscription of a formula is a sentence of second-order logic. The consequence relation of second-order logic is classical and monotonic. So how is circumscription non-monotonic? Circumscription preserves monotonicity to the extent that it makes use of classical consequence relation. But it is non-monotonic in the sense that, given a predicate P, it does not guarantee that for any conjunction of formulas Ψ and any two formulas σ and ϕ,

$CIRC[\Psi; P] \models \phi \Rightarrow CIRC[\Psi \wedge \sigma; P] \models \phi$. The non-monotonicity of a circumscription minimising a predicate P can be shown with the notation:

$\Psi \vdash \phi \Leftrightarrow CIRC[\Psi; P] \vDash \phi$.

John McCarthy [42] himself remarks:

"Circumscription is not a non-monotonic logic. It is a form if non-monotonic reasoning augmenting ordinary first-order logic."

Parallel circumscription: Circumscription also allows for parallel minimisation of predicates. If Φ is a formula containing the predicate symbols $\rho_1, \ldots, \rho_n$, the then parallel circumscription of Φ minimising the predicates $\rho_1, \ldots, \rho_n$, written as $CIRC[\Phi; \rho_1, \ldots, \rho_n]$, is the formula of second-order logic:

$$\Phi \wedge \neg\exists q_1, \ldots, q_n [\Phi(q_1, \ldots, q_n) \wedge \bigwedge_{i=1}^{n} q_i < \rho_i$$

where $q_1, \ldots, q_n$ are distinct predicate variables with the same arities and argument sorts as $\rho_1, \ldots, \rho_n$ respectively and $\Phi(q_1, \ldots, q_n)$ is the formula obtained from Φ by replacing each occurrence of $\rho_1, \ldots, \rho_n$ with $q_1, \ldots, q_n$ respectively.

Let us consider an example:

We have the following knowledgebase:

CE2.1 Initiates(SwitchOn, DeviceOn, t)

CE2.2 Terminates(SwitchOff, DeviceOn, t)

We have the following narrative:

CE2.3 Happens(SwitchOn, 2)

These formulas state what effects some actions have. But they say nothing about what effects they do not have and which events do not occur. For instance, the following could also be the case:

CE2.4 Initiates(SwitchOn, Snowing, t)

CE2.5 Happens(SwitchOff, 5)

By using circumscription, we ensure that this is not the case, i.e. no unintended events occur and events do not have unintended effects.

Result of using circumscription on CE2.1, written as *CIRC[CE2.1; Initiates]*, is the following formula:

CE2.6 $(e = SwitchOn \wedge f = DeviceOn) \Leftrightarrow Initiates(e, f, t)$

Similarly *CIRC[CE2.2; Terminates]* is:

CE2.7 $(e = SwitchOff \wedge f = DeviceOn) \Leftrightarrow Terminates(e, f, t)$

And *CIRC[CE2.3; Happens]* is:

CE2.8 $(e = SwitchOn \wedge t = 2) \Leftrightarrow Happens(e, t)$.

We will discuss how we compute circumscription shortly. For now, let us look at this example more closely. If we know that the only event occurring is CE2.3, then from CE2.6, CE2.7, CE2.8 and DEC axioms, we can conclude that *HoldsAt(DeviceOn, 6)*.

However, if later we find out that in addition to CE2.3, event CE2.5 also occurs, then the circumscription of *CIRC[CE2.3 ∧ CE2.5; Happens]* is:

$(e = SwitchOn \wedge t = 2) \vee (e = SwitchOff \wedge t = 5) \Leftrightarrow Happens(e, t)$.

From this, CE2.6, CE2.7 and DEC axioms we can no longer conclude that *HoldsAt(DeviceOn, 6)*. In fact, we can actually conclude *¬HoldsAt(DeviceOn, 6)*. This, clearly, shows the non-monotonicity property of circumscription. It also shows how elaboration tolerant the event calculus and circumscription are, i.e. we add new axioms without needing to change any of the previous ones.

Computing circumscription: For computing circumscription, there are two significant theorems which are due to Lifschitz [41]:

Theorem 1: Let ρ be an n-ary predicate symbol and $\Delta(x_1,\ldots,x_n)$ be a formula whose only free variables are $x_1,\ldots,x_n$. If $\Delta(x_1,\ldots,x_n)$ does not contain ρ, then the basic circumscription $CIRC[\forall x_1,\ldots,x_n\ (\Delta(x_1,\ldots,x_n) \Rightarrow \rho(x_1,\ldots,x_n));\ \rho]$ is equivalent to $\forall x_1,\ldots,x_n\ (\Delta(x_1,\ldots,x_n) \Leftrightarrow \rho(x_1,\ldots,x_n))$.

Proof: Please see the proof of proposition 2 in Lifschitz [41].

Using this, we can compute circumscription of ρ in a formula by rewriting the formula in the form:

$\forall x_1,\ldots,x_n\ (\Delta(x_1,\ldots,x_n) \Rightarrow \rho(x_1,\ldots,x_n))$

where $\Delta(x_1,\ldots,x_n)$ does not contain ρ and then apply Theorem 1.

Although this is the most widely used method for computing circumscription, rewriting the formula in this way may not always be possible. However, due to flexibility and robustness of the Event Calculus, we can often formulate the formulas in a way to get around this problem in the first place.

For instance:

- HoldsAt(Holding(a, o), t) $\wedge$
  Initiates(e, InRoom(a, r), t) $\Rightarrow$
  Initiates(e, InRoom(o, r), t)

This formula says that if an agent is holding an object and by some event such as entering a room then they will be in that room, and so will the object. Unfortunately this formula cannot be rewritten so that we can apply Theorem 1. However, we can solve the problem by writing the formula in a different way in the first place:

- Initiates(Walk(a, r_1, r_2), InRoom(a, r_2), t)
- HoldsAt(Holding(a, o), t) $\Rightarrow$
  Initiates(Walk(a, r_1, r_2), InRoom(o, r_2), t)

Now we can simply apply Theorem 1 to these *two* formulas.

The second theorem provides a method for computing parallel circumscription of the circumscription of several predicates. Before mentioning the theorem, however, we need to explain a definition: A formula Δ is positive relative to a predicate symbol ρ if and only if all occurrences of ρ in Δ are in the range of even number of negations in an equivalent formula obtained by eliminating $\Rightarrow$ and $\Leftrightarrow$ from Δ. We eliminate $\Rightarrow$ from a formula by replacing all instances of $(\alpha \Rightarrow \beta)$ with $(\neg\alpha \vee \beta)$. We eliminate $\Leftrightarrow$ from a formula by replacing all instances of $(\alpha \Leftrightarrow \beta)$ with $((\neg\alpha \vee \beta) \wedge (\neg\beta \vee \alpha))$.

Theorem 2: Let $\rho_1, \ldots, \rho_n$ be predicate symbols and Δ be a formula. If Δ is positive relative to every ρ_i, then the parallel circumscription $CIRC[\Delta; \rho_1, \ldots, \rho_n]$ is equivalent to the conjunction of the basic circumscription $\bigwedge_{i=1}^{n}$ CIRC[Δ; ρ_i].

Proof: Please see the proof of proposition 14 in Lifschitz [41].

Let us consider some examples:

Let Δ = *Happens(E1, T1)* $\wedge$ *Happens(E2, T2)*. We can compute *CIRC[*Δ*; Happens]* by rewriting Δ as the logically equivalent formula:

(e = E1 $\wedge$ t = T1) $\vee$ (e = E2 $\wedge$ t = T2) $\Rightarrow$ Happens(e, t).

Applying Theorem 1 to the above formula gives:

(e = E1 $\wedge$ t = T1) $\vee$ (e = E2 $\wedge$ t = T2) $\Leftrightarrow$ Happens(e, t).

which is the result of our circumscription.

Another example: Let Σ = Initiates(E1(x), F1(x), t) $\wedge$ Initiates(E2(x, y), F2(x, y), t). We can compute *CIRC[Σ; Initiates]* by rewriting *Σ* as

$\exists x(e = E1(x) \wedge f = F1(x)) \vee \exists x, y(e = E2(x, y) \wedge f = F2(x, y)) \Rightarrow$ Initiates(e, f, t).

Applying Theorem 1 to the above formula gives:

$\exists x(e = E1(x) \wedge f = F1(x)) \vee \exists x, y(e = E2(x, y) \wedge f = F2(x, y)) \Leftrightarrow$ Initiates(e, f, t).

2.4 A natural language example

Consider the following scenario involving three locations (Mueller [14, Ch. 10, 15]):

"The location *L1* is to the left of *L2*, which is to the left of *L3*. Our view of location *L2* is blocked by a screen. Suppose we observe the following (the only moves possible are between adjacent locations):

At timepoint 0, we observe an object, let us call it *O1*, at *L1* and nothing at *L3*. At timepoint 1, we observe no objects at *L1* or *L3*. At timepoint 2, we observe an object, let us call it *O2*, at *L3* and nothing at *L1*. We observe nothing about *L2* because it is blocked by a screen."

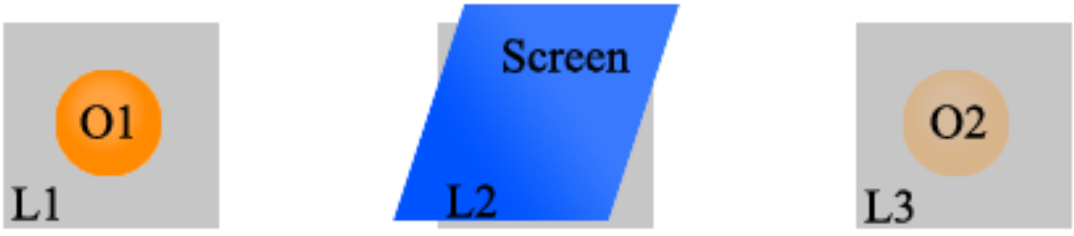

Figure 2.2: The Screen Example

Figure 2.2 shows a visual representation of the different timepoints. We examine how the event calculus can represent and reason with this description and conclude that *O1* and *O2* are indeed the same object:

First we need a simple spatial theory. Please note that this information is background knowledge and a commonsense reasoner should have these at its disposal:

An object is exactly at one location at a time:

Ex2.1 $HoldsAt(At(o, p1), t) \wedge HoldsAt(At(o, p2), t) \Rightarrow p1 = p2$

An object does have a location at a time:

Ex2.2 $\exists l\ HoldsAt(At(o, l), t)$

Two objects cannot occupy the same location at the same time:

Ex2.3 $HoldsAt(At(o1, l), t) \wedge HoldsAt(At(o2, l), t) \Rightarrow o1 = o2$

The Adjacent predicate is symmetric:

Ex2.4 $Adjacent(p1, p2) \Leftrightarrow Adjacent(p2, p1)$

If an object moves from a location to another adjacent location, then the object will be at the new location and no longer at the old location:

Ex2.5 $HoldsAt(At(o, p1), t) \bigwedge Adjacent(p1, p2) \Rightarrow$

$Initiates(Move(o, p1, p2), At(o, p2), t)$

Ex2.6 $HoldsAt(At(o, p1), t) \bigwedge Adjacent(p1, p2) \Rightarrow Terminates(Move(o,$

$p1, p2), At(o, p1), t)$

The rest of the formulas are domain specific information. Ex2.1 through Ex2.6 are general world knowledge which are true in general and could be reused to solve many other problems.

In any commonsense reasoning problem, there are two kinds of information:

- Background (world) knowledge
- Domain specific information

As we will shortly see, we can apply this small fragment of world knowledge we have to another example with a different domain-specific set of formulas and a different proposition. We focus on the current example.

We have three adjacent locations, let us call them *L1*, *L2* and *L3* as *L1* is to the left of *L2* and *L2* is to the left of *L3*. This is represented in EC as follows:

Ex2.7 $\text{Adjacent}(p1, p2) \Leftrightarrow (p1 = L1 \wedge p2 = L2) \vee (p1 = L2 \wedge p2 = L1) \vee (p1 = L2 \wedge p2 = L3) \vee (p1 = L3 \wedge p2 = L2)$

We also have the following observations:

Timepoint 0

(*Object O1 holds at location L1 at timepoint 0*)

Ex2.8 $\text{HoldsAt}(\text{At}(O1, L1), 0)$

(*There is nothing at location L3 at timepoint 0*)

Ex2.9 $\neg\text{HoldsAt}(\text{At}(o, L3), 0)$

Timepoint 1

(*There is nothing at location L1 at timepoint 1*)

Ex2.10 $\neg\text{HoldsAt}(\text{At}(o, L1), 1)$

(*There is nothing at location L3 at timepoint 1*)

Ex2.11 ¬HoldsAt(At(o, L3), 1)

Timepoint 2

(*Object O2 is at location L3 at timepoint 2*)

Ex2.12 HoldsAt(At(O2, L3), 2)

(*There is nothing at location L1 at timepoint 2*)

Ex2.13 ¬HoldsAt(At(o, L1), 2)

General

(*Commonsense law of Inertia is preserved at all timepoints.*

No fluent is released from this law at any timepoint.)

Ex2.14 ¬ReleasedAt(f, t)

ReleasedAt axiom in Ex2.14 was explained in section 2.2.2. To recap, this axiom basically says that Commonsense law of Inertia is preserved at all timepoints.
No fluent is released from this law at any timepoint. Therefore, a fluent holds or does not hold continuously (from a timepoint onwards) unless an event happens to initiate or terminate it. Whereas when a fluent that holds at a timepoint and does not submit to the commonsense law of Inertia does not necessarily hold for the next timepoint (and vice-versa: for a non-submissive fluent that does not hold for a timepoint, it is not the case that it does not hold for the following timepoint).

The event calculus makes use of Unification [58] to match its variables. Please note variables in the EC are represented with small capitals (such as *o*) whilst constants in big capitals (e.g. *O1*).

Proposition: We want to prove that *O1* and *O2* are the same object.

Let Σ be the conjunction of Ex2.5 and Ex2.6, Ψ the conjunction of Ex2.1 to Ex2.4 and Γ the conjunction of Ex2.7 to Ex2.14. Suppose

CIRC[Σ; Initiates, Terminates, Releases] ∧ Ψ ∧ Γ ∧ DEC

Then O1 = O2.

Proof:

First, we need to compute the circumscriptions to know the events and their intended effects in the scenario. Having computed the results of the circumscriptions, we can only consider the events and effects specified in our scenario.

We do not have any events in this example. So we only circumscribe *Initiates, Terminates* and *Releases*. We apply Theorems 1 and 2 to *CIRC[Σ; Initiates, Terminates, Releases]* to obtain:

Ex2.15 $\text{Initiates}(e, f, t) \Leftrightarrow \exists o, p_1, p_2(e = \text{Move}(o, p_1, p_2) \wedge f = \text{At}(o, p_2) \wedge \text{HoldsAt}(\text{At}(o, p_1), t) \wedge \text{Adjacent}(p_1, p_2))$

Ex2.16 $\text{Terminates}(e, f, t) \Leftrightarrow \exists o, p_1, p_2(e = \text{Move}(o, p_1, p_2) \wedge f = \text{At}(o, p_1) \wedge \text{HoldsAt}(\text{At}(o, p_1), t) \wedge \text{Adjacent}(p_1, p_2))$

In the text, we do not have any events happening which release any fluents from the commonsense law of Inertia at any timepoint. Therefore we have:

Ex2.17 ¬Releases(e, f, t)

Since axiom Ex2.17 indicates no event happens to release any fluents from Inertia law, and by Ex2.14 we know that no fluent is released from this law at any timepoint, we can conclude all fluents stay submissive to Inertia law at all times (and only *Initiates* and *Terminates* predicates change the state of the fluents).

From Ex2.12 and the contrapositive of DEC6 we have:

Ex2.18 HoldsAt(At(O2, L3), 1) ∨ ReleasedAt(At(O2, L3), 2) ∨

∃e(Happens(e, 1) ∧ Initiates(e, At(O2, L3), 1))

Which means that it is the case that either:

- O2 is at L3 at timepoint 1; which cannot be the case because by Ex2.11 we know there is nothing at L3 at timepoint 1.
- Fluent *At(O2, L3)* is released from Inertia; which by Ex2.14 we know is not the case.
- Or there is an event that happens at timepoint 1 that initiates the fluent *At(O2, L3)* at timepoint 1. Since this axiom must be true and the above two cases did not hold, therefore we can conclude that this must be the case.

So formally, from Ex2.18, Ex2.11 and Ex2.14 we have

Ex2.19 ∃e(Happens(e, 1) ∧ Initiates(e, At(O2, L3), 1))

By Ex2.15 we know that for an event to initiate a fluent at a time, that event must be the *Move* action, that fluent must be the *At* fluent, the object must hold at a

location and that the two locations must be adjacent. From Ex2.19 we know that an event happens to initiate *At* at timepoint 1. By unifying variables and constants of Ex2.15 and Ex2.19 we have:

$$\text{Initiates}(e, f, t) \Leftrightarrow \exists p_1(e = \text{Move}(O2, p_1, L3) \wedge f = \text{At}(O2, L3) \wedge \text{HoldsAt}(\text{At}(O2, p_1), t) \wedge \text{Adjacent}(p_1, L3))$$

Which says that the event *Move(O2, p_1, L3)* at a timepoint *t* initiates the fluent *At(O2, L3)* whilst *O2* is at location p_1 at that timepoint and locations p_1 and *L3* are adjacent.

From this and Ex2.7 we can associate p_1 with *L2* since it is the only location adjacent to *L3*. So our formula becomes:

$$\text{Initiates}(e, f, t) \Leftrightarrow e = \text{Move}(O2, L2, L3) \wedge f = \text{At}(O2, L3) \wedge \text{HoldsAt}(\text{At}(O2, L2), t) \wedge \text{Adjacent}(L2, L3)$$

We therefore have:

Ex2.20 $\text{HoldsAt}(\text{At}(O2, L2), 1)$

So far, we have concluded that object *O2* holds at location *L2* at timepoint 1.

From Ex2.10 we have ¬*HoldsAt(At(O1, L1), 1)*. From this and the contrapositive of DEC5 we have:

Ex2.21 $\neg\text{HoldsAt}(\text{At}(O1, L1), 0) \vee \text{ReleasedAt}(\text{At}(O1, L1), 1) \vee \exists e(\text{Happens}(e, 0) \wedge \text{Terminates}(e, \text{At}(O1, L1), 0))$

Ex2.8 says that object *O1* is at location *L1* at timepoint 0. Therefore ¬*HoldsAt(At(O1, L1), 0)* from the above axiom simply is not true.

Also, from Ex2.14 we know that no fluent is released from Inertia at any timepoint therefore *ReleasedAt(At(O1, L1), 1)* cannot be true. So the third condition in the disjunction in Ex2.21 must be the case.

Formally, from Ex2.21, Ex2.8 and Ex2.14 we have:

Ex2.22 $\exists e(Happens(e, 0) \wedge Terminates(e, At(O1, L1), 0))$

From this, Ex2.16 and Ex2.7 we have *Happens(Move(O1, L1, L2), 0)*. From this, Ex2.8, Ex2.7, Ex2.15 and DEC9 we have:

Ex2.23 HoldsAt(O1, L2), 1)

We now concluded that object *O1* holds at location *L2* at timepoint 1.

Ex2.20 says that object O2 holds at location L2 at timepoint 1 and Ex2.23 says that object O1 holds at location L2 at timepoint 1. By Ex2.3 we know that if two objects are at the same location at the same time, then those objects must be the same object. Therefore we infer that *O1 = O2* which is our proposition. End of proof.

We now present another scenario which uses the same background knowledge presented in the previous example (Ex2.1 to Ex2.6).

Consider the following scenario involving 5 locations: Location *L1* is to the left of *L2* which is to the left of *L3* which is to the left of *L4* which is to the left of *L5*. Our view of locations *L2* and *L4* are blocked by screens. Suppose we observe the following (the only moves possible are between adjacent locations):

At timepoint 0, we observe an object, let us call it *O1*, at *L1* and nothing at *L5*. At timepoint 4 we observe an object, let us call it *O2*, at *L5* and nothing at *L1*. We

never observe an object at *L3* and we never observe anything about *L2* or *L4* (because of the blocking caused by the screens). In this case, we can easily show that the two objects are indeed different. The proof for this example is quite similar to the former example [15].

The EC can be used to solve a large variety of examples. Many benchmark problems have been tacked and solved using the event calculus. For instance Shanahan has shown the egg-cracking scenario in [6].

Chapter 3

The Bucket World Scenario

3.1 The Bucket Scenario

We developed a framework based on the idea of a real life example given by Shin and Davis [40]. The framework is of modelling a world in which physical rules (in a commonsense perspective) apply. Our framework can handle different scenarios and we formally prove a proposition for one. This framework gives us a practical ground to compare our formalism against that of Shin and Davis on a similar scenario. They use PDDL+ to represent their theory. The scenario for which we constructed the EC representation and proved our proposition is as follows:

Fred is initially at home which is 20 meters away from the well. There are two empty 2-litre buckets at the well which potentially have spilling amount of 0.1 litre per second. There is a tap at the well which pours water at 0.1 litre per second when on. It is initially off. One of the buckets is initially under the tap. Fred can move at the speed of 1 meter per second. Fred goes to the well and then fills up both of the buckets of the total of 3 litres and pick them up and takes them home. He then pours them into a 10-litre bucket at home which already holds 2 litres of water.

For simplicity's sake, we assume some actions as atomic actions such as picking up and putting down buckets; representing these is trivial (in an abstract level) in this scenario; but will not impose any interesting challenges apart from adding to the timepoints. Expanding them in detail, however, would be an interesting follow up work; expanding and relating our framework with the Liquid Theory of Davis [9] to such a level of detail to deal with the movement of the liquid inside the buckets. That will be a first of its kind effort to expand two well developed frameworks and connect them together; trying the examples which previously worked separately on individual frameworks merged in a new environment, observing and analysing the difference in the proving process and timepoint changes.

3.2 Details of scenario

We have manually translated from the natural language description of the scenario above into the formulas of the EC. We will present our representations in the next section. However, we encountered interesting points during the translation process which will be of high importance in automating this process:

- How the initial state is identified in a scenario: since the reasoning (prediction, planning and model finding) in the EC starts from the initial state, and the fluents of the initial state have an important role in determining the fluents in the consecutive timepoints, it is crucial to identify and separate the initial state from the text in EC correctly.
- How the timepoints in a scenario are associated with narratives: being able to differentiate between timepoints in a scenario and temporally ordering

them is essential in constructing a precise domain description and narrative.

- How the temporal anaphora are handled in the EC: the temporal anaphora such as “then”, “after” and “before” significantly help us to temporally order the fluents and events of the domain description and narrative while translating. For instance if we know of event E1 occurring “before” event E2, we then know that *Happens(E1, t1)* $\wedge$ *Happens(E2, t2)* $\wedge$ *t1<t2*.
- How conditions in a scenario are dealt with: conditions in a discourse often depend on other fluents or events in the same discourse. For instance “Fred will go walking only if the weather is sunny” can be represented as: *Happens(GoWalking(Fred), t)* $\Rightarrow$ *HoldsAt(SunnyWeather, t)* – this is an action precondition: going walking event happens only if the weather is sunny. Conditions could be of more complicated form, as in our example.
- Flags: we implement and use a *flagging* system to control triggered events. Triggered events are events which are provoked based on all their conditions being satisfied. For instance consider:

$$\text{HoldsAt}(c_1, t) \wedge \text{HoldsAt}(c_2, t) \Rightarrow \text{Happens}(e, t).$$

In this formula event *e* happens only when both c_1 and c_2 hold. Now these two conditions can be *normal* formulas, for instance, the level of a bucket: *HoldsAt(Level(b_1), t)* as seen in BR50 in our scenario:

$$\text{HoldsAt}(\text{PouringFromTo}(a, b1, b2), t) \wedge \text{HoldsAt}(\text{BucketFull}(b2), t) \wedge \text{HoldsAt}(\text{Level}(b1, x), t) \wedge x>0 \wedge \text{HoldsAt}(\text{SpillAmount}(b1, y), t) \Rightarrow \text{Happens}(\text{IncreaseWastedLiquid}(y), t)$$

which states *if a bucket is receiving from another bucket and is full, then the liquid is being wasted*. It is important to note that the event *IncreaseWastedLiquid* keeps happening until one of its conditions does not hold anymore, for example the level of the pouring bucket becomes zero (i.e. the pouring bucket becomes empty). At times, especially when planning a strategy as in our scenario, we need to falsify a condition of a triggered event with a flag. For instance in BNE2:

$$\text{HoldsAt}(\text{At}(\text{Fred}, \text{Well}), t) \wedge \neg\text{HoldsAt}(\text{On}(\text{Tap}), t) \wedge$$
$$\text{HoldsAt}(\text{Cond1}, t) \Rightarrow$$
$$\text{Happens}(\text{TurnOnTap}(\text{Fred}, \text{Tap}), t)$$

we specify that *once Fred is at the well and the tap is not on, he turns the tap on*. Without a flag the event of turning on the tap would occur at every timepoint that Fred is at the well and the tap is not on. By introducing Cond1, a flag, we ensure this is not the case because once this event occurs, then it will have the effect axiom BNC1:

$$\text{HoldsAt}(\text{Cond1}, t) \Rightarrow$$
$$\text{Terminates}(\text{TurnOnTap}(\text{Fred}, \text{Tap}), \text{Cond1}, t).$$

This effect axiom (which only has this effect if Cond1 holds) falsifies Cond1 therefore one of the conditions of BNE2, the flag, does not hold anymore hence it happens only once in the scenario.

In another example in our scenario we have: "*Fred goes to the well and then fills up both of the buckets of the total of 3 litres and pick them up and takes them home."* . The "picking up" and "taking home" events are conditioned on the buckets satisfying the intended amount of liquid full and Fred being at the well. Once these conditions hold, then the respective

events are provoked. In order to prevent these events from being triggered repeatedly, we use flags. We need to identify the conditional relations between fluents and events in an EC scenario, and in this example, extracted from a scenario in natural language text.

- How we differentiate between permanent and temporary fluents: some of the fluents are temporary and will change over time, but some fluents in the same form do not. This usually happens in fluents dealing with integers or constant changes. For instance, *DistanceBetween(Well, Home, 20)* is a permanent fluent and not likely to ever change where as *DistanceBetween(George, Home, 10)* or *DistanceBetween(Car, Office, 50)* are not. The difference in translating these fluents is using a variable for the timepoint. For instance for the distance between Home and Well above we will translate this in EC as: *HoldsAt(DistanceBetween(Well, Home, 20), t)*. Note that we used a timepoint *variable* '*t*' here, so it would hold for all timepoints in the scenario. Such a translation for a temporary fluent will result in contradiction when the fluent changes (there will be uniqueness of values constraints such as:

$$\text{HoldsAt(DistanceBetween(loc1, loc2, distance1), t)} \wedge \text{HoldsAt(DistanceBetween(loc1, loc2, distance2), t)} \Rightarrow \text{distance1} = \text{distance2.}).$$

Using this constraint (which is necessary to have), when the value of a fluent changes if it holds for all timepoints we will reach a contradiction. This is why we need to correctly identify permanent and temporary fluents. One solution is to always use temporary fluents as the permanent fluents will be available in the consecutive timepoints (as nothing is supposed to

happen to *terminate* or *release* them. However this will result in unnecessary computation).

- How agents in a scenario are identified: in most of the scenarios and worlds, many actions are directly performed by an agent. Depending on the formulation, and as in our example, an agent is sometimes required to be specified as an agent fluent.
- "Intelligent" formulas can be posed in the EC. Unlike a typical programming language and similar to a logic programming language, the formulas in the EC are defined as a whole. One rule represented in the EC can only represent so much on its own. So formulas in the EC can be looked at as individual methods in a programming language that heavily depend on each other. We experienced that, with enough basic formulas, more interesting formulas can be constructed representing more intelligent behaviour. For instance, formula BR52 of our framework represent such behaviour:

  BR52 indicates if we are pouring into a bucket (from the tap or another bucket), if the bucket that we are filling in gets full and if there is another bucket around which is not full, then we replace the full bucket with the non-full bucket.

In our proof, we refer to some fluents with a *C* postfix such as BNI7*C*. The *C* stands for *Constant*. The constant fluents keep their value since they are not *released* and nothing happens to terminate them between the timepoint they hold at and the timepoint we are addressing them. If t_2 is the timepoint we are addressing the fluent, we formally have:

For *C* Fluents:

$$\text{HoldsAt}(f, t_1) \Rightarrow \neg\exists t\ (\text{Happens}(e, t) \wedge (\text{Terminates}(e, f, t) \vee \text{Releases}(e, f, t)) \wedge t_1 \leqslant t \leqslant t_2$$

and

$$\neg\text{HoldsAt}(f, t_1) \Rightarrow \neg\exists t\ (\text{Happens}(e, t) \wedge \text{Initiates}(e, f, t) \wedge t_1 \leqslant t \leqslant t_2$$

So if nothing happens to release a fluent from Inertia in between the timepoints (DEC11) then by DEC5 and DEC6 we can deduce that the fluent holds or does not hold (depending on its condition in t_1, the former timepoint) at all consecutive timepoints at least till t_2, our referring timepoint.

This convention is for saving space and as just discussed above is correct.

BX1, BX2, BX3 and BX4 are the computed circumscription of *Happens*, *Initiates*, *Terminates* and *Releases*. This means the only events that *can happen* (in the framework) and *do happen* (in the narrative) in our scenario are the ones in BX1. In other words we can say nothing else *can happen* (in the framework) nor *happens* (in the narrative). Also, BX2 and BX3 state the only effects that these events have are the ones in these two formulas. For instance the event of *TurnTapOn(Fred, Tap)* does not initiate *Walking(Fred)*. This is not allowed in our framework by using circumscription. BX4 also circumscribes *Releases* which releases a fluent from Inertia. As we can see, *PickUp(a, b)* only releases *At(b, l)* which is the location of the item *b*, not its level for instance. In all these four formulas, we use a bi-implication that indicates an if-and-only-if relationship, therefore formally nothing else is in the scope of *Happens*, *Initiates*, *Terminates* and *Releases* apart from those stated in the bi-implication.

In computing circumscription, we take into account all the instances of these four predicates in the narrative (scenario-specific) and in the framework (general theory). This is only logical because by using default reasoning we want to say that the only events that can happen (framework) and do happen (narrative) are the ones that we know of.

3.3 Representation in Event Calculus

We represent the domain description and narrative of the scenario in this section.

The narrative:

Narrative of the Bucket scenario	
	Initial state
BNI1	HoldsAt(At(Fred, Home), 0)
BNI2	HoldsAt(BucketEmpty(Bucket1), 0)
BNI3	HoldsAt(BucketEmpty(Bucket2), 0)
BNI4	HoldsAt(At(Bucket1, Well), 0)
BNI5	HoldsAt(At(Bucket2, Well), 0)
BNI6	HoldsAt(Level(Bucket3, 2), 0)
BNI7	¬HoldsAt(On(Tap), 0)
BNI8	HoldsAt(Beneath(Bucket1, Tap), 0)
BNI9	HoldsAt(AlreadyPoured(0), 0)
BNI10	HoldsAt(WastedLiquid(0), 0)
BNI11	¬HoldsAt(Carrying(Fred, Bucket1), 0)
BNI12	¬HoldsAt(Carrying(Fred, Bucket2), 0)
	Permanent Fluents
BNP1	HoldsAt(DistanceBetween(Home, Well, 20), t)
BNP2	HoldsAt(Capacity(Bucket1, 2), t)
BNP3	HoldsAt(Capacity(Bucket2, 2), t)
BNP4	HoldsAt(At(Tap, Well), t)
BNP5	HoldsAt(FlowRate(Tap,0.1), t)
BNP6	HoldsAt(At(Bucket3, Home), t)
BNP7	HoldsAt(Capacity(Bucket3, 10), t)
BNP8	HoldsAt(WalkingSpeed(Fred, 1), t)
BNP9	HoldsAt(IntendedAmount(3), t)
BNP10	HoldsAt(Agent(Fred), t)
BNP11	HoldsAt(SpillAmount(Bucket1, 0.1), t)
BNP12	HoldsAt(SpillAmount(Bucket2, 0.1), t)
	Conditional events of the narrative
BNE1	Fred leaves home to go to the well at timepoint 0. Happens(GoFromTo(Fred, Home, Well), 0)
BNE2	Once Fred is at the well and the tap is not on, he turns the tap on. By using a 'flag' here we ensure that this event happens only once, and that only at the beginning of our scenario. Had we not used a flag, once Fred is finished at the well and wants to go back home, he turns off the tap but then this event would

	be triggered again and Fred would turn on the tap just after turning it off. A flag prevents this from happening. HoldsAt(At(Fred, Well), t) ⋀ ¬HoldsAt(On(Tap), t) ∧ HoldsAt(Cond1, t) ⇒ Happens(TurnOnTap(Fred, Tap), t)
BNE3	Once Fred is at home carrying bucket 1 which is not empty, Fred moves bucket 1 over bucket 3. Using the flag Cond2 here ensures that we have not yet started the pouring process. HoldsAt(At(Fred, Home), t) ⋀ HoldsAt(Carrying(Fred, Bucket1), t) ∧¬HoldsAt(BucketEmpty(Bucket1), t) ∧ HoldsAt(Cond2, t) ⇒ Happens(MoveOver(Fred, Bucket1, Bucket3), t)
BNE4	Once Fred is at home carrying bucket 1 which is not empty, if bucket 1 is over bucket 3 (i.e. bucket 3 is under bucket 1) Fred starts pouring from bucket 1 into bucket 3. Using the flag Cond2 here ensures that this event is triggered only once. HoldsAt(At(Fred, Home), t) ⋀ HoldsAt(Carrying(Fred, Bucket1), t) ∧¬HoldsAt(BucketEmpty(Bucket1), t) ∧ HoldsAt(Beneath(Bucket3,Bucket2), t)∧ HoldsAt(Cond2, t) ⇒ Happens(PourFromTo(Fred, Bucket1, Bucket3), t)
BNE5	Once Fred is at home carrying bucket 2 which is not empty, if bucket 1 is empty (making sure it has already been poured), Fred moves bucket 2 over bucket 3. Using the flag Cond3 here ensures that we have not yet started the pouring process. HoldsAt(At(Fred, Home), t) ⋀ HoldsAt(Carrying(Fred, Bucket2), t) ⋀HoldsAt(BucketEmpty(Bucket1), t) ∧¬HoldsAt(BucketEmpty(Bucket2), t) ∧ HoldsAt(Cond3, t) ⇒ Happens(MoveOver(Fred, Bucket2, Bucket3), t)
BNE6	Once Fred is at home carrying bucket 2 which is not empty, if bucket 2 is over bucket 3 (i.e. bucket 3 is under bucket 2) Fred starts pouring from bucket 2 into bucket 3. Using the flag Cond3 here ensures that this event is triggered only once. HoldsAt(At(Fred, Home), t) ⋀ HoldsAt(Carrying(Fred, Bucket2), t) ⋀HoldsAt(BucketEmpty(Bucket1), t) ∧¬HoldsAt(BucketEmpty(Bucket2), t) ∧ HoldsAt(Beneath(Bucket3,Bucket2), t)∧ HoldsAt(Cond3, t) ⇒ Happens(PourFromTo(Fred, Bucket2, Bucket3), t)
BNE7	Once Fred is at the well and has sufficient amount of liquid in his buckets which he is carrying then he leaves the well and starts going towards home. HoldsAt(Carrying(Fred, Bucket1), t) ⋀ HoldsAt(Carrying(Fred, Bucket2), t) ⋀HoldsAt(Level(Bucket1, x), t) ⋀ HoldsAt(Level(Bucket2, y), t) ∧ HoldsAt(IntendedAmount(z), t) ∧ x+y=z ∧ HoldsAt(At(Fred, Well), t) ⇒

	Happens(GoFromTo(Fred, Well, Home), t)
BNE8	Once Fred is at the well and he has sufficient liquid in his buckets, if he is not already carrying bucket 1 then he picks it up. HoldsAt(At(Fred, Well), t) ∧ HoldsAt(Level(Bucket1, x), t) ∧ HoldsAt(Level(Bucket2, y), t) ∧ HoldsAt(IntendedAmount(z), t) ∧ x+y=z ∧¬HoldsAt(Carrying(Fred, Bucket1), t) ⇒ Happens(PickUp(Fred, Bucket1), t)
BNE9	Once Fred is at the well and he has sufficient liquid in his buckets, if he is not already carrying bucket 2 then he picks it up. HoldsAt(At(Fred, Well), t) ∧ HoldsAt(Level(Bucket1, x), t) ∧ HoldsAt(Level(Bucket2, y), t) ∧ HoldsAt(IntendedAmount(z), t) ∧ x+y=z ∧¬HoldsAt(Carrying(Fred, Bucket2), t) ⇒ Happens(PickUp(Fred, Bucket2), t)
	Effect axioms to deal with conditions of the narrative events (flags in EC)
	(Terminate CondX fluents so these events are triggered only once)
BNC1	HoldsAt(Cond1, t) ⇒ Terminates(TurnOnTap(Fred, Tap), Cond1, t)
BNC2	HoldsAt(Cond2, t) ⇒ Terminates(PourFromTo(Fred, Bucket1, Bucket3), Cond2, t)
BNC3	HoldsAt(Cond3, t) ⇒ Terminates(PourFromTo(Fred, Bucket2, Bucket3), Cond3, t)
	Initial condition flags:
BNC6	HoldsAt(Cond1), 0)
BNC7	HoldsAt(Cond2), 0)
BNC8	HoldsAt(Cond3), 0)
BIN1	f ≠ At(x, y) ⇒ ¬ReleasedAt(f, t)

Domain framework:

Domain Formalisation of the Bucket World	
	Actions and their effects:
	(Actions on tap)
BR2	Initiates(TurnOnTap(a, p), On(p), t)
BR4	Terminates(TurnOffTap(a, p), On(p), t)
	(Moving Axioms)
BR5	Initiates(MoveOver(a, b1, b2), Beneath(b2, b1), t)
BR6	Initiates(PickUp(a, b), Carrying(a, b), t)
BR7	When an agent picks up an item, then the location of that item is not submissive to Inertia anymore. That means its location is not at a fixed place, but that of its carrier. HoldsAt(At(b, l), t) $\Rightarrow$ Releases(PickUp(a, b), At(b, l), t)
BR9	Putting down an object will result in the location of object being re-established at the same as the agent by reinforcing Inertia. HoldsAt(At(a, l), t) $\Rightarrow$ Initiates(PutDown(a, b), At(b, l), t)
BR10	Terminates(PutDown(a, b), Carrying(a, b), t)
BR12	l1 $\neq$ l2$\Rightarrow$ Initiates(GoFromTo(a, l1, l2), Walking(a), t)
BR13	HoldsAt(At(a, l1), t) $\wedge$ l1 $\neq$ l2 $\Rightarrow$ Terminates(GoFromTo(a, l1, l2), At(a, l1), t)
BR15	Initiates(GoFromTo(a, 1l, l2), Destination(a, l2), t)
BR16	HoldsAt(DistanceBetween(l1, l2, x), t) $\Rightarrow$ Initiates(GoFromTo(a, l1, l2), DistanceToWalk(a, x), t)
BR17	Initiates(Arrive(a, l), At(a, l), t)
BR18	Terminates(Arrive(a, l), Walking(a), t)
	(Pouring from a bucket into another:)
BR20	Initiates(PourFromTo(a, b1, b2), PouringFromTo(a, b1, b2), t)
	(Replacing buckets by each other)
BR24	Initiates(MoveUnder(a, x, y), Beneath(x, y), t)

BR27	Terminates(MoveAside(a, x, y) , Beneath(x, y), t)
BR28	HoldsAt(At(a, l), t) ⇒ Initiates(MoveAside(a, x, y) , At(x, l), t)
BR29	When replacing an item, we move the first item aside from its location. HoldsAt(Beneath(b1, p), t) ∧ Happens(Replace(a, b1, b2), t) ⇒ Happens(MoveAside(a, b1, p), t)
BR25	When replacing an item, we move the second item into the position of the first item (under an object). HoldsAt(Beneath(b1, p), t) ∧ Happens(Replace(a, b1, b2), t) ⇒ Happens(MoveUnder(a, b2, p), t)
	(Stop pouring from a bucket to another)
BR30	Initiating the fluent Level makes it submissive towards the commonsense law of Inertia. Therefore the fluent's value will hold until something happens to change it. HoldsAt(Level(b1, x), t) ⇒ Initiates(StopPouringFromTo(a, b1, b2), Level(b1, x), t)
BR31	Initiating the fluent Level makes it submissive towards the commonsense law of Inertia. Therefore the fluent's value will hold until something happens to change it. HoldsAt(Level(b2, x), t) ⇒ Initiates(StopPouringFromTo(a, b1, b2), Level(b2, x), t)
BR32	Terminates(StopPouringFromTo(a, b1, b2), PouringFromTo(a, b1, b2), t)
	Actions for changing the levels of integer fluents
BR33	HoldsAt(Level(b, y), t) ⇒ Initiates(IncreaseLiquidLevel(b, x), Level(b, x+y), t)
BR34	Increase in the liquid level makes the old value obsolete. HoldsAt(Level(b, y), t) ⇒ Terminates(IncreaseLiquidLevel(b, x), Level(b, y), t)
BR35	HoldsAt(Level(b, y), t) ∧ y>x ⇒ Initiates(DecreaseLiquidLevel(b, x), Level(b, y-x), t)
BR37	Decrease in the liquid level makes the old value obsolete. HoldsAt(Level(b, y), t) ⇒ Terminates(DecreaseLiquidLevel(b, x), Level(b, y), t)
BR38	HoldsAt(AlreadyPoured(y), t) ⇒ Initiates(IncreasePouredLiquid(x),

	AlreadyPoured(x+y), t)
BR39	Increase in the amount of AlreadyPouredLiquid makes its old value obsolete. HoldsAt(AlreadyPoured(y), t) ⇒ Terminates(IncreasePouredLiquid(x), AlreadyPoured(y), t)
BR40	HoldsAt(WastedLiquid(y), t) ⇒ Initiates(IncreaseWastedLiquid(x), WastedLiquid(x+y), t)
BR41	The amount of wasted liquid being increased makes the old value obsolete. HoldsAt(WastedLiquid(y), t) ⇒ Terminates(IncreaseWastedLiquid(x), WastedLiquid(y), t)
BR42	HoldsAt(DistanceToWalk(a, od) ⇒ Initiates(DecreaseDistanceToWalk(a, nd), DistanceToWalk(a, od-nd), t)
BR43	HoldsAt(DistanceToWalk(a, od) ⇒ Terminates(DecreaseDistanceToWalk(a, nd), DistanceToWalk(a, od), t)
	State constraints (general knowledge):
	(General commonsense knowledge of the world related to our domain)
BR44	If an agent is walking and the distance he walks is greater than 0 then decrease the distance to walk. HoldsAt(Walking(a), t) ∧ HoldsAt(DistanceToWalk(a, d), t) ∧ 0<d ∧ HoldsAt(WalkingSpeed(a, ws), t) ∧ 0<=d-ws ⇒ Happens(DecreaseDistanceToWalk(a, ws), t)
BR45	If an agent is walking and the distance to walk is less than 0 then the agent arrives at the destination. HoldsAt(Walking(a), t) ∧ HoldsAt(DistanceToWalk(a, d), t) ∧ d<=0 ∧ HoldsAt(Destination(a, l), t) ⇒ Happens(Arrive(a, l), t)
BR46	A bucket is being filled if and only if it is not full and either it is under an open tap or another bucket is pouring into it. HoldsAt(Filling(b), t) ⇔ ¬HoldsAt(BucketFull(b), t) ∧ (HoldsAt(Beneath(b, p), t) ∧ HoldsAt(On(p), t)) ∨ (HoldsAt(PouringFromTo(a, b2, b), t) ∧ HoldsAt(Level(b2, x), t) ∧ x>0)
BR47	If a tap is on and the bucket is under the tap and the bucket is not full [implicit in the Filling fluent], then increase the level of already poured amount. HoldsAt(Filling(b), t) ∧ HoldsAt(On(p), t) ∧ HoldsAt(Beneath(b, p), t) ∧ HoldsAt(FlowRate(p, f), t) ⇒ Happens(IncreasePouredLiquid(f), t)

BR74	If a tap is on and the bucket is under the tap and the bucket is not full [implicit in the Filling fluent], then increase the level of . HoldsAt(Filling(b), t) ∧ HoldsAt(On(p), t) ∧ HoldsAt(Beneath(b, p), t) ∧ HoldsAt(FlowRate(p, f), t) ⇒ Happens(IncreaseLiquidLevel(b, f), t)
BR48	If a bucket is pouring to another and still has liquid, then if the receiving bucket is not full (implicit in Filling), increase its level. HoldsAt(Filling(b), t) ∧ HoldsAt(PouringFromTo(a, b2, b), t) ∧ HoldsAt(SpillAmount(b, x), t) ∧ HoldsAt(Level(b2, y), t) ∧ y>0 ⇒ Happens(IncreaseLiquidLevel(b, x), t)
BR75	If a bucket is pouring to another and still has liquid, then if the receiving bucket is not full (implicit in Filling), decrease the level of the pouring bucket. HoldsAt(Filling(b), t) ∧ HoldsAt(PouringFromTo(a, b2, b), t) ∧ HoldsAt(SpillAmount(b, x), t) ∧ HoldsAt(Level(b2, y), t) ∧ y>0 ⇒ Happens(DecreaseLiquidLevel(b2, x), t)
BR49	If a bucket is receiving from a tap and the bucket is full, then the liquid is being wasted. HoldsAt(BucketFull(b), t) ∧ HoldsAt(Beneath(b, p), t) ∧ HoldsAt(On(p), t) ∧¬HoldsAt(Beneath(b2, p), t) ∧¬HoldsAt(BucketFull(b2), t) ∧ HoldsAt(FlowRate(p, f), t) ⇒ Happens(IncreaseWastedLiquid(f), t)
BR50	If a bucket is receiving from another bucket and is full, then the liquid is being wasted. HoldsAt(PouringFromTo(a, b1, b2), t) ∧ HoldsAt(BucketFull(b2), t) ∧ HoldsAt(Level(b1, x), t) ∧ x>0 ∧ HoldsAt(SpillAmount(b1, y), t) ⇒ Happens(IncreaseWastedLiquid(y), t)
BR51	When pouring if the giving bucket runs out of liquid, then the pouring process is stopped. HoldsAt(PouringFromTo(a, b1, b2), t) ∧ HoldsAt(Level(b1, x), t) ∧ x<=0 ⇒ Happens(StopPouringFromTo(a, b1, b2), t)
BR52	If the bucket we are filling by the tap or another bucket is full, if there is another bucket at the same location and that bucket is not full, then replace the current bucket with that bucket. (HoldsAt(PouringFromTo(a, bo, b), t) ∨ (HoldsAt(Beneath(b, p), t) ∧ HoldsAt(On(p), t))) ∧ HoldsAt(BucketFull(b), t) ∧ HoldsAt(IntendedAmount(x), t) ∧ HoldsAt(AlreadyPoured(y), t) ∧ y<x ∧ HoldsAt(At(b, l), t) ∧ HoldsAt(At(b2, l), t) ∧ ¬HoldAt(BucketFull(b2), t) ∧ HoldsAt(Agent(a), t) ⇒ Happens(Replace(a, b, b2), t)

BR53	If we have reached the intended amount, if we are filling the bucket with the tap, we turn off the tap. HoldsAt(Filling(b), t) ∧ HoldsAt(IntendedAmount(x), t) ∧ HoldsAt(AlreadyPoured(y), t) ∧ y>=x ∧ HoldsAt(At(p, l), t), ∧ HoldsAt(Agent(a), t) ∧ HoldsAt(At(a, l), t) ∧ HoldsAt(On(p), t) ∧ HoldsAt(Beneath(b, p), t) ⇒ Happens(TurnOffTap(a, p), t)
BR54	HoldsAt(Beneath(x, y), t) ⇔ HoldsAt(Over(y, x), t)
BR55	HoldsAt(Level(b, x), t) ∧ HoldsAt(Capacity(b, y), t) ∧ y<=x ⇔ HoldsAt(BucketFull(b), t)
BR56	HoldsAt(Level(b, x), t) ∧ x=0 ⇔ HoldsAt(BucketEmpty(b), t)
BR57	HoldsAt(On(x), t) ⇔ ¬HoldsAt(Off(x), t)
BR58	HoldsAt(DistanceBetween(l1, l2, x) ⇔ HoldsAt(DistanceBetween(l2, l1, x), t)
	Uniqueness of values for fluents dealing with integer sort
BR59	HoldsAt(Level(b, x1), t) ∧ HoldsAt(Level(b, x2), t) ⇒ x1=x2
BR60	HoldsAt(Capacity(b, x1), t) ∧ HoldsAt(Capacity(b, x2), t) ⇒ x1=x2
BR61	HoldsAt(DistanceBetween(l1, l2, d1), t) ∧ HoldsAt(DistanceBetween(l1, l2, d2), t) ⇒ d1=d2
BR62	HoldsAt(DistanceBetween(l1, l2, d1), t) ∧ HoldsAt(DistanceBetween(l2, l1, d2), t) ⇒ d1=d2
BR63	HoldsAt(DistanceBetween(l1, l2, d), t) ∧ d=0 ⇒ l1=l2
BR64	HoldsAt(DistanceToWalk(a, d1), t) ∧ HoldsAt(DistanceToWalk(a, d2), t) ⇒ d1=d2
BR65	HoldsAt(At(o, l1), t) ∧ HoldsAt(o, l2), t) ⇒ l1=l2
BR66	HoldsAt(WalkingSpeed(a, ws1), t) ∧ HoldsAt(WalkingSpeed(a, ws2), t) ⇒ ws1=ws2
BR67	HoldsAt(Beneath(x, y), t) ⇒¬HoldsAt(Beneath(y, x), t)
BR68	HoldsAt(AlreadyPoured(x1), t) ∧ HoldsAt(AlreadyPoured(x2), t) ⇒ x1=x2

BR69	HoldsAt(WastedLiquid(x1), t) $\wedge$ HoldsAt(WastedLiquid(x2), t) $\Rightarrow$ x1=x2
BR70	HoldsAt(IntendedAmount(x1), t) $\wedge$ HoldsAt(IntendedAmount(x2), t) $\Rightarrow$ x1=x2
BR71	HoldsAt(FlowRate(p, f1), t) $\wedge$ HoldsAt(FlowRate(p, f2), t) $\Rightarrow$ f1=f2
BR72	HoldsAt(BucketEmpty(b), t) $\Rightarrow \neg$HoldsAt(BucketFull(b), t)
BR73	HoldsAt(BucketFull(b), t) $\Rightarrow \neg$HoldsAt(BucketEmpty(b), t)

3.4 Proof of propositions

Computed circumscription of Σ and Δ (defined in the proof):

Circumscription of the events and the effect axioms	
BX1	Happens(e, t) ⇔ ∃a, l1, l2, t (e= GoFromTo(a, l1, l2) ∧ a=Fred ∧ l1=Home ∧ l2=Well ∧ t=0) ∨ ∃a, p, l, c, t (e= TurnOnTap (a, p) ∧ HoldsAt(c, t) ∧ HoldsAt(At(a, l), t) ∧ ¬HoldsAt(On(p), t) ∧ l=Well ∧ p=Tap ∧ a=Fred ∧ c=Cond1) ∨ ∃a, b1, b2, l, c, t (e= MoveOver (a, b1, b2) ∧ HoldsAt(At(a, l), t) ∧ HoldsAt(Carrying(a, b1), t) ∧ ¬HoldsAt(BucketEmpty(b1, t) ∧ HoldsAt(c, t) ∧ a=Fred ∧ l=Home ∧ b1=Bucket1 ∧b2=Bucket3 ∧ c=Cond2) ∨ ∃a, b1, b2, l, c, t (e= PourFromTo (a, b1, b2) ∧ HoldsAt(At(a, l), t) ∧ HoldsAt(Carrying(a, b1), t) ∧ ¬HoldsAt(BucketEmpty(b1), t) ∧ HoldsAt(Beneath(b3, b2), t) ∧ HoldsAt(c, t) ∧ a=Fred ∧ b1=Bucket1 ∧ b2=Bucket3 ∧ l=Home ∧ c=Cond2) ∨ ∃a, b1, b2, b3, l, c, t (e= MoveOver (a, b2, b3) ∧ HoldsAt(At(a, l), t) ∧ HoldsAt(Carrying(a, b2), t) ∧ ¬HoldsAt(BucketEmpty(b2), t) ∧ HoldsAt(BucketEmpty(b1), t) ∧ HoldsAt(c, t) ∧ a=Fred ∧ b1=Bucket1 ∧ b2=Bucket2 ∧ l=Home ∧ b3=Bucket3 ∧ c=Cond3) ∨ ∃a, b1, b2, b3, l, c, t (e= PourFromTo (a, b2, b3) ∧ HoldsAt(At(a, l), t) ∧ HoldsAt(Carrying(a, b2), t) ∧ ¬HoldsAt(BucketEmpty(b2), t) ∧ HoldsAt(BucketEmpty(b1), t) ∧ HoldsAt(Beneath(b3, b2), t) ∧ HoldsAt(c, t) ∧ a=Fred ∧ b1=Bucket1 ∧ b2=Bucket2 ∧ l=Home ∧ b3=Bucket3 ∧ c=Cond3) ∨ ∃a, b1, b2, x, y, z, l1, l2, t (e= GoFromTo (a, l1, l2) ∧ HoldsAt(Carrying(a, b1), t) ∧ HoldsAt(Carrying(a, b2), t) ∧ HoldsAt(Level(b1, x), t) ∧ HoldsAt(Level(b2, y), t) ∧ HoldsAt(IntendedAmount(z), t) ∧ HoldsAt(At(a, l), t) ∧ HoldsAt(c, t) ∧ x+y=z ∧ a=Fred ∧ b1=Bucket1 ∧ b2=Bucket2 ∧ l1=Well ∧ l2=Home) ∨ ∃a, b1, b2, x, y, z, l, t (e= PickUp (a, b1) ∧ ¬HoldsAt(Carrying(a, b1), t) ∧ HoldsAt(Level(b1, x), t) ∧ HoldsAt(Level(b2, y), t) ∧ HoldsAt(IntendedAmount(z), t) ∧ HoldsAt(At(a, l), t) ∧ HoldsAt(c, t) ∧ x+y=z ∧ a=Fred ∧ b1=Bucket1 ∧ b2=Bucket2 ∧ l=Well) ∨ ∃a, b1, b2, x, y, z, l, t (e= PickUp (a, b2) ∧ ¬HoldsAt(Carrying(a, b2), t) ∧ HoldsAt(Level(b1, x), t) ∧ HoldsAt(Level(b2, y), t) ∧ HoldsAt(IntendedAmount(z), t) ∧ HoldsAt(At(a, l), t) ∧ HoldsAt(c, t) ∧ x+y=z ∧ a=Fred ∧ b1=Bucket1 ∧ b2=Bucket2 ∧ l=Well) ∨ ∃a, ws, d (e=DecreaseDistanceToWalk(a, ws) ∧ HoldsAt(Walking(a), t) ∧ HoldsAt(DistanceToWalk(a, d), t) ∧ 0<d ∧ HoldsAt(WalkingSpeed(a, ws), t) ∧ 0<=d-ws) ∨ ∃a, l, d (e= Arrive(a, l) ∧ HoldsAt(Walking(a), t) ∧ HoldsAt(DistanceToWalk(a, d), t) ∧ d<=0) ∧ HoldsAt(Destination(a, l), t)) ∨ ∃f, b, p (e=IncreasePouredLiquid(f) ∧ e=IncreaseLiquidLevel(b, f) ∧ HoldsAt(Filling(b), t) ∧ HoldsAt(On(p), t) ∧ HoldsAt(Beneath(b, p), t) ∧

	HoldsAt(FlowRate(p, f), t)) ∨ ∃b, f, p, t (e=IncreaseLiquidLevel(b, f) ∧ HoldsAt(Filling(b), t) ∧ HoldsAt(Beneath(b, p), t) ∧ HoldsAt(FlowRate(p, f), t)) ∨ ∃b, x, b2, y, t (e=IncreaseLiquidLevel(b, x) ∧ HoldsAt(Filling(b), t) ∧ HoldsAt(SpillAmount(b, x), t) ∧ HoldsAt(Level(b2, y), t) ∧ y>0) ∨ ∃a, b, x, y, b2, t (e= DecreaseLiquidLevel (b2, x) ∧ HoldsAt(Filling(b), t) ∧ HoldsAt(SpillAmount(b, x), t) ∧ HoldsAt(Level(b2, y), t) ∧ y>0 ∧ HoldsAt(PouringFromTo(a, b2, b), t)) ∨ ∃f, b, p, b2, f (e=IncreaseWastedLiquid(f) ∧ HoldsAt(BucketFull(b), t) ∧ HoldsAt(Beneath(b, p), t) ∧ HoldsAt(On(p), t) ∧¬HoldsAt(Beneath(b2, p), t) ∧¬HoldsAt(BucketFull(b2), t) ∧ HoldsAt(FlowRate(p, f), t)) ∨ ∃y, a, b1, b2, x (e=IncreaseWastedLiquid(y) ∧ HoldsAt(PouringFromTo(a, b1, b2), t) ∧ HoldsAt(BucketFull(b2), t) ∧ HoldsAt(Level(b1, x), t) ∧ x>0 ∧ HoldsAt(SpillAmount(b1, y), t)) ∨ ∃a, b1, b2, x (e=StopPouringFromTo(a, b1, b2) ∧ HoldsAt(PouringFromTo(a, b1, b2), t) ∧ HoldsAt(Level(b1, x), t) ∧ x<=0) ∨ ∃a, b, b2, bo, p, x, y (e= Replace(a, b, b2) ∧ (HoldsAt(PouringFromTo(a, bo, b), t) ∨ (HoldsAt(Beneath(b, p), t) ∧ HoldsAt(On(p), t))) ∧ HoldsAt(BucketFull(b), t) ∧ HoldsAt(IntendedAmount(x), t) ∧ HoldsAt(AlreadyPoured(y), t) ∧ y<x ∧ HoldsAt(At(b, l), t) ∧ HoldsAt(At(b2, l), t) ∧ ¬HoldAt(BucketFull(b2), t) ∧ HoldsAt(Agent(a), t)) ∨ ∃a, p, b, x, y, l (e=TurnOffTap(a, p) ∧ HoldsAt(Filling(b), t) ∧ HoldsAt(IntendedAmount(x), t) ∧ HoldsAt(AlreadyPoured(y), t) ∧ y>=x ∧ HoldsAt(At(p, l), t), ∧ HoldsAt(Agent(a), t) ∧ HoldsAt(At(a, l), t) ∧ HoldsAt(On(p), t) ∧ HoldsAt(Beneath(b, p), t))
BX2	Initiates(e, f, t) ⇔ ∃a, p, t (e=TurnOnTap(a, p) ∧ f=On(p)) ∨ ∃a, b1, b2, t (e=MoveOver(a, b1, b2) ∧ f=Beneath(b2, b1)) ∨ ∃a, b, t (e=PickUp(a, b) ∧ f=Carrying(a, b)) ∨ ∃a, b, l, t (e=PutDown(a, b) ∧ f=At(b, l)) ∧ HoldsAt(At(a, l), t)) ∨ ∃a, l1, l2, t (e=GoFromTo(a, l1, l2) ∧ f =Walking(a) ∧ l1 ≠ l2) ∨ ∃a, l1, l2, t (e=GoFromTo(a, 1l, l2) ∧ f=Destination(a, l2)) ∨ ∃a, l1, l2, x, t (e=GoFromTo(a, l1, l2) ∧ f =DistanceToWalk(a, x) ∧ HoldsAt(DistanceBetween(l1, l2, x), t)) ∨ ∃a, l, t (e=Arrive(a, l) ∧ f=At(a, l)) ∨ ∃a, b1, b2, t (e=PourFromTo(a, b1, b2) ∧ f=PouringFromTo(a, b1, b2)) ∨ ∃a, x, y, t (e=MoveUnder(a, x, y) ∧ f=Beneath(x, y)) ∨ ∃a, x, y, l, t (e=MoveAside(a, x, y) ∧ f=At(x, l) ∧ HoldsAt(At(a, l), t)) ∨ ∃a, b1, b2, x, t (e=StopPouringFromTo(a, b1, b2) ∧ f=Level(b1, x) ∧ HoldsAt(Level(b1, x), t)) ∨ ∃a, b1, b2, x, t (e=StopPouringFromTo(a, b1, b2) ∧ f=Level(b2, x) ∧ HoldsAt(Level(b2, x), t)) ∨ ∃b, x, y, t (e=IncreaseLiquidLevel(b, x) ∧ f=Level(b, x+y) ∧ HoldsAt(Level(b, y), t)) ∨ ∃b, x, y, t (e=DecreaseLiquidLevel(b, x) ∧ f=Level(b, y-x) ∧HoldsAt(Level(b, y), t) ∧ y>x) ∨

	∃x, y, t (e=IncreasePouredLiquid(x), ∧ f=AlreadyPoured(x+y) ∧HoldsAt(AlreadyPoured(y), t)) ∨ ∃x, y, t (e=IncreaseWastedLiquid(x) ∧ f=WastedLiquid(x+y) ∧HoldsAt(WastedLiquid(y), t)) ∨ ∃a, nd, od, t (e=DecreaseDistanceToWalk(a, nd) ∧ f=DistanceToWalk(a, od nd) ∧HoldsAt(DistanceToWalk(a, od))
BX3	Terminates(e, f, t) ⇔ ∃a, p, c, t (e=TurnOnTap(a, p) ∧ f=c ∧ c=Cond1 ∧ HoldsAt(c, t) ∧ a=Fred ∧ p=Tap) ∨ ∃a, b1, b2, c, t (e=PourFromTo(a, b1, b2) ∧ f=c ∧ c=Cond2 ∧ HoldsAt(c, t) ∧ a=Fred ∧ b1=Bucket1 ∧ b2=Bucket3) ∨ ∃a, b1, b2, c, t (e=PourFromTo(a, b1, b2) ∧ f=c ∧ c=Cond3 ∧ HoldsAt(c, t) ∧ a=Fred ∧ b1=Bucket2 ∧ b2=Bucket3) ∨ ∃a, p, t (e=TurnOffTap(a, p) ∧ f=On(p)) ∨ ∃a, b, t (e=PutDown(a, b) ∧ f=Carrying(a, b)) ∨ ∃a, l1, l2, t (e=GoFromTo(a, l1, l2) ∧ f= At(a, l1) ∧ HoldsAt(At(a, l1), t) ∧ l1 ≠ l2) ∨ ∃a, l, t (e=Arrive(a, l)∧ f=Walking(a)) ∨ ∃a, x, y, t (e=MoveAside(a, x, y)∧ f=Beneath(x, y)) ∨ ∃a, b1, b2, x, y, t (e=StopPouringFromTo(a, b1, b2) ∧ f=PouringFromTo(a, b1, b2)) ∨ ∃b, x, y, t (e=IncreaseLiquidLevel(b, x) ∧ f=Level(b, y) ∧ HoldsAt(Level(b, y), t) ∧ x>0) ∨ ∃b, x, y, t (e=DecreaseLiquidLevel(b, x) ∧ f=Level(b, y)∧HoldsAt(Level(b, y), t) ∧ x>0) ∨ ∃x, y, t (e=IncreasePouredLiquid(x)∧ f=AlreadyPoured(y) ∧ HoldsAt(AlreadyPoured(y), t) ∧ x>0) ∨ ∃x, y, t (e=IncreaseWastedLiquid(x)∧ f=WastedLiquid(y)∧HoldsAt(WastedLiquid(y), t) ∧ x>0) ∨ ∃a, nd, od, t (e=DecreaseDistanceToWalk(a, nd)∧ f=DistanceToWalk(a, od)∧HoldsAt(DistanceToWalk(a, od))
BX4	Releases(e, f, t) ⇔ ∃a, b, l (e=PickUp(a, b) ∧ f=At(b, l) ∧ HoldsAt(At(b, l), t))

The proof our proposition is presented below:

Proof of a proposition for the Bucket scenario	
	Let Σ be the conjunction of (Initiates, Terminates and Releases formulas): BR2, BR5, BR6, BR9, BR12, BR15, BR16, BR17, BR20, BR24, BR28, BR30, BR31, BR33, BR35, BR38, BR40, BR42, BNC1, BNC2, BNC3, BR4, BR10, BR13, BR18, BR27, BR32, BR34, BR37, BR39, BR41, BR43 and BR7. Let Δ be conjunction of (Happens formulas): BNE1, BNE2, BNE3, BNE4, BNE5, BNE6, BNE7, BNE8, BNE9, BR44, BR45, BR47, BR74, BR48, BR75, BR49, BR50, BR51, BR52 and BR53. Let Ψ be the conjunction of all other formulas in our scenario.
	Proposition
	Our proposition states that the level of Bucket 3 will be 5 at a timepoint. Our proof shows this and provides the timepoint at which this will be true. Formally, CIRC[Σ; Initiates, Terminates, Releases] $\wedge$ CIRC[Δ; Happens] $\wedge\Psi\wedge$ DEC $\models$ HoldsAt(Level(Bucket3, 5), t) Circumscription of Σ is computed in BX2, BX3 and BX4 respectively. Circumscription of Δ is computed in BX1. Computing circumscription of Happens ensures that the only events that occur in our scenario are the ones computed in their circumscription and *nothing else happens*. Also, by circumscribing Initiates, Terminates and Releases we ensure that the effects of these events are the ones in their circumscription and they *do not have other effects*.
BXP1 **BXP2** **BXP3** **BXP4**	BNE1 happens at timepoint 0 - this is in the narrative: Happens(GoFromTo(Fred, Home, Well), 0) From **this and**: • BR12: Initiates(GoFromTo(Fred, Home, Well), Walking(Fred), 0) From this, BNE1 and DEC9 we have: **BXP1**: HoldsAt(Walking(Fred), 1) • BNI1, BR13 and BNE1:

Terminates(GoFromTo(Fred, Home, Well), At(Fred, Home), 0)

From this, BNE1 and DEC10 we have:

BXP2: ¬HoldsAt(AT(Fred, Home), 1)

- BR15:

  Initiates(GoFromTo(Fred, Home, Well), Destination(Fred, Well), 0)

  From this, BNE1 and DEC9 we have:

  BXP3: HoldsAt(Destination(Fred, Well), 1)

- BNP1 and BR16:

  Initiates(GoFromTo(Fred, Home, Well), DistanceToWalk(Fred, 20), 0)

  From this, BNE1 and DEC9 we have:

  BXP4: HoldsAt(DistanceToWalk(Fred, 20), 1)

BXP5
BXP6
BXP7

From BR44, BXP1, BXP4 and BNP8 we have:

BXP5: Happens(DecreaseDistanceToWalk(Fred, 1), 1)
(this event keeps occurring at every timepoint as long as all the conditions hold - therefore the derived deductions will also be repeated)

From **this and**:

- BR42:

  BXP6: Initiates(DecreaseDistanceToWalk(Fred, 1), DistanceToWalk(Fred, 19), 1)

- BR43:

  BXP7: Terminates(DecreaseDistanceToWalk(Fred, 1), DistanceToWalk(Fred, 20), 1)

BXP8 From BXP5, BXP6 and DEC9 we have:

HoldsAt(DistanceToWalk(Fred, 19), 2)

The same deduction is repeatedly at every timepoint applied. Therefore we have a decreasing distance till timepoint 21. To save space, we do not repeat this.
….
[we eventually reach the following:]

BXP8: HoldsAt(DistanceToWalk(Fred, 0), 21)

BXP9 From BXP5, BXP7 and DEC10 we have:

¬HoldsAt(DistanceToWalk(Fred, 20), 2)

The same deduction is repeatedly at every timepoint applied. Therefore we have an obsolete decreasing distance till timepoint 21. To save space, we do not repeat this.
….
[we eventually reach the following:]

BXP9: ¬HoldsAt(DistanceToWalk(Fred, 1), 21)

BXP10 From BR45 and BXP1*C* and BXP8 and BXP3*C* we have:
BXP11 **BXP10**: Happens(Arrive(Fred, Well), 21)
BXP12
BXP13 From **this and**:
BXP14

- BR17 and DEC9:

  BXP11: Initiates(Arrive(Fred, Well), At(Fred, Well), 21)

  From this, BXP10 and DEC9 we have:
  BXP12: HoldsAt(At(Fred, Well), 22)

- BR18 and DEC10:

  BXP13: Terminates(Arrive(Fred, Well), Walking(Fred), 21)

  From this, BXP10 and DEC10 we have:

  BXP14: ¬HoldsAt(Walking(Fred), 22)

BXP15 From BNE2, BXP12, BNI7*C* and BNC6*C* we have:
BXP16 **BXP15**: Happens(TurnOnTap(Fred, Tap), 22)
BXP17
BXP18 From **this and**:
BXP19

- BNC1 and BNC6*C*:

  BXP16: Terminates(TurnOnTap(Fred, Tap), Cond1, 22)

  From this, BIN1 and DEC10 we have:
  BXP17: ¬HoldsAt(Cond1, 23)

- BR2:

  BXP18: Initiates(TurnOnTap(Fred, Tap), On(Tap), 22)

  From this, BXP15 and DEC9 we have:

  BXP19: HoldsAt(On(Tap), 23)

BXP20 From BNI2 and BR72 we have:

BXP20: ¬HoldsAt(BucketFull(Bucket1), 0)

BXP21 From BR46, BXP20*C*, BNI8C and BXP19 we have:

BXP21: HoldsAt(Filling(Bucket1), 23)

BXP22 From BXP21, BXP19, BNI8*C* and BNP5 **and**:
BXP23

- BR47:

  BXP22: Happens(IncreasePouredLiquid(0.1), 23)

- BR74:

  BXP23: Happens(IncreaseLiquidLevel(Bucket1, 0.1), 23)

  (*this event keeps occurring at every timepoint as long as all the conditions hold - therefore the derived deductions will also be repeated*)

BXP24 From BNI2*C* and BR56 we have:
BXP24: HoldsAt(Level(Bucket1, 0), 0)

BXP25 From BXP23, BXP24*C* **and**:

BXP26

- BR33:

  BXP25: Initiates(IncreaseLiquidLevel(Bucket1, 0.1), Level(Bucket1, 0.1), 23)

  (*the event that this deduction has come from (BXP23) is repeated at every timepoint as long as all conditions hold - so this deduction is also repeated*)

- BR34:

  BXP26: Terminates(IncreaseLiquidLevel(Bucket1, 0.1), Level(Bucket1, 0), 23)

  (*the event that this deduction has come from (BXP23) is repeated at every timepoint as long as all conditions hold - so this deduction is also repeated*)

BXP27 From BXP23, BXP25 and DEC9 we have:

HoldsAt(Level(Bucket1, 0.1), 24)

The same deduction is repeatedly at every timepoint applied. Therefore we have an increasing level till timepoint 43. To save space, we do not repeat this.
….
[we eventually reach the following:]

BXP27: HoldsAt(Level(Bucket1, 2), 43)

BXP28 From BXP23, BXP26 and DEC10 we have:

¬HoldsAt(Level(Bucket1, 0), 24)

The same deduction is repeatedly at every timepoint applied. Therefore we have an obsolete increasing level till timepoint 43. To save space, we do not repeat this.
….
[we eventually reach the following:]

BXP28: ¬HoldsAt(Level(Bucket1, 1.9), 43)

BXP29 From BR55 and BNP2 and BXP27*V* we have:

BXP29: HoldsAt(BucketFull(Bucket1), 43)

BXP30	From BR46 and BXP29 we have: **BXP30**: ¬HoldsAt(Filling(Bucket1), 43)
BXP31 **BXP32**	From BXP22, BNI9*C* **and**: • BR38: **BXP31**: Initiates(IncreasePouredLiquid (0.1), AlreadyPoured(0.1), 23) (*the event that this deduction has come from (BXP22) is repeated at every timepoint as long as all conditions hold - so this deduction is also repeated*) • BR39: **BXP32**: Terminates(IncreasePouredLiquid (0.1), AlreadyPoured(0), 23) (*the event that this deduction has come from (BXP22) is repeated at every timepoint as long as all conditions hold - so this deduction is also repeated*)
BXP33	From BXP22, BXP31 and DEC9 we have: HoldsAt(AlreadyPoured(0.1), 24) The same deduction is repeatedly at every timepoint applied. Therefore we have an increasing poured level till timepoint 43. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP33**: HoldsAt(AlreadyPoured(2), 43)
BXP34	From BXP22, BXP32 and DEC10 we have: ¬HoldsAt(AlreadyPoured(0), 24) The same deduction is repeatedly at every timepoint applied. Therefore we have an obsolete increasing poured level till timepoint 43. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP34**: ¬HoldsAt(AlreadyPoured(1.9), 43)
BXP35	From BNI3*C* and BR72 we have: **BXP35**: ¬HoldsAt(BucketFull(Bucket2), 0)

BXP36 From BNI3*C* and BR56 we have:

BXP36: HoldsAt(Level(Bucket2, 0), 0)

BXP37
BXP38
BXP39
BXP40
BXP41
BXP42
BXP43
BXP44
BXP45

From BR52, BNI8*C*, BXP19*C*, BXP29, BNP9, BXP33*C*, BNI4*C*, BNI5*C*, BXP35 and BNP10 we have:

BXP37: Happens(Replace(Fred, Bucket1, Bucket2), 43)

From **this and**:

- BNI8*C* and BR29:

  BXP38: Happens(MoveAside(Fred, Bucket1, Tap), 43)

  From **this and**:

  - BR27:

    BXP39: Terminates(MoveAside(Fred, Bucket1, Tap), Beneath(Bucket1, Tap), 43)

    From this and BXP38 and DEC10 we have:

    BXP40: ¬HoldsAt(Beneath(Bucket1, Tap), 44)

  - BR28 and BXP12*C*:

    BXP41: Initiates(MoveAside(Fred, Bucket1, Tap), At(Bucket1, Well), 43)

    From this, BXP38 and DEC9 we have:

    BXP42: HoldsAt(At(Bucket1, Well), 44)

- BNI8C and BR25:

  BXP43: Happens(MoveUnder(Fred, Bucket2, Tap), 43)

  From this and BR24 we have:

  BXP44: Initiates(MoveUnder(Fred, Bucket2, Tap), Beneath(Bucket2, Tap),

	43) From this, BXP38 and DEC10 we have: **BXP45**: HoldsAt(Beneath(Bucket2, Tap), 44)
BXP46	From BR46, BXP35*C*, BXP45, BXP19*C* we have: **BXP46**: HoldsAt(Filling(Bucket2), 44)
BXP47 **BXP48** **BXP49** **BXP50** **BXP51** **BXP52**	From BXP46, BXP19*C*, BXP45 and BNP5 **and**: • BR47: **BXP47**: Happens(IncreasePouredLiquid(0.1), 44) (*this event keeps occurring at every timepoint as long as all the conditions hold - therefore the derived deductions will also be repeated*) From **this**, BXP33*C* **and**: BR38: **BXP48**: Initiates(IncreasePouredLiquid (0.1), AlreadyPoured(2.1), 44) BR38: **BXP49**: Terminates(IncreasePouredLiquid (0.1), AlreadyPoured(2), 44) • BR74: **BXP50**: Happens(IncreaseLiquidLevel(Bucket2, 0.1), 44) (*this event keeps occurring at every timepoint as long as all the conditions hold - therefore the derived deductions will also be repeated*)

	From **this**, BXP36*C* **and**: BR33: **BXP51**: Initiates(IncreaseLiquidLevel(Bucket2, 0.1), Level(Bucket2, 0.1), 44) BR34: **BXP52**: Terminates(IncreaseLiquidLevel(Bucket2, 0.1), Level(Bucket2, 0), 44)
BXP53	From BXP47, BXP48 and DEC9 we have: HoldsAt(AlreadyPoured(2.1), 45) The same deduction is repeatedly at every timepoint applied. Therefore we have an increasing poured level till timepoint 54. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP53**: HoldsAt(AlreadyPoured(3), 54)
BXP54	From BXP47, BXP49 and DEC10 we have: ¬HoldsAt(AlreadyPoured(2), 45) The same deduction is repeatedly at every timepoint applied. Therefore we have an obsolete increasing poured level till timepoint 43. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP54**: ¬HoldsAt(AlreadyPoured(2.9), 54)
BXP55	From BXP50, BXP51 and DEC9 we have: HoldsAt(Level(Bucket2, 0.1), 45) The same deduction is repeatedly at every timepoint applied. Therefore we have an increasing level till timepoint 43. To save space, we do not repeat this. …. [we eventually reach the following:]

	BXP55: HoldsAt(Level(Bucket2, 1), 54)
BXP56	From BXP50, BXP52 and DEC10 we have: ¬HoldsAt(Level(Bucket2, 0), 45) The same deduction is repeatedly at every timepoint applied. Therefore we have an obsolete increasing level till timepoint 43. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP56**: ¬HoldsAt(Level(Bucket2, 0.9), 54)
BXP57 **BXP58** **BXP59**	From BR53, BX46*C*, BNP9, BXP53, BNP4, BNP10, BXP12*C*, BXP19*C*, BXP45*C* we have: **BXP57**: Happens(TurnOffTap(Fred, Tap), 54) From **this and**: BR4: **BXP58**: Terminates(TurnOffTap(Fred, Tap), On(Tap), 54) From this BXP57 and DEC10 we have: **BXP59**: ¬HoldsAt(On(Tap), 55)
BXP60 **BXP61** **BXP62** **BXP63** **BXP64**	From BNE8, BXP12*C*, BXP27*C*, BXP55*C*, BNP9 and BNI11*C* we have: **BXP60**: Happens(PickUp(Fred, Bucket1), 54) From **this and**: • BR6: **BXP61**: Initiates(PickUp(Fred, Bucket1), Carrying(Fred, Bucket1), 54) From this and DEC9 we have: **BXP62**: HoldsAt(Carrying(Fred, Bucket1), 55) • BR7 and BXP42*C*: **BXP63**: Releases(PickUp(Fred, Bucket1), At(Bucket1, Well), 54)

From this and DEC11 we have:

BXP64: releasedAt(At(Bucket1, Well), 55)

BXP65 From BNE9, BXP12*C*, BXP27*C*, BXP55*C*, BNP9 and BNI12*C* we have:
BXP66
BXP67 **BXP65**: Happens(PickUp(Fred, Bucket2), 54)
BXP68
BXP69 From **this and**:

- BR6:

BXP66: Initiates(PickUp(Fred, Bucket2), Carrying(Fred, Bucket2), 54)

From this and DEC9 we have:

BXP67: HoldsAt(Carrying(Fred, Bucket2), 55)

- BR7 and BNI5*C*:

BXP68: Releases(PickUp(Fred, Bucket2), At(Bucket2, Well), 54)

From this and DEC11 we have:

BXP69: ReleasedAt(At(Bucket2, Well), 55)

BXP70 From BNE7, BXP62, BXP67, BXP27*C*, BXP55*C*, BNP9, BXP12*C* we have:
BXP71
BXP72 **BXP70**: Happens(GoFromTo(Fred, Well, Home), 55)
BXP73
BXP74 From **this and**:
BXP75

- BR12:

BXP76
BXP77 **BXP71**: Initiates(GoFromTo(Fred, Well, Home), Walking(Fred), 55)
BXP78

From this, BXP70 and DEC9 we have:

BXP72: HoldsAt(Walking(Fred), 56)

- BR13 and BXP12*C*:

  BXP73: Terminates(GoFromTo(Fred, Well, Home), At(Fred, Well), 55)

  From this, BXP70 and DEC10 we have:

  BXP74: ¬HoldsAt(AT(Fred, Well), 56)

- BR15:

  BXP75: Initiates(GoFromTo(Fred, Home, Well), Destination(Fred, Well), 0)

  From this, BXP70 and DEC9 we have:

  BXP76: HoldsAt(Destination(Fred, Home), 56)

- BNP1 and BR16:

  BXP77: Initiates(GoFromTo(Fred, Well, Home), DistanceToWalk(Fred, 20), 55)

  From this, BXP70 and DEC9 we have:

  BXP78: HoldsAt(DistanceToWalk(Fred, 20), 56)

BXP79 From BR44, BXP72, BXP78 and BNP8 we have:
BXP80
BXP81 **BXP79**: Happens(DecreaseDistanceToWalk(Fred, 1), 56)
(*this event keeps occurring at every timepoint as long as all the conditions hold - therefore the derived deductions will also be repeated*)

From **this and**:

- BR42:

  BXP80: Initiates(DecreaseDistanceToWalk(Fred, 1), DistanceToWalk(Fred,

19), 56)

- BR43:

  BXP81: Terminates(DecreaseDistanceToWalk(Fred, 1), DistanceToWalk(Fred, 20), 56)

BXP82 From BXP79, BXP80 and DEC9 we have:

HoldsAt(DistanceToWalk(Fred, 19), 57)

The same deduction is repeatedly at every timepoint applied. Therefore we have a decreasing distance till timepoint 21. To save space, we do not repeat this.
….
[we eventually reach the following:]

BXP82: HoldsAt(DistanceToWalk(Fred, 0), 76)

BXP83 From BXP79, BXP81 and DEC10 we have:

¬HoldsAt(DistanceToWalk(Fred, 20), 57)

The same deduction is repeatedly at every timepoint applied. Therefore we have an obsolete decreasing distance till timepoint 21. To save space, we do not repeat this.
….
[we eventually reach the following:]

BXP83: ¬HoldsAt(DistanceToWalk(Fred, 1), 76)

BXP84 From BR45 and BXP72*C* and BXP82 and BXP76*C* we have:
BXP85 **BXP84**: Happens(Arrive(Fred, Home), 76)
BXP86
BXP87 From **this and**:
BXP88

- BR17 and DEC9:

  BXP85: Initiates(Arrive(Fred, Home), At(Fred, Home), 76)

  From this, BXP84 and DEC9 we have:
  BXP86: HoldsAt(At(Fred, Home), 77)

- BR18 and DEC10:

	BXP87: Terminates(Arrive(Fred, Home), Walking(Fred), 76) From this, BXP84 and DEC10 we have: **BXP88**: ¬HoldsAt(Walking(Fred), 77)
BXP89	From BR56 and BXP27*C* we have: **BXP89**: ¬HoldsAt(BucketEmpty(Bucket1), 77)
BXP90	From BR56 and BXP55*C* we have: **BXP90**: ¬HoldsAt(BucketEmpty(Bucket2), 77)
BXP91 **BXP92** **BXP93**	From BNE3, BXP86, BXP62*C*, BXP89 and BNC7*C* we have: **BXP91**: Happens(MoveOver(Fred, Bucket1, Bucket3), 77) From this and BR5 we have: **BXP92**: Initiates(MoveOver(Fred, Bucket1, Bucket3), Beneath(Bucket3, Bucket1), 77) From this and DEC9 we have: **BXP93**: HoldsAt(Beneath(Bucket3, Bucket1), 78)
BXP94 **BXP95** **BXP96** **BXP97** **BXP98**	From BNE3, BXP86*C*, BXP62*C*, BXP89, BNC7*C* and BXP93 we have: **BXP94**: Happens(PourFromTo(Fred, Bucket1, Bucket3), 78) From **this and**: • BR20: **BXP95**: Initiates(PourFromTo(Fred, Bucket1, Bucket3), PouringFromTo(Fred, Bucket1, Bucket3), 78) From this and DEC9 we have: **BXP96**: HoldsAt(PouringFromTo(Fred, Bucket1, Bucket3), 79) • BNC2 and BNC7*C*: **BXP97**: Terminates(PourFromTo(Fred, Bucket1, Bucket3), Cond2, 78)

	From this and DEC10 we have: **BXP98**: ¬HoldsAt(Cond2, 79)
BXP99	From BR55 and BNI6 we have: **BXP99**: ¬HoldsAt(BucketFull(Bucket3), 0)
BXP100	From BR46, BXP99*C*, BXP96, BXP27*C* we have: **BXP100**: HoldsAt(Filling(Bucket3), 79)
BXP101 **BXP102** **BXP103** **BXP104** **BXP105** **BXP106**	From BXP100, BXP96, BNP11, BXP27*C* **and**: • BR48: **BXP101**: Happens(IncreaseLiquidLevel(Bucket3, 0.1), 79) (*this event keeps occurring at every timepoint as long as all the conditions hold - therefore the derived deductions will also be repeated*) From **this**, BNI6*C* **and**: BR33: **BXP102**: Initiates(IncreaseLiquidLevel(Bucket3, 0.1), Level(Bucket3, 2.1), 79) BR34: **BXP103**: Terminates(IncreaseLiquidLevel(Bucket3, 0.1), Level(Bucket3, 2), 79) • BR75: **BXP104**: Happens(DecreaseLiquidLevel(Bucket1, 0.1), 79) (*this event keeps occurring at every timepoint as long as all the conditions*

	hold - therefore the derived deductions will also be repeated) From **this**, BXP27*C* **and**: BR35: **BXP105**: Initiates(DecreaseLiquidLevel(Bucket1, 0.1), Level(Bucket1, 1.9), 79) BR37: **BXP106**: Terminates(DecreaseLiquidLevel(Bucket1, 0.1), Level(Bucket1, 2), 79)
BXP107	From BXP101, BXP102 and DEC9 we have: HoldsAt(Level(Bucket3, 2.1), 80) The same deduction is repeatedly at every timepoint applied. Therefore we have an increasing level till timepoint 99. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP107**: HoldsAt(Level(Bucket3, 4), 99)
BXP108	From BXP101, BXP103 and DEC10 we have: ¬HoldsAt(Level(Bucket3, 2), 80) The same deduction is repeatedly at every timepoint applied. Therefore we have an obsolete increasing level till timepoint 99. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP108**: ¬HoldsAt(Level(Bucket3, 3.9), 99)
BXP109	From BXP104, BXP105 and DEC9 we have: HoldsAt(Level(Bucket1, 1.9), 80) The same deduction is repeatedly at every timepoint applied. Therefore we have a decreasing level till timepoint 99. To save space, we do not repeat this. …. [we eventually reach the following:]

	BXP109: HoldsAt(Level(Bucket1, 0), 99)
BXP110	From BXP104, BXP106 and DEC10 we have: ¬HoldsAt(Level(Bucket1, 2), 80) The same deduction is repeatedly at every timepoint applied. Therefore we have an obsolete decreasing level till timepoint 99. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP110**: ¬HoldsAt(Level(Bucket1, 0.1), 99)
BXP111	From BXP109 and BR56 we have: **BXP111**: HoldsAt(BucketEmpty(Bucket1), 99)
BXP112	From BR46 and BXP110 we have: **BXP112**: ¬HoldsAt(Filling(Bucket3), 99)
BXP113 **BXP114** **BXP115**	From BNE5, BXP86*C*, BXP67*C*, BXP111, BXP90*C* and BNC8*C* we have: **BXP113**: Happens(MoveOver(Fred, Bucket2, Bucket3), 99) From this and BR5 we have: **BXP114**: Initiates(MoveOver(Fred, Bucket1, Bucket3), Beneath(Bucket3, Bucket1), 99) From this and DEC9 we have: **BXP115**: HoldsAt(Beneath(Bucket3, Bucket1), 100)
BXP116 **BXP117** **BXP118** **BXP119**	From BNE6, BXP86*C*, BXP67*C*, BXP111, BXP90*C*, BXP115 and BNC8*C* we have: **BXP116**: Happens(PourFromTo(Fred, Bucket2, Bucket3), 100) From **this and**: • BR20: **BXP117**: Initiates(PourFromTo(Fred, Bucket2, Bucket3), PouringFromTo(Fred, Bucket2, Bucket3), 100)

	From this and DEC9 we have: **BXP118**: HoldsAt(PouringFromTo(Fred, Bucket2, Bucket3), 101) • BNC3 and BNC8*C*: **BXP118**: Terminates(PourFromTo(Fred, Bucket2, Bucket3), Cond3, 100) From this and DEC10 we have: **BXP119**: ¬HoldsAt(Cond3, 101)
BXP120	From BR55 and BXP107 we have: **BXP120**: ¬HoldsAt(BucketFull(Bucket3), 99)
BXP121	From BR46, BXP120*C*, BXP118, BXP55*C* we have: **BXP121**: HoldsAt(Filling(Bucket3), 101)
BXP122 **BXP123** **BXP124** **BXP125** **BXP126** **BXP127**	From BXP121, BXP118, BNP12, BXP55*C* **and**: • BR48: **BXP122**: Happens(IncreaseLiquidLevel(Bucket3, 0.1), 101) (*this event keeps occurring at every timepoint as long as all the conditions hold - therefore the derived deductions will also be repeated*) From **this**, BXP107*C* **and**: BR33: **BXP123**: Initiates(IncreaseLiquidLevel(Bucket3, 0.1), Level(Bucket3, 4.1), 101) BR34: **BXP124**: Terminates(IncreaseLiquidLevel(Bucket3, 0.1), Level(Bucket3, 4), 101)

- BR75:

  BXP125: Happens(DecreaseLiquidLevel(Bucket2, 0.1), 101)

  (*this event keeps occurring at every timepoint as long as all the conditions hold - therefore the derived deductions will also be repeated*)

  From **this**, BXP55*C* **and**:

  BR35:

  BXP126: Initiates(DecreaseLiquidLevel(Bucket2, 0.1), Level(Bucket2, 0.9), 101)

  BR37:

  BXP127: Terminates(DecreaseLiquidLevel(Bucket2, 0.1), Level(Bucket2, 1), 101)

BXP128	From BXP122, BXP123 and DEC9 we have: HoldsAt(Level(Bucket3, 4.1), 102) The same deduction is repeatedly at every timepoint applied. Therefore we have an increasing level till timepoint 111. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP128**: HoldsAt(Level(Bucket3, 5), 111)
BXP129	From BXP122, BXP124 and DEC10 we have: ¬HoldsAt(Level(Bucket3, 4), 102) The same deduction is repeatedly at every timepoint applied. Therefore we have an obsolete increasing level till timepoint 111. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP129**: ¬HoldsAt(Level(Bucket3, 4.9), 111)
BXP130	From BXP125, BXP126 and DEC9 we have:

	HoldsAt(Level(Bucket2, 0.9), 102) The same deduction is repeatedly at every timepoint applied. Therefore we have a decreasing level till timepoint 111. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP130**: HoldsAt(Level(Bucket2, 0), 111)
BXP131	From BXP104, BXP106 and DEC10 we have: ¬HoldsAt(Level(Bucket2, 1), 102) The same deduction is repeatedly at every timepoint applied. Therefore we have an obsolete decreasing level till timepoint 111. To save space, we do not repeat this. …. [we eventually reach the following:] **BXP131**: ¬HoldsAt(Level(Bucket2, 0.1), 111)
END OF PROOF	BXP128 is the proof of our proposition: **HoldsAt(Level(Bucket3, 5), 111)** *The level of Bucket3 will be 5 litres at timepoint 111.*

3.5 Critical remarks

In the framework BR7:

$$\text{HoldsAt}(\text{At}(b, l), t) \Rightarrow \text{Releases}(\text{PickUp}(a, b), \text{At}(b, l), t)$$

states that the action of picking up an item releases the location of that item from Inertia. This means that, unlike fluents that are not released from Inertia, for released fluents we cannot use DEC5 to deduce that they hold for the timepoints afterwards. By picking up an item, the location of that item will that of its carrier which is only commonsense. Therefore that item does not have a specific location. This is represented on our framework by BR7. Inertia is restored in a released fluent by *initiating* or *terminating* that fluent. In our framework BR9 indicates:

$$\text{HoldsAt}(\text{At}(a, l), t) \Rightarrow \text{Initiates}(\text{PutDown}(a, b), \text{At}(b, l), t)$$

although this formula is not employed in our scenario (not in our natural language scenario, Fred never puts down the buckets), our framework supports this. So if we were to modify the scenario by putting down the buckets in the middle of the way, that would be perfectly fine in our framework: the buckets would have a location - where they were dropped - and they could be picked up again. Once again this shows the elaboration tolerance and robustness of the Event Calculus.

Our framework makes use of *Beneath* and *Over* as two fluents. BR54, however, defines the relationship between these two fluents:

$$\text{HoldsAt}(\text{Beneath}(x, y), t) \Leftrightarrow \text{HoldsAt}(\text{Over}(y, x), t)$$

this shows these fluents have an exclusive relationship saying if x is beneath y, then y is over x. We also make use of two actions *MoveUnder* and *MoveOver*. We need to emphasise while it is good practice to use *Beneath* and *Over* interchangeably (taking into account to swap *x* and *y*), it is not allowed to do so

with *MoveUnder* and *MoveOver*. This is due to the commonsense fact that while it might be possible to squeeze something under another, it is not always possible to do so with moving something over another. For instance, in our scenario it is ok to use *MoveUnder* for moving a bucket under a tap *or* another bucket, it is not ok to use *MoveOver* for moving a tap over a bucket (but it is ok to move a bucket over another).

Our representation of the domain description can handle the scenario and also variations to the scenario. There is room for further improvement however and making it more robust. For instance, in the calculating of wasted liquid, the following concurrency is not handled: the tap is left on (with no bucket underneath it, so liquid being wasted); and we are pouring water from a bucket into another, and the receiving bucket gets full and starts spilling (wasting liquid). This concurrency is however trivial and can be handled by small modifications. It was of no interest to this scenario and that is the reason why it was not handled.

Our representation initially did not handle a small change to the scenario: if Fred stops in the middle of the way for a moment and then continues on his way. With modifications made to the representation, it calculates the distance between current location of Fred and the destination.

Shin and Davis representation (Appendix C) does not handle a scenario in which a bucket is pushed underneath an open tap (say by a dog). In their representation the only way to increase the water level of a bucket is for an agent to turn on the tap (and initiate the Filling process). But in our representation, this is represented as a state constraint representing rules of the world. So if a bucket gets pushed under an open tap by any means, it will start filling regardless of the reason why and by whom it was pushed there. The effect of turning on the tap is only for the

tap to be on, not filling the bucket. So if we were to represent this in Shin and Davis representation, we would need to change at least one action (turn on tap) and add new actions. However this is not the case in our representation of the problem in EC due to the elaboration tolerance of EC.

New additions or modifications can be made to our representation easily. For instance, if we want to say that an agent can only pick up a bucket if it is not "too heavy", we could represent this by adding:

$$\text{Happens}(\text{PickUp}(\text{agent}, \text{bucket}), t) \Rightarrow$$

$$\neg\text{HoldsAt}(\text{TooHeavy}(\text{bucket}), t).$$

And "TooHeavy" can be defined in another assertion (yet again, elaboration tolerance).

Our framework is able to calculate the amount of wasted liquid. Wasted liquid is defined as if a tap is left on with no empty bucket underneath it or pouring from a bucket into another bucket which is full. However, it would be interesting to integrate the Liquid Theory of Davis [9] and represent what happens to the liquid being wasted: if it is pouring to a full bucket how it will move to the sides of the bucket and slowly flows down. This would be an interesting follow up of this work.

By alterations to our presented scenario we can show how wasted liquid can also be calculated: If Fred moves the bucket underneath the tap aside, opens the tap and after 5 timepoints put the bucket back underneath the tap, the framework calculates the liquid being wasted for those 5 timepoints.

Shin and Davis's framework is less developed in this sense as the condition they impose is by turning on the tap, if bucket is as at the same location as the tap, then a filling process is initiated (turning off the tap works in a similar fashion).

Therefore adding a new feature to the framework, such as calculating wasted liquid, needs major changes to the existing formulas; as in our framework, these formulas work separately. If we remove the wasted liquid calculating formulas, our proved proposition is proved yet again with no change. Or if the TurnOnTap or TurnOffTap are changed or omitted, our framework does not suffer. Although omitting these events might result in never being able to calculate the wasted liquid from the tap (but that is only logical, if a tap cannot be turned on, then there will not be any wasted liquid from it), our wasted liquid formulas are still valid for pouring from a bucket into another. This shows elaboration tolerance of the EC and our representations in comparison to that of Shin and Davis.

Chapter 4

Commonsense Reasoning with the Event Calculus

4.1 Acquisition of world knowledge

As described in Chapter 1 and shown in an example in Chapter 2, use of world knowledge is essential to make commonsense inferences. It is important, however, to determine to what extent we need this background knowledge. For example consider this scenario: Fred leaves home at 8am, stops by the shop at 8:30 and arrives at work at 9am. In a scenario in which Fred starts working at 9 and leaves his office at 5pm, we are not concerned with where Fred has been before starting his job (from the card reader point of view at work). But if we are interested to find out why the fuel of Fred's car is less than usual, we would like to know if he has stopped by any place on his way to work. In another scenario of the same world, if Fred's car is broken, we might want to know how he got to his work, but we did not care about the means in the previous scenario.

Let us make this clearer by a more detailed example. The Fluid Theory of Davis [9] defines a commonsense framework in which movements of the liquid inside a pitcher pouring the liquid inside a pail is formalised. His theory deals with the movement of liquid from a commonsense perspective, meaning it does not go into

that much detail of the movements of molecules of the liquid over each other; yet it is not so abstract to take *Happens(PourFromTo(Pitcher, Pail), t)* as an event of the theory either. Finding the correct level of abstraction is very important to make inferences possible, especially if this external knowledge is being imported to the reasoning system. An example which we worked on and will give more details later in the report is of an agent carrying buckets and pouring liquid from a bucket into another. In that example, we have not gone into as much detail as in Davis's theory therefore *Happens(PourFromTo(Pitcher, Pail), t)* would suffice there. We need to identify, for a given scenario, what knowledge we need to automatically import for automated reasoning.

4.2 Encoding from the Event Calculus

An EC problem can be proved using a SAT solver. A SAT problem is a specific constraint satisfaction problem in which every variable ranges over the values {true, false}. We describe how we can encode EC problems into propositional calculus problems.

In this thesis, we use two methods to carry out reasoning with Event Calculus:

- Manual theorem proving: the examples of this can be seen in the literature such as the egg cracking scenarios tackled by Shanahan [6] and Morgestern [13]. We also present some examples we have represented and manually proved in this report in a later chapter.
- Automated theorem proving: Shanahan and Witkowski [64] proposed that event calculus planning be carried out using satisfiability. They presented

a method for encoding EC planning problems as satisfiability problems. Mueller [65] extended their method for a larger subset of EC.

The domain description of an EC description is as follows:

CIRC[$\sum$; Initiates, Terminates, Releases] $\wedge$ CIRC[Δ; Happens] $\wedge$ Ψ $\wedge$ Γ $\wedge$ DEC

In which:

- $\sum$ is a conjunction of Initiates, Terminates and Releases formulas
- Δ is a conjunction of Happens and temporal ordering formulas (t1<t2)
- Ψ is a conjunction of state constraints
- Γ is a conjunction of HoldsAt and ReleasedAt formulas

Satisfiability solvers (SAT solvers) take a set of Boolean variables and a propositional formula over those variables as input and as output produce zero or more models or satisfying truth assignments for the variables such that the formula is true. A complete SAT solver produces all satisfying truth assignments. As mentioned before, SAT solvers take a propositional formula in the conjunctive normal form (CNF) which is a conjunction of clauses; where each clause is a disjunction of literals; where each literal is a variable or a negated variable.

We start by introducing some definitions over the EC:

Def. 4.2.1: a comparison is a formula of the form $t1<t2$, $t1\leq t2$, $t1=t2$, $t1\geq t2$, $t1>t2$ or $t1\neq t2$ where *t1* and *t2* are terms.

Def. 4.2.2: If *t* is a variable then a condition over *t* is defined:

- A comparison is a condition over *t*
- If *f* is a term then *HoldsAt(f, t)* and $\neg$*HoldsAt(f, t)* are conditions over *t*

- If c_1 and c_2 are conditions over t, then $c_1 \wedge c_2$ and $c_1 \vee c_2$ are conditions over t. If v is a variable and c is a condition over t, then $\exists v\ c$ is a condition over t

Def. 4.2.3: If π is the predicate symbol *Initiates*, *Terminates* or *Releases*, then a π effect axiom is a formula of the form $\forall e, f, t[\Theta(e, f, t) \Rightarrow \pi(e, f, t)]$, where $\Theta(e, f, t)$ is a condition over t with only e, f and t free.

Def. 4.2.4: A π effect description is a collection of π effect axioms written as a single, logically equivalent π effect axiom of the form $\forall e, f, t[\Theta(e, f, t) \Rightarrow \pi(e, f, t)]$, where $\Theta(e, f, t)$ is a condition over t with only e, f and t free.

- Let $\sum_{init}$ be the Initiates effect description $\forall$e, f, t[Θ_{init}(e, f, t) $\Rightarrow$ Initiates(e, f, t)].
- Let $\sum_{term}$ be the Terminates effect description $\forall$e, f, t[Θ_{term}(e, f, t) $\Rightarrow$ Terminates(e, f, t)].
- Let $\sum_{rel}$ be the Releases effect description $\forall$e, f, t[Θ_{rel}(e, f, t) $\Rightarrow$ Releases(e, f, t)].

Def. 4.2.5: A trigger axiom is a formula of the form $\forall e, t[Y(e, t) \Rightarrow Happens(e, t)]$ where $Y(e, t)$ is a condition over t with only e and t free.

Def. 4.2.6: A trigger description is a collection of trigger axioms written as a single, logically equivalent trigger axiom of the form $\forall e, t[Y(e, t) \Rightarrow Happens(e, t)]$ where $Y(e, t)$ is a condition over t with only e and t free.

Def. 4.2.7: An event occurrence is a formula of the form $Happens(e, t)$ where e is an event ground term and t is a timepoint ground term.

Def. 4.2.8: An event occurrence description is a collection of event occurrences written as a single, logically equivalent trigger axiom of the form $\forall e, t[Y(e, t) \Rightarrow Happens(e, t)]$ where $Y(e, t)$ is a condition over t with only e and t free.

Def. 4.2.9: An event description is a trigger description and an event occurrence description written as a single, logically equivalent trigger axiom of the form $\forall e, t[Y(e, t) \Rightarrow Happens(e, t)]$ where $Y(e, t)$ is a condition over t with only e and t free.

Let Δ be an event description.

Def. 4.2.10: A state constraint is a formula of the form $c_1 \Rightarrow c_2$ *or* $c_1 \Leftrightarrow c_2$ where c_1 and c_2 are conditions over some variable t.

Let Ψ be a conjunction of state constraints.

Def. 4.2.11: A state description is a conjunction of formulas of the form $HoldsAt(f, t)$, $\neg HoldsAt(f, t)$, $ReleasedAt(f, t)$ or $\neg ReleasedAt(f, t)$ where f is a fluent ground term and t is a timepoint ground term.

Let Γ be a state description.

We will use circumscription which was formally introduced in Chapter 2.

EC and DEC contain atoms involving *Initiates*, *Terminates* and *Releases* which lead to a large number of ground atoms. For instance, $Initiates(e, f, t)$ gives rise to $E*F*T$ ground atoms where E is the number of events, F is the number of fluents and T is the number of timepoints. In order to eliminate such atoms, we expand DEC by performing the following substitutions:

- Initiates(e, f, t) replaced by $\Theta_{init}(e, f, t)$
- Terminates(e, f, t) replaced by $\Theta_{term}(e, f, t)$
- Releases(e, f, t) replaced by $\Theta_{rel}(e, f, t)$

For instance, if $\sum_{init}$ is:

[e = Hold(a, o) $\wedge$ f = Holding(a, o)] $\Rightarrow$ Initiates(e, f, t)

Then we replace DEC9 of Appendix B with:

[Happens(e, t) $\wedge$ [e = Hold(a, o) $\wedge$ f = Holding(a, o)]] $\Rightarrow$ HoldsAt(f, t+1)

The next step is to circumscribe Happens. So using the methods explained in Chapter 2, we compute: *CIRC[Δ; Happens]*. We then conjoin Ψ, Γ the new substituted DEC and *CIRC[Δ; Happens]*.

Then we instantiate quantifiers by replacing $\forall x\ \Phi(x)$ with $\wedge_i\ \Phi(x_i)$ and $\exists x\ \Phi(x)$ with $\vee_i\ \Phi(x_i)$ where x_i are constants of the sort of x. This gives us a propositional calculus formula.

We simplify the formula using standard techniques [44].

We construct a one-to-one and onto map B that maps the ground atoms of the formula to Boolean variables. We construct an inverse map B^{-1} from B for converting the results back into the EC formulas.

We construct a formula to pass to the SAT solvers by replacing each ground atom u in the formula with $B(u)$. In order to perform model finding we give the formula to a SAT solver. We can then decode the satisfying truth assignments produced by the solver by applying B^{-1} inverse map.

4.3 An example of a domain description

We convert and feed the following domain description to a SAT solver and see how the SAT solver will find a model based on the model at the initial time.

We have an Initiates effect description that states that if a person holds an object, then the person will be holding the object (example adopted from [65]):

Ex3.4.1 $[e = Hold(a, o) \wedge f = Holding(a, o)] \Rightarrow Initiates(e, f, t)$

There is a state description that says at timepoint 0 agent *A1* is not holding object *O1* and this fact is not released from the commonsense law of Inertia:

Ex3.4.2 $\neg HoldsAt(Holding(A1, O1), 0) \wedge \neg RleasedAt(Holding(A1, O1), 0)$

There is an event description that says at timepoint 0 agent *A1* holds object *O1*:

Ex3.4.3 $[e = Hold(A1, O1) \wedge t = 0) \Rightarrow Happens(e, t)$

We assume that 0 and 1 are the only constants of the timepoint sort, *A1* is the only constant of the sort *Agent* and *O1* is the only constant of the sort *Object*. The conjunctive normal form from encoding this domain description will consist of 10 clauses.

We have the following clauses for Ex3.4.2:

C_1: $\neg HoldsAt(Holding(A1, O1), 0)$.

C_2: $\neg ReleasedAt(Holding(A1, O1), 0)$.

From Ex3.4.1 and expansion of DEC5,6, 7,9 and DEC12 we have:

C_3: $ReleasedAt(Holding(A1, O1), 1) \vee HoldsAt(Holding(A1, O1), 1) \vee \neg HoldsAt(Holding(A1, O1), 0)$.

C_4: $Happens(Hold(A1, O1), 0) \vee HoldsAt(Holding(A1, O1), 0) \vee ReleasedAt(Holding(A1, O1), 1) \vee \neg HoldsAt(Holding(A1, O1), 1)$.

C_5: $Happens(Hold(A1, O1), 0) \vee ReleasedAt(Holding(A1, O1), 1) \vee \neg HoldsAt(Holding(A1, O1), 0)$.

C_6: $ReleasedAt(Holding(A1, O1), 0) \vee \neg ReleasedAt(Holding(A1, O1), 1)$.

C_7: HoldsAt(Holding(A1, O1), 1) ∨ ¬Happens(Hold(A1, O1), 0).

C_8: ¬Happens(Hold(A1, O1), 0) ∨ ¬ReleasedAt(Holding(A1, O1), 1).

Formulas DEC10 and DEC11 are trivially satisfied since there are no *Terminates* or *Releases* formulas in the domain description.

The circumscription of *Happens* results in the following clauses:

C_9: ¬Happens(Hold(A1, O1), 1).

C_{10}: Happens(Hold(A1, O1), 0).

We construct our *B* map from ground atoms to Boolean variables - assigning each axiom a number:

Happens(Hold(A1, O1), 0) → 1

HoldsAt(Holding(A1, O1), 0) → 2

ReleasedAt(Holding(A1, O1), 0) → 3

Happens(Hold(A1, O1), 1) → 4

ReleasedAt(Holsing(A1, O1), 1) → 5

HoldsAt(Holding(A1, O1), 1) → 6

Converting the clauses into the standard DIMACS format for satisfiability problems [43] results in Figure 4.1 which represents the translation of our clauses (C_1 to C_{10}) to their equivalent Boolean variables we just assigned (1 to 6).

Clauses	Equivalent Boolean variables in the clause				
C_1	-2	0			
C_2	-3	0			
C_3	5	6	-2	0	
C_4	1	2	5	-6	0
C_5	1	5	-3	0	
C_6	3	-5	0		
C_7	6	-1	0		
C_8	-1	-5	0		
C_9	-4	0			
C_{10}	1	0			

Figure 4.1 – Converted clauses C_1 to C_{10} into CNF using Boolean variables 1 to 6

A negated variable *v* is represented by *–v* and a non-negated variable *v* is represented by *v*. Each row in Figure 4.1 is a sequence of numbers and terminates with the number 0.

So for instance clause C_5:

Happens(Hold(A1, O1), 0) ∨

ReleasedAt(Holding(A1, O1), 1) ∨

¬HoldsAt(Holding(A1, O1), 0).

is represented respectively as:

1

5

-3

0 *(indicates the termination of the sequence)*

Invoking a SAT solver on this problem will produce one model as output:

1 -2 -3 -4 -5 6

By applying the inverse map, B^{-1}, we get:

Happens(Hold(A1, O1), 0). (*1*)

¬HoldsAt(Holding(A1, O1), 0). (-*2*)

¬ReleasedAt(Holding(A1, O1), 0). (-*3*)

¬Happens(Hold(A1, O1), 1). (-4)

¬ReleasedAt(Holding(A1, O1), 1). (-*5*)

HoldsAt(Holding(A1, O1), 1). (*6*)

Our proposition is proved by a SAT solver successfully.

Table 4.1 shows a list of commonsense encoders which make use of SAT solvers.

Description	Technique	Reasoning Type
Event Calculus Planner [62, 63]	Abductive logic programming	Abduction
Event calculus planner [64]	Propositional satisfiability	Abduction
Discrete Event Calculus Reasoner [65, 66]	Propositional satisfiability	Deduction, Abduction, Postdiction, Model Finding
Discrete event calculus theorem prover [67, 68, 69]	First-order logic automated theorem proving	Deduction

Table 4.1: Commonsense encoders which make use of SAT solvers

Chapter 5

Automated Reasoning Methods for the Event Calculus

5.1 Introduction

Over the past decades, large amounts of time and resources have been dedicated to research in Automated Reasoning (AR) methods. Generally, the methods which have specifically dealt with the Event Calculus are the following:

- SAT Solving: there are many implementations which use a SAT solver to perform reasoning on the Event Calculus. For instance, E-RES [71] system which deals with Language E and Modular E [70] (essentially EC representations) use RelSat. Mueller's DECReasoner [72] also ultimately uses three different SAT solvers (RelSat, Walksat and MiniSat) to perform reasoning. We elaborate how a SAT solver works in this chapter.
- Logic Programming: There are implementations of reasoning on EC using logic programming. For instance, Shanahan's Abductive Event Calculus

Planner [62] or ACLP [74]. We describe Constraint Programming in more detail in Appendix E but will not focus any more on it since it is not in the scope of this report. In the literature there is more usage of SAT solving methods over logic programming. For performing reasoning over EC, Mueller argues for superiority and high efficiency of SAT solvers and their success in the International Planning Competition from 2004 to 2006 and their ability to perform abduction, deduction and model finding. It should also be noted that since SAT solvers use propositional logic which is decidable.

- First-order automated theorem proving: in the literature, there has only been one attempt to use a first-order theorem prover on EC problems [73]. However Mueller and Sutcliffe show that SAT solving method is much more efficient by taking significantly lower time to solve some benchmark problems. Although they conclude ATP has the benefit that the derivation retains meaning and can be understood by humans, it not only takes longer to solve the problems, in many cases it fails to terminate. SAT solving method results, however, can also be humanly understandable by reverse mapping, as discussed before in this report.

In this chapter we will explain Propositional satisfiability solving (SAT solving) in more detail; an AR technology that has been developing and widely used in the industry during the recent decades.

5.2 SAT Solving

SAT solving approach provides a generic language that can be used to express complex problems such as scheduling or hardware verification. It employs a general-purpose algorithm to automatically search for solutions to the problem at hand. This algorithm is called a *solver*. Where a general purpose algorithm (i.e. a solver) is at use, the need is avoided to redevelop new algorithmic solutions from scratch for each application where intelligent search is needed.

There are generally two categories of constraint solving methods:

- Incomplete methods: In this method, the aim is to find solutions by means of heuristics without exhaustively covering the whole search space. Therefore, it is most often the case that these methods are unable to determine that no solution exists. There is, however, a time threshold after which the search is stopped and "no solution" is generated. In this case it is not possible to tell if the solutions were actually missed in the search or the problem is indeed unsatisfiable. There are mainly two categories of methods in this approach:
    - Population based algorithms: a list of individuals which typically correspond to points of the search space is maintained, and iteratively modified; with the goal of an individual which satisfies all constraints. An example of this type would be the ant colony optimisation algorithms [38] and other swarm-based collective intelligence algorithms.
    - Local search methods: a unique point at every timepoint is considered until a solution is reached by exploring the

neighbourhood of the current selected point, moving stochastically along the search space.

- Complete methods: In this method, the aim is to find solutions by exploring the entire search space by means of backtrack searches and local reasoning at each node to prune away certain branches. Exhaustive enumeration of search space would, however, be too costly. Therefore pruning techniques are employed to determine certain areas of the search space do not contain a solution, hence better efficiency.

Satisfiability is the problem of determining if the variables of a given Boolean formula can be assigned in such a way to make the formula evaluate to true (satisfiable) or to determine whether no such assignments exist which implies that the function expressed by the formula is false for all possible variable assignments (unsatisfiable). Therefore we can define a SAT solver as: given a propositional formula on a set of Boolean variables, a SAT solver determines if there exists an assignment of the variables such that the formula evaluates to true or proves no such assignment exists.

A very simple yet concrete example of the kinds of problems handled by SAT solvers is the following:

For each vertex (for instance V), we have 3 Boolean variables (V_1, V_2 and V_3) which will be true if one of the colours 1, 2, or 3 is assigned to the vertex. There is also the constraint that each vertex must be assigned exactly one colour. In other words, we could rewrite this constraint as: At least one colour is assigned to the vertex and the vertex cannot have two colours.

So for V we have:

$V_1 \vee V_2 \vee V_3$ $\neg V_1 \vee \neg V_2$ $\neg V_2 \vee \neg V_3$ $\neg V_3 \vee \neg V_1$

These constraints are clauses. A clause is a disjunction of literals where a literal is either a positive or a negative instance of a variable. SAT solvers take the input in the format of conjunctive normal form (or CNF). A CNF is a conjunction of one or more clauses.

Now let us make the problem more interesting by introducing more vertices (for instance add *W, X, Y* and *Z*) and the constraint that each edge must have a different colour to its extremities.

Then we will have:

$\neg V_1 \vee \neg W_1$ $\neg V_2 \vee \neg W_2$ $\neg V_3 \vee \neg W_3$ ...

So far, we have 15 variables and 41 clauses to express this simple problem. SAT is quite a low-level language. Were we to add more constraints or simply add another colour to the problem, say 4, the number of clauses and variables would significantly grow higher. Therefore for practical reasons, the formulas are typically generated from automatic translation of a problem, instead of handwriting them.

In general, there are two ways to use a SAT solver in an application. One way is for the application to generate a Boolean formula and ask a SAT solver to determine its satisfiability and produce a satisfying assignment (if any). This is called the eager approach [39]. The other way is for the application to reduce the problem to a series of related SAT queries which are incrementally solved by the SAT solver. Subsequent SAT queries are then dynamically generated based on the results of previous queries (the lazy approach [39]).

SAT solving approach is essentially a black box. Once the problem is stated, the solver is to find a solution without the need for any external interaction. However, in SAT, constraints (clauses) are expressed in an indirect way. That is although they can be used to state complex problems, once translated into CNF which is very low-level, the internal structure of the problems are lost and the clauses would not make much structural sense to a human eye. But the advantage of this simple representational language is that all the effort is focused on a single representation resulting in more optimised datastructures and efficient implementation of the reasoning and performing deductions on CNF formulas. SAT solvers are generally used as a target utility by higher-level reasoning tools which automatically translate other formalisms into CNF formulas.

Generally, a SAT solver only needs to answer true or false depending on the satisfiability of the formula. Some SAT solvers may also produce a satisfying assignment (a model) if the formula is satisfiable. Arithmetics are not native to SAT solvers, since they only deal with CNF formulas. Therefore they need to be represented in Boolean logic. One simple way is, given x as a numerical variable, to create a Boolean variable B_i for each possible value *i* of *x*. *Bi* will be true if and only if $x = i$. There needs to be a constraint stating that only one of the *Bi* variables is true at any given moment.

5.2.1 Branching

In SAT, a common way to solve problems is to use backtracking search methods, most of which are variations of the DPLL algorithm [37]. In SAT, only two choices are possible for each variable, making variable selection and assignment

very important since different branching decisions lead to very different search trees being explored. At each step DPLL assigns one of the two possible values to a variable, applying a restricted form of resolution called Unit Propagation (UP). If there is only one literal present in a unary clause, then UP automatically sets it to True as it must be True. Having done so, UP also reduces the size of any clause containing the opposite of this literal by one as that instance is definitely False. This process is repeated until no more unary clauses appear or an empty clause is derived. In the latter case, the other value of the currently selected variable will be tried. If this causes an empty clause as well (called a dead-end), then the algorithm backtracks to the previously assigned variable.

Using UP, the SAT problem is reduced to fewer and smaller clauses, while dead-ends are discovered at earlier stages. Therefore, the length of a clause C is essentially the number of unassigned literals in C.

The heuristics of choosing values are more or less arbitrary, however. They are usually based on some obvious statistics. The solver has to search the entire search space in one way or another, therefore discovering conflicts and dead-ends as early as possible is vital to higher and more efficient performance. Although branching heuristics are important in determining the efficiency of SAT solvers, they must also be cheap to evaluate. A heuristic that requires iterating through all the clauses of the problem is not affordable on (typically) large problems.

Moskewicz et al. proposed a branching heuristic called Variable State Independent Decaying Sum (VSIDS) [29]. VSIDS keeps a score for each of the two Boolean values of a SAT variable (True or False). The scores are initially the

number of occurrences of the corresponding literals in the original CNF formula. But as the search progresses, additional clauses and literals are added to the clause database. The score of a literal is increased by a constant value whenever a newly added clause contains this literal.

All the scores are periodically divided by a constant (the decaying-scores effect). VSIDS does this in order to overcome the problems encountered by older solvers; namely, the significant overhead of recalculating all the free variables at every branching point; and the dependability of the counts on the current state of the solver (as in GRASP [26]).

In VSIDS, however, more recently added literals have higher weight for calculating a score which is essentially a literal count. VSIDS will then branch on the free literal with the highest score.

Scores in VSIDS are very cheap to maintain because they are variable-state independent (i.e. unrelated to the current variable assignment). In VSIDS, the scores are not static statistics; they take the search history into consideration. Variables that are recently active will have preference to be branched on. The activity of a variable is determined by the score that is related to the literal's occurrences. The decaying-scores effect mentioned above helps to keep the focus on recent events.

Search-based solvers detect the consequences of the assignments imposed by the branching heuristics through deduction mechanisms called *pruning*.

5.2.2 Pruning

Resolution: A main deduction mechanism in SAT solvers is propositional resolution [23]. Mathematically, it can be shown as:

$A \vee x, \qquad \neg x \vee B \vdash A \vee B.$

This is to say that if there are both a clause containing a positive occurrence of variable *x* and a clause with a negated occurrence of *x*, then we can deduce a new clause by merging these two clauses and removing the occurrences of *x*, remaining other literals from the original clauses (*A* and *B* here for example).

This new clause is called a resolvent (i.e. $A \vee B$ here).

Resolution is a complete deduction mechanism on its own. It computes the resolvents of the problem until saturation is achieved. This means we have the guarantee that the empty clause will be generated if and only if the problem is inconsistent.

Unit Resolution (UR) is an important type of resolution. It is a restriction of resolution in which we impose that one of the clauses we resolve on, be a single literal (or a unit clause, defined further down) [37]. We can mathematically show UR as:

$x, \neg x \vee A \vdash A$ and that $\neg x, x \vee A \vdash A.$

UR is not a complete deduction mechanism. However, it can be performed efficiently. It essentially works well with search algorithms, owing to the fact that branching works by assigning values to the variables of the problem (i.e. adding unit clauses to the problem).

Unit resolution's philosophy is simple: if the assignment of variable *x* contradicts the value imposed to it by the clause, then one of the other literals in the clause has to be true. Specifically, when all literals of the clause, but one, contradict the current assignment, then the remaining literal has to be true and we can assign the variable of this literal accordingly. The clause in this case is called a *unit clause*. For instance, under the current assignment: $x = 0, z = 1$, using UR in the clause $x \vee \neg y \vee \neg z$, we can assign *y* to 0.

Search-based solvers use UR to propagate the consequences of every decision made. The process of iteratively applying this rule until no unit clause exists in CNF is called *unit propagation*. A clause whose literals all evaluate to false is called a *conflicting clause*. When there exists a conflicting clause in the formula, the current assignment cannot be extended to a solution; therefore we must backtrack.

The process of giving assignments in a chain using UR rule and of detecting conflicts is called *Boolean constraint propagation* (BCP) [22].

SAT solvers perform some reasoning before the search in order to help simplify the problem (*preprocessing*). Since preprocessing is applied only once, it is possible to incorporate some deduction rules that are otherwise too expensive to be applied at every node of the search tree. For instance, it is generally the case to perform operations such as variable renaming or elimination in preprocessing to generate simpler SAT instances. Such operations are usually difficult and too costly to perform during the search process due to the bookkeeping overhead.

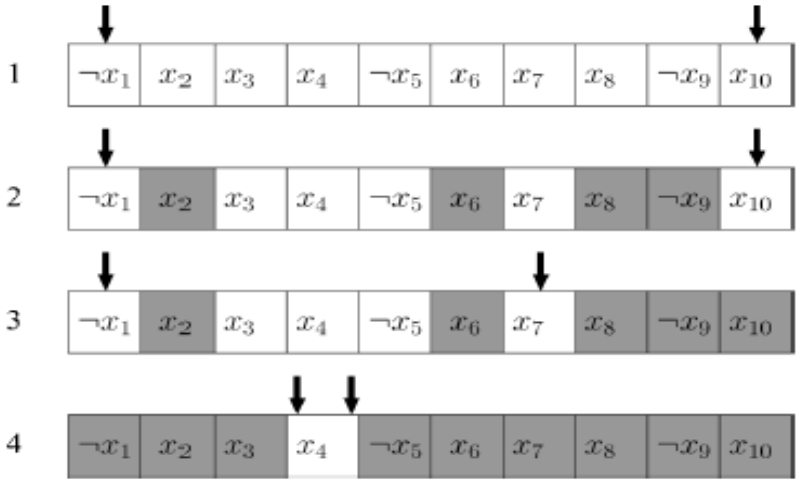

Figure 5.1 – Head/tail approach towards BCP

In BCP concept, there are three possibilities for a literal:

1. It is *free*: if it is not assigned a value by the current partial assignment.
2. It has value 0 if the partial assignment contradicts the value imposed by the clause.
3. It has value 1 if the partial assignment satisfies it.

An example of case 2 would be that an assignment imposes *x = 1*, while the clause contains the literal ¬x. Case 3 means that the clause is satisfied.

Having determined the literals of a clause, BCP has to determine one of these cases for every clause, for every new assignment:

- If all literals of the clause have value 0 then we have reached inconsistency by detecting a conflicting clause and must backtrack.
- The clause is a unit clause (i.e. all literals but one have value 0) in which case the remaining literal forces a new assignment for propagation.

- Undeterministic situation because the clause either contains at least two free literals or already a value 1 literal. In this case this clause is useless in making further deductions until either more variables get assigned (or a backtrack occurs).

There are mechanisms to improve the efficiency of propagation. Zhang and Stickel [21] proposed the use of a mechanism for BCP using head/tail lists. The algorithm states that as long as a clause contains two different literals with a value other than 0 (including free literals too), then the clause is neither unit nor conflicting. Therefore they propose keeping track of two nonzero literals and avoid performing any action as long as these literals exist. In this algorithm, the two literals we keep track of are the first and last nonzero literals of each clause. These clauses will be pointed to by *head* and *tail pointers*.

In this algorithm, an invariant is maintained for the head pointer to be on the first nonzero literal of the clause and the tail pointer on the last one, as shown in Figure 5.1 (taken from [32]). The algorithm has it that if both the head and tail pointers point to the same literal, then the clause is either unit or conflicting, depending on the value of that particular literal. Because in this algorithm we keep track of only two literals, it is more efficient than the literal counting algorithm.

Figure 5.1 comments: For simplicity, the clause considered here consists of variables x_1 to x_{10}. The darker cells correspond to literals with value 0 and empty ones to free literals. At phase 1 (initially), the head and tail pointers point to the

first and last free variables in the array, respectively (because there has not been any assignments yet, these happen to be the first and last items in the array). At phase 2, new assignments have been imposed on the literals of the clause ($x_2 = 0$, $x_6 = 0$, $x_8 = 0$ and $x_9 = 1$, hence $\neg x_9$ cell being darkened) but no action is needed here because the new assignment does not affect the head and tail literals (i.e. the literals we are watching). At phase 3, a new assignment of 0 is imposed on x_{10}, a literal we are watching. ***The algorithm states that if a watched literal is assigned value 0, the pointer is moved right (for head literal) and left (for tail literal) until a free or value 1 variable is reached – or they reach the same free variable hence a unit clause or a conflict if they reach the same variable with value 0.*** Therefore here pointer is moved to x_7 and our tail literal is now x_7 and no longer x_{10}. At phase 4, with the new assignments imposed, the pointers move again to find a free variable. The pointers meet each other at x_4 since it is the only free variable left in the clause. We therefore have a unit clause here which means that x_4 must have value 1.

5.2.3 Conflict analysis and backtracking

Sometimes the choices made at an earlier stage in the search tree, causes a conflict at later levels in the search. Any branch that does not reconsider these decisions will eventually become inconsistent. To find a solution and avoid being stuck in the search space, the SAT solver needs to backtrack to a previous branch and explore another branch, when a conflict is found. A simple algorithm would be to go back up only one level of the search tree (to the immediate most recent decision level) and assign a different value for the branching variable

(*chronological backtracking*). This approach could take a very long time to search an entire search tree, of an average size. Instead, intelligent backtracking algorithms try to analyse the conflict and then backtrack to a decision level that will resolve it (*non-chronological backtracking*).

SAT solvers use a process called *clause learning* which is to analyse a situation and gain some knowledge from it and store the knowledge to prevent similar conflicts. When based on a current assignment, all literals of a clause evaluate to false, we have a conflict. This is called a *conflicting clause*, as mentioned earlier. Clause learning in SAT can learn from a conflicting situation by memorising the clause that lead to the conflict in order to prevent it from happening again; and also figure out the decision level to backtrack to, that would lead to a different search tree where the conflict would not happen anymore.

Using *implication graphs* [18] is one of the main methods used for conflict analysis.

Implication graphs: a representation that captures the variable assignments made by the solver both by propagation and by branching. This representation is a directed acyclic graph (DAG). In this DAG, the vertices represent the assignment of values 0 or 1 to variables and the edges the dependencies between the assignments. For instance, an arc from a to b (represented as $a \rightarrow b$ in the graph) means that assignment of a is one of the factors that lead to the assignment of b (for instance $x_1 \rightarrow x_4$ in Figure 5.2). Logically, each assignment is a consequence of the conjunction of all its predecessors in the graph (i.e. all the arcs leading to that particular variable).

For the purpose of conflict analysis and being able to backtrack to another different decision level, each node in the graph has a depth level, indicating its branching level. The depth levels start from 1 and increase for subsequent branchings. All the variables implied by a decision variable then have the same depth level as their corresponding decision variable. (x_9 and x_7 in Figure 5.2 for instance, since x_7 is implied by x_9, it has the same depth level, shown by $P_9 - P$ standing for phase). The current depth level is the highest decision level in the branching stack (9 in this example). After backtracking, some variables become unassigned, and we decrease the current depth level accordingly.

In the DAG, the vertices with no predecessor are the decision variables *assigned* by the solver (e.g. x_9 and x_1 in Figure 5.2).

Constraints of the problem related to our example (there well may be more constraints in the problem – but they may not affect the branching we are discussing here):

C_1: $x_9 \vee x_7$

C_2: $x_5 \vee \neg x_7 \vee x_8$

C_3: $\neg x_2 \vee x_3$

C_4: $\neg x_4 \vee x_1 \vee x_5 \vee x_6$

C_5: $\neg x_6 \vee \neg x_8 \vee x_1$

C_6: $x_4 \vee x_6 \vee \neg x_3$

Figure 5.2 comments and analysis: shows a partial DAG of an implication graph of a conflict analysis problem. The filled-in circles (namely x_1, x_2, x_6 and x_9) represent decision variables; hollow circles (namely x_3, x_4, x_6, x_7 and x_8) represent solver deductions using the constraints and other implied or assigned variables. The current depth level is 9 (represented by P_9).

In this example, we only consider phases 3, 4, 6 and 9 (represented by P_3, P_4, P_6 and P_9 respectively). The decisions at each phase are represented by similar colours in this graph. The reason we do not consider other phases the solver goes through and the decisions it makes, is because those do not affect our branching decisions in this part of the search space (i.e. *they are not connected to our graph - somehow unrelated*). We call these *unrelated events* because whatever events that do happen in those phases and whatever decisions are made there, they do not affect this part of the tree.

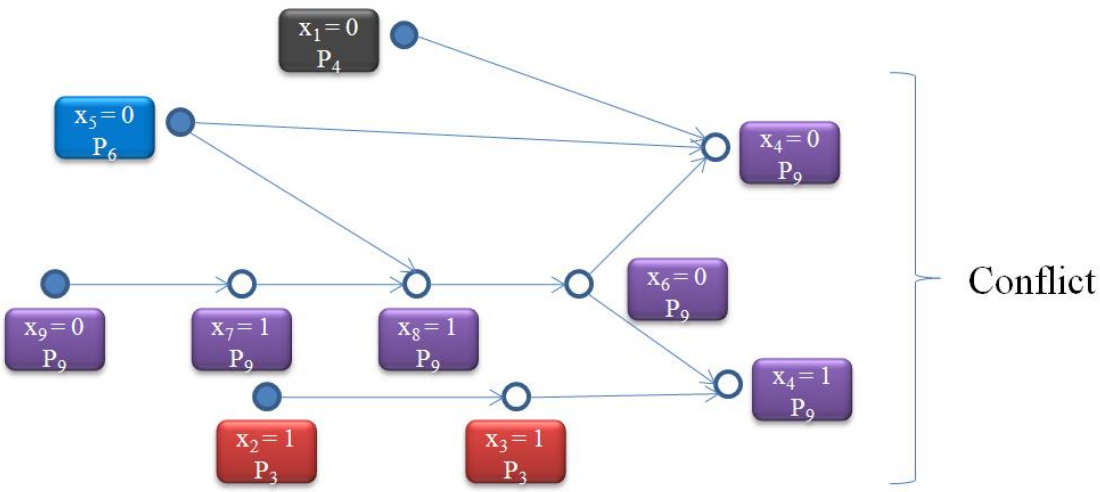

Figure 5.2 - DAG of conflict analysis using an implication graph

We now observe the chronological decision making and consequences of those decisions of the solver in this search tree:

***P_1 and P_2*:** Unrelated events happen.

***P_3*:**

The solver assigns value 1 to x_2:

- by C_3 and $x_2 = 1$ we then have: $x_3 = 1$

***P_4*:**

The solver assigns value 0 to x_1.

***P_5*:** Unrelated events.

***P_6*:** The solver assigns value 0 to x_5.

***P_7* and *P_8*:** Unrelated events.

***P_9*:**

The solver assigns value 0 to x_9.

- by C_1 and $x_9 = 0$ we then have: $x_7 = 1$;
- by C_2, $x_5 = 0$ and $x_7 = 1$ we then have: $x_8 = 1$;
- by C_5, $x_1 = 0$ and $x_8 = 1$ we then have: $x_6 = 0$;

  (*at this point we have reached a unit clause, so we can determine a value for x_4*).
- by C_6, $x_3 = 1$ and $x_6 = 0$ we then have: $x_4 = 1$;
- by C_4, $x_1 = 0$, $x_5 = 0$ and $x_6 = 0$ we then have: $x_4 = 0$.

Although we did reach a unit clause and assigned a value to x_4, but the solver found two values for x_4. x_4 cannot both be 0 and 1 at the same time therefore we have reached a conflict. We call x_4 a conflicting variable.

There are many ways of how to interpret this conflict and determining where to backtrack to, to avoid doing redundant computations and being stuck in an endless loop in this local branching problem. But the solver can also use this information to prevent future conflicts which follow the same pattern. Marques Silva and Sakallah explain the Unit Implication Point in [17]. However, these methods are out of the scope of this report and we will not discuss them further here.

We did reach a conflict in this example, but we also reached a unit clause (hence the solver being able to determine the values for x_4). We have also presented the head/tail approach leading up to the conflict, in Figure 5.3, following similar conventions from Figure 5.1. Figure 5.3 is self-explanatory. Since there are a number of consequences in phase 9, we represent each step in a separate table for more clarification. At table P_9-4, we have reached the unit clause and also the conflict.

P0	Head/tail	↓								↓
	Value									
	Variable	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

P3	Head/tail	↓								↓
	Value		1	1						
	Variable	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

P4	Head/tail				↓					↓
	Value	0	1	1						
	Variable	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

P6	Head/tail				↓					↓
	Value	0	1	1		0				
	Variable	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

P9-1	Head/tail				↓				↓	
	Value	0	1	1		0				0
	Variable	x_1	x_2	x_3	x_4	x_5	x_6		x_8	x_9

P9-2	Head/tail				↓				↓	
	Value	0	1	1		0		1		0
	Variable	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

P9-3	Head/tail				↓		↓			
	Value	0	1	1		0		1	1	0
	Variable	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

P9-4	Head/tail				↓↓					
	Value	0	1	1		0	0	1	1	0
	Variable	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

Figure 5.3 – Head/tail trace of the DAG from Figure 5.2.

Chapter 6

Conclusion and Future Work

6.1 Summary

In this report we aimed to show and emphasise on the Event Calculus to be a robust and flexible formalisation to deal with commonsense reasoning. We started by describing commonsense reasoning and some of its features and reasoning types in Chapter 1. A detailed description of the Cyc project was given and it was explained how it gathers information about the world.

Cyc's world knowledge is automatically collected and filtered by machine and then approved by human ontologists. Cyc crawls the web to collect the knowledge it finds interesting based on what it already knows. However, it does allow for manual assertions as well. New concepts in Cyc are encouraged to be formed if there are more than 10 properties of that concept to be stored on the knowledge base otherwise it is advised to use a combination of existing concepts. This is a good example of a systematic approach towards tackling acquisition of commonsense knowledge in the real world.

In Chapter 2, an introduction of the Event Calculus was given: a representational tool for formalising commonsense. We also talked about the features of the EC and discussed how to compute circumscription, the mathematical device to deal with default reasoning. An example was presented: an EC representation of a real world scenario was constructed from a natural language description.

We then presented our framework of the Bucket World in Chapter 3 in which we simulate the real world in an abstract commonsense point of view. Our agent in this scenario has a goal to achieve and we elaborate the steps of the proof. In this chapter we compared our methods with those of Shin and Davis [40]. Our flagging system was described which deals with triggered events - to stop them from occurring repeatedly. In formalising a world scenario in the Event Calculus, we have to define a framework which simulates the world and the relationship between entities and actions can potentially happen and their effects. We then have to formalise the scenario which will be the narrative of our simulation. The framework and the narrative together form our domain description.

Chapter 4 discussed the acquisition of commonsense knowledge and the problem of identifying the correct level of detail of knowledge to be imported to the reasoning system. We introduced a method of transforming formulas of the Event Calculus into propositional logic in the Conjunctive Normal Form; which would be suitable for feeding into SAT solvers. We presented an example of a domain description in the EC and then converted it into propositional logic; and showed that using this method we could automatically prove propositions by a SAT solver as we could do manually.

Chapter 5 discussed different automated reasoning methods that deal with the Event Calculus. We then go into detail of one of these methods: SAT solving. Branching and pruning methods in SAT solving were described.

In general, performing commonsense reasoning based on the domain specific problem heavily depends on the background knowledge about the world. For a

systematic approach towards gathering background knowledge for solving a commonsense problem we need to determine:

- What knowledge is related to our specific domain of problem;
- The consistency of this knowledge with our domain-specific representation;
- To what extent of detail we need the extra knowledge on a given problem: For instance when Fred leaves home and stops by a shop on his way to the office, do we care how he got to the office? Do we care by which means he went to the shop and from there to the office? Or do we care to know that he has stopped by the shop? In different scenarios we care about different aspects of the problem.

We showed various features of the Event Calculus in this report. In the comparison of our scenario with the one of Shin and Davis, we showed the superiority of the EC.

6.2 Future Work

An interesting and useful follow up of this work would be to convert the Fluid Theory of Davis [9] into the event calculus formalism and then combine it with the example presented in Chapter 3. This will be the first instance of its kind to combine two well developed commonsense aspects of the real world in a merged framework. The example in Chapter 3 is investigated at a general level of commonsense, one which we use every day: it does not go into too much details of the entities. For instance it does not care about the details of the walking action such as that the left leg should go forward after the right leg has landed and so on. However, this is not to say that this level of detail is not useful; quite the opposite it potentially is. It all depends on what we need it for. From a robotic point of view for instance, the smallest details of a robot walking are the most important for the walking action. The approach towards using the Event Calculus in performing commonsense reasoning for robots is already being tackled by pioneer researchers of the field such as Murray Shanahan, Mark Witkowsky and David Randell [75, 76, 77]. This is a promising approach since it works on an abstract level to deal with everyday problems that a robot has to deal with. On the other hand, a detailed perspective is very useful in different situations. For instance, Davis is researching on a detailed commonsense level (as to movements of a liquid molecules) that is used in a factory (he uses PDDL+ for his formalism but as discussed in Chapter 3, the EC can outperform PDDL+ on various grounds in commonsense reasoning).

Both perspectives (i.e. general and detailed) are necessary and useful. However, there has not been any attempt to combine these and create a new merged framework in which both views can be investigated.

We will create such framework and analyse the result of this combination. So it will not only model for an agent to fill in a bucket until it is full, and then move the bucket, but also it will model the movement of water inside the bucket on a molecular movement level. Therefore the filling-in action will have actual physical representation in the framework. This will render the Event Calculus as a suitable representational task for planning some robotics actions.

In Chapters 2 and 3, we translated natural language scenarios into the Event Calculus and then proved our propositions on those translations. We discussed automation of this process with Murray Shanahan and Erik Mueller in verbal and correspondence. We aim to investigate this by means of shallow parsing of the text and then derive algorithms to represent them in EC. However, this is a difficult task considering the notes pointed out in section 3.2 of Chapter 3 but quite fruitful: a machine with the ability to translate natural language text into the EC and perform commonsense reasoning on them. This will be of interest to researchers in various fields of Artificial Intelligence such as automated reasoning, natural language processing, robotics and commonsense reasoning. The Cyc project has developed a similar fashion, but as an independent and private company. This will show the strengths of the Event Calculus and establish it as a strong representational tool for handling commonsense reasoning extracted from natural language text. Developing such algorithms will also result in a reduction of the time spent on manual translation [2, 3]. This automatic translation is non-trivial and implicates the following tasks:

- Find relations between entities of the discourse or sentence
- Identify the commonsense relations with help of external knowledge
- Formulate this knowledge in the EC

We have identified Cyc's world knowledge to be a good source of external knowledge for this purpose. For using Cyc's knowledgebase, however, we will need to:

- Match and extract relations from Cyc, based on our domain-specific knowledge
- Check for inconsistency between the knowledge coming from Cyc and our problem representation
- Formulate the knowledge from CycL into EC

Bibliography

[1] Douglas Lenat. Computers versus Common Sense, May 30th 2006. Google TechTalk, available on Google Videos.

[2] E. Mueller. Understanding script-based stories using commonsense reasoning. ScienceDirect July 2004. IBM Thomas J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, USA. Cognitive Systems Research 5, pp. 307-340, 2004.

[3] E. Mueller. Story understanding through multi-representation model construction. IBM Thomas J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, USA

[4] John McCarthy. Programs with common sense. In Mechanisation of Thought Processes: Proceedings of a symposium held at the National Physical Laboratory on 24th, 25th, 26th and 27th November 1958 (Vol. 1 pp.75-91). London: Her Majesty's Stationery Office, 1958.

[5] Murray Shanahan. The event calculus explained. In Wooldridge and Veloso, Artificial Intelligence today: Recent trends and developments (Lecture Notes in Computer Science, Vol. 1600 pp. 409-430). Berlin: Springer, 1999.

[6] M. Shanahan. An attempt to formalise a non-trivial benchmark problem in common sense reasoning. Artificial Intelligence 153, pp. 141-165, 2004.

[7] Lenat and Guha. Building large knowledge-based systems: Representation and inference in the Cyc project. Reading, MA: Addison-Wesley, 1990.

[8] M. Ginsberg, D. Smith. Reasoning about action II: The qualification problem: - Artificial Intelligence, 1988

[9] E. Davis. Pouring Liquids: A Study in Commonsense Physical Reasoning. Artificial Intelligence, Elsevier, 2008.

[10] V. Lifschitz. Benchmark problems for formal nonmonotonic reasoning: Version 2.00, 1989. In Reinfrank, Kleer, Ginsberg, and Sandewall, Non-Monotonic reasoning: Proceedings of the second international workshop (Lecture Notes in Computer Science, Vol. 346, pp. 202-219). Berlin: Springer.

[11] Morgenstern and Miller. Common sense problem page, 2004. From Stanford University Web-site: http://www-formal.stanford.edu/leora/commonsense/

[12] Randell, Cui and Cohn. A spatial logic based on regions and connection, 1992. In Nebel, Rich and Swartout, Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (pp. 165-176). San Mateo, CA: Morgan Kaufmann.

[13] L. Morgenstern. Mid-sized axiomatizations of commonsense problems: A case study in egg cracking. Studia Logica, 2001

[14] E. Mueller. Commonsense Reasoning, 2006. Elselvier, Morgan Kaufmann

[15] N. Cassimatis. Polyscheme: A cognitive architecture for integrating multiple representation and inference schemes, 2002. Unpublished doctoral dissertation, Massachusetts Institute of Technology, Cambridge, MA.

[16] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. Computational Intelligence. Vol. 9, pp. 268–299, 1993.

[17] Jao Marques Silva and Karem Sakallah. A new search algorithm for satisfiability. International Conference on Computer Aided Design (ICCAD). pp. 220–227, 1996.

[18] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Said Jabbour and Lakhdar Sais. A Generalized Framework for Conflict Analysis. Microsoft Research, Technical Report MSR-TR-2008-34, 2008.

[19] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. Information Sciences Vol. 7, 2, pp. 85–132, 1974.

[20] David Waltz. Generating semantic descriptions from drawings of scenes with shadows. The Psychology of Computer Vision, McGraw-Hill, Chapter 3, 1972 (Preliminary version as MIT research report (MACAI- TR-271), 1972.)

[21] Hanto Zhang and Mark Stickel. Implementing the Davis-Putnam method. Journal of Automated Reasoning. Vol. 24, Issues 1 and 2, pp. 277–296, 2000.

[22] David McAllester. Truth Maintenance. North American National Conference on Artificial Intelligence (AAAI), pp. 1109–1116, 1990.

[23] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. J. ACM Vol. 7, Issue 3, pp. 201–215, 1960.

[24] Kathy Panton, Cynthia Matuszek, Douglas Lenat, Dave Schneider, Michael Witbrock, Nick Siegel and Blake Shepard. From Cyc to Intelligent Assistant, 2006. In Cai and Abascal, Ambient Intelligence in Everyday Life, LNAI 3864 pp. 1-31. Berlin: Springer.

[25] Falkenhainer, Forbus and Genter. The structure-mapping engine: Algorithm and examples. Artificial Intelligence, 3 pp. 251-288, 1989.

[26] João Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. 9th Portuguese Conference on Artificial Intelligence (EPIA), 1999.

[27] John McCarthy. Situations, actions and causal laws (Memo 2), 1963. Stanford, CA: Stanford Artificial Intelligence Project, Stanford University.

[28] McCarthy and Hayes. Some philosophical problems fram the standpoint of artificial intelligence, 1969. In Meltzer and Michie, Machine intelligence 4 pp. 463-502. Edinburgh, Scotland: Edinburgh University Press.

[29] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang and Sharad Malik. Chaff: Engineering an efficient SAT solver. International Design Automation Conference (DAC), pp. 530–535, 2001.

[30] Pedro Meseguer. Interleaved depth-first search. International Joint Conference on Artificial Intelligence (IJCAI), pp. 1382–1387, 1997.

[31] William Harvey and Matthew Ginsberg. Limited discrepancy search. Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1, pp. 607-613 1995.

[32] Lucas Bordeaux, Youssef Hamadi and Lintao Zhang. Propositional Satisfiability and Constraint Programming: A Comparative Survey. ACM Computing Surveys, Volume 38, Issue 4, 2006.

[33] Fredric Boussemart, Fred Hemery, Christopher Lecoutre and Sais Lakhdar. Boosting systematic search by weighting contraints. Proceedings of ECAI, 2004.

[34] Rina Dechter and Itay Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. Artificial Intelligence 68, pp. 211–241, 1994.

[35] Robert M. Haralick and Gordon L. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. Artificial Intelligence 14, pp. 263–313, 1980.

[36] Yannis Argyropoulos and Kostas Stergiou. A Study of SAT-Based Branching Heuristics for the CSP. Artificial Intelligence: Theories, Models and Applications, pp. 38-50, 2008.

[37] Martin Davis, George Logemann, Donald Loveland. A machine program for theorem-proving. Communications of the ACM, VOL. 5, Issue 7, pp. 393–397, 1962.

[38] Marco Dorigo, Mauro Birattari, Thomas Stutzle,. Ant Colony Optimization. Computational Intelligence, VOL. 1, Issue 4, pp. 28-39, 2006.

[39] Robert Nieuwenhuis, and Albert Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. International Conference on Computer-Aided Verification. 321–334, 2005.

[40] J. Shin and E. Davis. Processes and Continuous Change in a SAT-based Planner. Artificial Intelligence, Vol. 166, Issue 1-2, PP. 194 – 253, 2005.

[41] Vladimir Lifschitz. Circumscription, 1994. In Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 3, pages 297-352. Oxford University Press.

[42] John McCarthy. Circumscription: A form of non-monotonic reasoning, 1980. Artificial Intelligence 13(1-2):23-79.

[43] DIMACS. Satisfiability suggested format. Technical report, Center for Discrete Mathematics and Theoretical Computer Science, 1993.

[44] K. Doets. From logic to logic programming. MIT Press, Cambridge, 1994. PP. 17-18.

[45] Cycorp. Ontological engineer's handbook version 0.7, 2002. Austin, TX: Cycorp.

[46] Lenat. Cyc: A large-scale investment in knowledge infrastructure, 1995. Communications of the ACM, 38(11) pp. 33-48.

[47] Cynthia Matuszek, Michael Witbrock, Robert Kahlert, John Cabral, Dave Schneider, Purvesh Shah and Dough Lenat. Searching for Common Sense: Populating Cyc from the Web, 2005. In Proceedings of the Twentieth National Conference on Artificial Intelligence, Pittsburgh, Pennsylvania.

[48] Murray Shanahan. Representing continuous change in the event calculus, 1990. In Aiello, Proceedings of the Ninth European Conference on Artificial Intelligence (pp. 598-603). London: Pitman.

[49] R. Reiter. Natural actions, concurrency and continuous time in situation calculus, 1996. In Aiello, Doyle and Shapiro, Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (pp. 2-13). San Francisco: Morgan Kaufmann.

[50] Kowalski and Sergot. A logic-based calculus of events, 1986. New Generation Computing, 4(1) pp. 67-95.

[51] Miller and Shanahan. The event calculus in classical logic-alternative axiomatisations, 1999. Linkoping Electronic Articles in Computer and Information Science, 4(016).

[52] Miller and Shanahan. Some alternative formulations of the event calculus, 2002. In Kakas and Sadri, Computational logic: Logic programming and beyond: Essays in honour of Robert Kowalski, part II (Lecture Notes in Computer Science Vol. 2048 pp. 452-490). Berlin: Springer.

[53] Kowalski and Sadri. Reconciling the event calculus with the situation calculus, 1997. Journal of Logic Programming, 31(1-3) pp. 39-58.

[54] John McCarthy. Generality in artificial intelligence, 1987. Communications of ACM, 30(12) pp. 1030-1035.

[55] John McCarthy. First-order theories of individual concepts and propositions, 1979. In Hayes, Michie and Mikulich, Machine intelligence 9 pp. 129-148. Chichester, UK: Ellis Horwood.

[56] Murray Shanahan. Solving the Frame Problem, 1997. Cambridge, MA: MIT Press.

[57] Kowalski and Sadri. The situation calculus and event calculus compared, 1994. In Bruynooghe, Logic Programming: The 1994 international symposium (pp. 539-553). Cambridge, MA: MIT Press.

[58], M. Kay. Unification, 1992. In Ronser and Johnson, Computational Linguistics and Formal Semantics. Cambridge University Press.

[59] Hanks and McDermott. Temporal reasoning and default logics (Tech. Rep. No. YALE/DCS/tr430), 1985. New Haven, CT: Computer Science Department, Yale University.

[60] Hanks and McDermott. Default reasoning, nonmonotonic logics and the frame problem, 1986. In Proceedings of the Fifth National Conference on Artificial Intelligence (pp. 328-333). Los Altos, CAL Morgan Kaufmann.

[61] Hanks and McDermott. Nonmonotonic logic and temporal projection, 1987. Artificial Intelligence, 33(3) pp. 379-412.

[62] Murray Shanahan. An abductive event calculus planner. Journal of Logic Programming, 44(1-3) pp.207-240, 2000.

[63] Murray Shanahan. Abductive event calculus planner [Computer Software]. From Department of Electrical and Electronic Engineering, Imperial College London. Website: http://www.iis.ee.ic.ac.uk/~mpsha/planners.html

[64] Shanahan and Witkowski. Event calculus planning through satisfiability, 2004. Journal of Logic and Computation, 14(5) pp. 731-745.

[65] Erik Mueller. Event calculus reasoning through satisfiability. Journal of Logic and Computation, 14(5) pp. 703-730, 2004.

[66] Erik Mueller. A tool for satisfiability-based commonsense reasoning in the event calculus. In Barr and Markov, Proceedings of the Seventeenth International Florida Artificial Intelligence Reasearch Society Conference (pp. 147-152). Menlo Park, CA: AAAI Press, 2004.

[67] Erik Mueller. Discrete event calculus deduction using first-order automated theorem proving. In Konev and Schulz, Proceedings of the Fifth International Workshop on the Implementation of Logics (pp. 43-56). Liverpool, UK: Department of Computer Science, University of Liverpool, 2005.

[68] Erik Mueller. Reasoning in the event calculus using first-order automated theorem proving, 2005. In Russel and Markov, Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference (pp. 840-841). Melno Park, CA: AAAI Press.

[**69]** Sutcliffe and Suttner. The TPTP problem library for automated theorem proving, 2005. From Department of Computer Science, University of Miami. Website: http://www.cs.miami.edu/~tptp

[**70]** Kakas, Michael and Miller. Modular E: an elaboration tolerant approach to the ramification and qualification problems. 8th International Conference on Logic Programming and Nonmonotonic Reasoning, 2005.

[**71]** Kakas, Miller and Toni. E-RES: Reasoning about Actions, Events and Observations. the proceedings of LPNMR2001, LNAI Vol. 2173, pp. 254-266, Springer Verlag, 2001.

[72] Erik Muller. Discrete Event Calculus Reasoner. Implementation available at: http://decreasoner.sourceforge.net/

[**73]** Mueller and Sutcliffe. Discrete event calculus deduction using first-order automated theorem proving. In Proceedings of the Fifth International Workshop on the Implementation of Logics, 2005.

[74] Kakas and Mourlas. ACLP: A Case for Non-Monotonic Reasoning, Proceedings of the 7th International Workshop on Non- Monotonic Reasoning, 1998.

[75] Shanahan and Witkowski. High-Level Robot Control Through Logic, Proceedings ATAL 2000, published as *Intelligent* Agents VII, Springer-Verlag, pp. 104-121, 2001.

[76] Shanahan and Randell. A Logic-Based Formulation of Active Visual Perception, Proceedings KR 2004, pp. 64-72.

[77] Murray Shanahan. Perception as Abduction: Turning Sensor Data into Meaningful Representation, Cognitive Science, vol. 29, pp. 103-134, 2005.

Appendices

Appendix A

Event Calculus Axioms

EC1: $Clipped(t_1, f, t_2) \equiv \exists e, t\ (Happens(e, t) \wedge t_1 \leq t \leq t_2 \wedge Terminates(e, f, t))$

EC2: $Declipped(t_1, f, t_2) \equiv \exists e, t\ (Happens(e, t) \wedge t_1 \leq t \leq t_2 \wedge Initiates(e, f, t))$

EC3: $StoppedIn(t_1, f, t_2) \equiv \exists e, t\ (Happens(e, t) \wedge t_1 < t < t_2 \wedge Terminates(e, f, t))$

EC4: $StartedIn(t_1, f, t_2) \equiv \exists e, t\ (Happens(e, t) \wedge t_1 < t < t_2 \wedge Initiates(e, f, t))$

EC5: $(Happens(e, t_1) \wedge Initiates(e, f_1, t_1) \wedge 0 < t_2 \wedge Trajectory(f_1, t_1, f_2, t_2) \wedge \neg StoppedIn(t_1, f_1, t_1 + t_2)) \Rightarrow HoldsAt(f_2, t_1 + t_2)$

EC6: $(Happens(e, t_1) \wedge Terminates(e, f_1, t_1) \wedge 0 < t_2 \wedge AntiTrajectory(f_1, t_1, f_2, t_2) \wedge \neg StartedIn(t_1, f_1, t_1 + t_2)) \Rightarrow HoldsAt(f_2, t_1 + t_2)$

EC7: $PersistsBetween(t_1, f, t_2) \equiv \neg\exists t\ (ReleasedAt(f, t) \wedge t_1 < t \leq t_2)$

EC8: $ReleasedBetween(t_1, f, t_2) \equiv \exists t\ (Happens(e, t) \wedge t_1 \leq t < t_2 \wedge Releases(e, f, t))$

EC9: $(HoldsAt(f, t_1) \wedge t_1 < t_2 \wedge PersistsBetween(t_1, f, t_2) \wedge \neg Clipped(t_1, f, t_2)) \Rightarrow HoldsAt(f, t_2)$

EC10: $(\neg HoldsAt(f, t_1) \wedge t_1 < t_2 \wedge PersistsBetween(t_1, f, t_2) \wedge \neg Declipped(t_1, f, t_2)) \Rightarrow \neg HoldsAt(f, t_2)$

EC11: $(ReleasedAt(f, t_1) \wedge t_1 < t_2 \wedge \neg Clipped(t_1, f, t_2) \wedge \neg Declipped(t_1, f, t_2)) \Rightarrow ReleasedAt(f, t_2)$

EC12: $(\neg ReleasedAt(f, t_1) \wedge t_1 < t_2 \wedge \neg ReleasedBetween(t_1, f, t_2)) \Rightarrow ReleasedAt(f, t_2)$

EC13: $\text{ReleasedIn}(t_1, f, t_2) \equiv \exists e, t\ (\text{Happens}(e, t) \wedge t_1 < t < t_2 \wedge \text{Releases}(e, f, t))$

EC14: $(\text{Happens}(e, t_1) \wedge \text{Initiates}(e, f, t_1) \wedge t_1 < t_2 \wedge \neg\text{StoppedIn}(t_1, f, t_2) \wedge \neg\text{ReleasedIn}(t_1, f, t_2)) \Rightarrow \text{HoldsAt}(f, t_2)$

EC15: $(\text{Happens}(e, t_1) \wedge \text{Terminates}(e, f, t_1) \wedge t_1 < t_2 \wedge \neg\text{StartedIn}(t_1, f, t_2) \wedge \neg\text{ReleasedIn}(t_1, f, t_2)) \Rightarrow \neg\text{HoldsAt}(f, t_2)$

EC16: $(\text{Happens}(e, t_1) \wedge \text{Releases}(e, f, t_1) \wedge t1 < t_2 \wedge \neg\text{StoppedIn}(t_1, f, t_2) \wedge \neg\text{StartedIn}(t_1, f, t_2)) \Rightarrow \text{ReleasedAt}(f, t_2)$

EC17: $(\text{Happens}(e, t_1) \wedge (\text{Initiates}(e, f, t_1) \vee \text{Terminates}(e, f, t_1)) \wedge t_1 < t_2 \wedge \neg\text{ReleasedIn}(t_1, f, t_2) \Rightarrow \neg\text{ReleasedAt}(f, t_2)$

Appendix B

Discrete Event Calculus Axioms

DEC1: $\text{StoppedIn}(t_1, f, t_2) \equiv \exists e, t\ (\text{Happens}(e, t) \wedge t_1 < t < t_2 \wedge \text{Terminates}(e, f, t))$

DEC2: $\text{StartedIn}(t_1, f, t_2) \equiv \exists e, t\ (\text{Happens}(e, t) \wedge t_1 < t < t_2 \wedge \text{Initiates}(e, f, t))$

DEC3: $(\text{Happens}(e, t_1) \wedge \text{Initiates}(e, f_1, t_1) \wedge 0 < t_2 \wedge \text{Trajectory}(f_1, t_1, f_2, t_2) \wedge \neg\text{StoppedIn}(t_1, f_1, t_1 + t_2)) \Rightarrow \text{HoldsAt}(f_2, t_1 + t_2)$

DEC4: $(\text{Happens}(e, t_1) \wedge \text{Terminates}(e, f_1, t_1) \wedge 0 < t_2 \wedge \text{AntiTrajectory}(f_1, t_1, f_2, t_2) \wedge \neg\text{StartedIn}(t_1, f_1, t_1 + t_2)) \Rightarrow \text{HoldsAt}(f_2, t_1 + t_2)$

DEC5: $(\text{HoldsAt}(f, t) \wedge \neg\text{ReleasedAt}(f, t + 1) \wedge \neg\exists e\ (\text{Happens}(e, t) \wedge \text{Terminates}(e, f, t))) \Rightarrow \text{HoldsAt}(f, t + 1)$

DEC6: $(\neg\text{HoldsAt}(f, t) \wedge \neg\text{ReleasedAt}(f, t+1) \wedge \neg\exists e\ (\text{Happens}(e, t) \wedge \text{Initiates}(e, f, t))) \Rightarrow \neg\text{HoldsAt}(f, t+1)$

DEC7: $(\text{ReleasedAt}(f, t) \wedge \neg\exists e\ (\text{Happens}(e, t) \wedge (\text{Initiates}(e, f, t) \vee \text{Terminates}(e, f, t))) \Rightarrow \text{ReleasedAt}(f, t + 1)$

DEC8: $(\neg\text{ReleasedAt}(f, t) \wedge \neg\exists e\ (\text{Happens}(e, t) \wedge (\text{Releases}(e, f, t))) \Rightarrow \neg\text{ReleasedAt}(f, t + 1)$

DEC9: $(\text{Happens}(e, t) \wedge \text{Initiates}(e, f, t)) \Rightarrow \text{HoldsAt}(f, t + 1)$

DEC10: $(\text{Happens}(e, t) \wedge \text{Terminates}(e, f, t)) \Rightarrow \neg\text{HoldsAt}(f, t + 1)$

DEC11: $(\text{Happens}(e, t) \wedge \text{Releases}(e, f, t)) \Rightarrow \text{ReleasedAt}(f, t + 1)$

DEC12: $(\text{Happens}(e, t) \wedge (\text{Initiates}(e, f, t) \vee \text{Terminates}(e, f, t))) \Rightarrow \neg\text{ReleasedAt}(f, t + 1)$

Appendix C

Shin and Davis description of the Bucket domain

```
;;
=============================================================
=====
;; "Bucket" Domain: ;;
;;  Deliver a specified amount of water to a specified ;;
;;  location(s) by a specified deadline. ;;
;; ;;
;;  Assumptions: ;;
;; - An agent can carry at most one bucket at a time. ;;
;; - Zero or more than one tap are in each location. ;;
;; - Each tap fills only one bucket at a time. ;;
;; - Each bucket can be filled by more than one tap ;;
;;  in a location at a time, allowing concurrent continuous ;;
;;  changes on the level of a bucket. ;;
;;
=============================================================
=========
(define          (domain         Buckets)
(:requirements          :time    :continuous-effects)
(:types agent   bucket tap      location)
(:predicates     (at ?o - (either agent bucket tap) ?l - location)
(on      ?t - tap)
(filling          ?t - tap ?b - bucket)
(carrying         ?a - agent ?b - bucket)
(is_walking     ?a - agent ?d - location)
(connected      ?s - location ?d - location)
)
(:functions      (capacity ?b - bucket) - float
(flow_rate       ?t - tap) - float
(walking_speed          ?a - agent) - float
(distance         ?s - location ?d - location) - float
(amount_of_water      ?l - location) - fluent
(distance_to_walk      ?a - agent ?d - location) - fluent
(level   ?b - bucket) - fluent
)
;;
=============================================================
===
;; Filling buckets with taps ;;
;;
=============================================================
===
(:action          turnOnTap
          :parameters (?a - agent ?t - tap ?b - bucket ?l - location)
```

```
        :precondition (and (at ?a ?l)
                (at ?b ?l)
                (at ?t ?l)
                (not (on ?t)))
        :effect (and (on ?t)
                (filling ?t ?b))
)

(:action        turnOffTap
        :parameters (?a - agent ?t - tap ?b - bucket ?l - location)
        :precondition (and (at ?a ?l)
                (at ?t ?l)
                (on ?t)
                (filling ?t ?b))
        :effect (and (not (on ?t))
                (not (filling ?t ?b)))
)

(:process       fillingBucket
        :parameters (?b - bucket ?t - tap ?l - location)
        :precondition (and (at ?b ?l)
                (at ?t ?l)
                (filling ?t ?l)
                (<= (level ?b) (capacity ?b)))
        :effect (increase (level ?b) (* #t (flow_rate ?t)))
)

;;
=============================================================
===
;; Moving buckets between locations ;;
;;
=============================================================
===

(:action        pickUp
        :parameters (?a - agent ?b - bucket ?l - location)
        :precondition (and (at ?a ?l)
                (at ?b ?l))
        :effect (and (not (at ?b ?l))
                (carrying ?a ?b))
)

(:action        putDown
        :parameters (?a - agent ?b - bucket ?l - location)
        :precondition (and (at ?a ?l)
                (carrying ?a ?b))
        :effect (and (at ?b ?l)
                (not (carrying ?a ?b)))
)
```

```
 (:action        go
        :parameters (?a - agent ?s - location ?d - location)
        :precondition (and (at ?a ?s)
                (or (connected ?s ?d) (connected ?d ?s))
                (not (is_walking ?a ?d)))
        :effect (and (not (at ?a ?s))
                (is_walking ?a ?d)
                (assign (distance_to_walk ?a ?d)
                (distance ?d ?s)))
)

(:process       walking
        :parameters (?a - agent ?d - location)
        :precondition (and (is_walking ?a ?d)
                (>= (distance_to_walk ?a ?d) 0)
        :effect (decrease (distance_to_walk ?a ?d)
                (* #t (walking_speed ?a)))
)

(:event         arrive
        :parameters (?a - agent ?d - location)
        :precondition (and (is_walking ?a ?d)
                (<= (distance-to-walk ?a ?d) 0))
        :effect (and (not (is_walking ?a ?d))
                (at ?a ?d))
)

;;
==========================================================
=
;; Filling among buckets in a location ;;
;;
==========================================================
=

(:action        pour
        :parameters
                (?a - agent ?s - bucket ?d - bucket ?q - real ?l - location)
        :precondition (and (at ?a ?l)
                (carrying ?a ?s)
                (at ?d ?l)
                (> ?q 0)
                (<= ?q (level ?s))
                (<= ?q (- (capacity ?d) (level ?d))))
        :effect (and (decrease (level ?s) ?q)
                (increase (level ?d) ?q))
)

(:action        deliver
```

```
        :parameters (?a - agent ?b - bucket ?l - location ?q - real)
        :precondition (and (at ?a ?l)
                (carrying ?a ?b)
                (> ?q 0)
                (<= ?q (level ?b)))
        :effect (and (increase (amount_of_water ?l) ?q)
                (decrease (level ?b) ?q))
)
```

Appendix D

Shin and Davis version of a scenario for the Bucket domain

```
;;
=============================================================
======
;; ;;
;; A possible solution: ;;
;;      1. turnOnTap(ERNIE,TAP1,BUCKET1,SL) ;;
;;              ==> fillingBucket(BUCKET1,TAP1,SL) on ;;
;;      2. turnOffTAP(ERNIE,TAP1,BUCKET1,SL) ;;
;;      3. turnOnTap(ERNIE,TAP1,BUCKET2,SL) ;;
;;              ==> fillingBucket(BUCKET2,TAP1,SL) on ;;
;;      4. pickUp(ERNIE,BUCKET1,SL) ;;
;;      5. go(ERNIE,SL,DL) ==> walking(ERNIE,DL) on ;;
;;      6. arrive(ERNIE,DL) ;;
;;      7. deliver(ERNIE,BUCKET1,DL,1) ;;
;;      8. go(ERNIE,DL,SL) ==> walking(ERNIE,SL) on ;;
;;      9. arrive(ERNIE,SL) ;;
;;      10. turnOffTAP(ERNIE,TAP1,BUCKET2,SL) ;;
;;      11. pickUp(ERNIE,BUCKET2,SL) ;;
;;      12. go(ERNIE,SL,DL) ==> walking(ERNIE,DL) on ;;
;;      13. arrive(ERNIE,DL) ;;
;;      14. deliver(ERNIE,BUCKET2,DL,4) ;;
;;
=============================================================
======
(define         (problem problem1)
(:domain        Buckets)
(:requirements          :time :continuous-effects)
(:objects       SL - location DL - location
        TAP1 - tap
        BUCKET1 - bucket BUCKET2 - bucket
        ERNIE - agent
)

(:init  (at ERNIE SL)
        (at BUCKET1 SL)
        (at BUCKET2 SL)
        (at TAP1 SL)
        (= (flow_rate TAP1) 0.1)
        (= (walking_speed ERNIE) 5)
        (= (capacity BUCKET1) 4)
        (= (capacity BUCKET2) 4)
        (= (distance SL DL) 100)
```

```
        (= (distance DL SL) 100)
        (= (amount_of_water SL) 0)
        (= (amount_of_water DL) 0)
        (= (distance_to_walk ERNIE SL) 0)
        (= (distance_to_walk ERNIE DL) 0)
        (= (level BUCKET1) 0)
        (= (level BUCKET2) 0)
        (connected SL DL)
        (connected DL SL)
)

(:goal  (and    (>= (amount_of_water DL) 5))
        (<= ?total-time 70))
)
)
```

Appendix E

Constraint Programming

Constraint Programming typically provides languages or libraries whose aim is to allow the development of application-specific search algorithms. Therefore, the way a problem is expressed in CP is generally dependent on the tool which is used. In CP the problem is modelled using variables ranging over a finite domain.

Formally, a CP problem is defined [36] as a triple *(X, D, C)*, where $X = \{x_1, x_2, .. x_n\}$ is a finite set of *n* variables, $D = \{D(x_1), D(x_2)...., D(x_n)\}$ is a set of their respective finite domains, and *C* is a set of constraints. For any constraint *c* and a set $x_1, x_2, ... x_m$ of *m* variables, *vars(c)* denotes the variables involved, and $rel(c) \subseteq D(x_1) \times D(x_2) \times .. \times D(x_m)$ is the set of combinations (or tuples) of assignments for the variables $x_1, x_2, .. x_m$ that satisfy the constraint.

A tuple $T \in rel(c)$ is called valid when all the values assigned to the respective variables $x_1, x_2, ... x_m \in vars(c)$ are available in the corresponding domains.

For any variable x, |dom(x)| denotes the cardinality of the variable's current domain; and forward degree (fwdeg) denotes the number of constraints with unassigned variables, where *x* is involved in. The arity of a constraint *c* is the number of variables involved in *c*.

For instance, the graph colouring problem from the introduction of the previous section (SAT Solving) is expressed in CP as:

$V \in \{1, 2, 3\}, W \in \{1, 2, 3\}, \ldots$

In CP, constraints need to be imposed to enforce inequality for each edge of the graph:

$V \neq W, W \neq X, X \neq Y, Y \neq Z, V \neq X, \ldots$

In CP, a problem must be specified by defining its constraints and variables. CP provides users with high-level tools for tuning to express structure of the problem and problem-specific knowledge and program the best algorithm for the application at hand. Therefore constrains in CP are directly expressible. Although this ability makes CP capable of tuning, makes it more difficult to handle because the user has to have strong knowledge of these tools in order to use them efficiently. For instance, in CP the order in which variables are instantiated during the search might determine a better performance than another order. This is in contrast to the SAT approach in which CNF formulas would create a black box environment in which external tuning is not generally possible. CP tools provide a rich set of constraints to express the relations between the variables of the problem in the most direct way. For instance, there are constraints to directly express numerical relations such as $2a + b = c$. Also datastructures such as arrays can also be expressed: $a[x] = b$ where a is a variable representing an array, x one representing an index and b a value. There are also constraints to express higher level constraints with a more complex meaning such as a constraint on a set of variables: $\{x_i \mid i \in 1..n\}$ imposing that $\forall i\ \forall j > i . x_i \neq x_j$.

Branching

In CP, there are two main heuristics for branching [32]:

1. Variable and Value Ordering: this follows the fail-first principle [35]: in order to succeed, try where you are most likely to fail, as soon as possible. This means to try and discover dead-ends as early as possible. Heuristics in this category are *mindom* [35] (selects the variable with the smallest current domain size), *max forward degree* (fwdeg) [34] (selects the variable connected to the largest number of constraints with unassigned variables). Alternatively, the solver can simply consider variables according to a user-defined variable ordering (or LEX, standing for lexicographical ordering). The recently proposed weighted degree heuristics *wdeg* and *dom/wdeg* [33] base their choices on information learned from conflicts discovered during search. These heuristics are currently considered amongst the most efficient general-purpose CP heuristics.
2. Intelligent Search Strategies: In order not to get eternally stuck when a branching heuristic makes a wrong choice, evolved search strategies have been proposed in the CP framework to explore the search tree in an intelligent and diversified way. These include:
    - Limited discrepancy search [31] which is based on the assumption that a well-chosen heuristic is wrong only a few times along the sequence of choices. Search therefore starts by applying the heuristics, then exploring other sequences of choices by increasing order in the number of discrepancies (i.e. the number of times where the heuristic is violated).
    - Interleaved Depth-First Search (IDFS, [30]) searches a number of subtrees in parallel in an interleaved manner. The assumption is

that the bad choices that are most important to avoid are the ones occurring at an early branching stage because they can lead to exploring huge subtrees.

Pruning, conflict analysis and backtracking

Propagation methods first appeared in the context of constraint satisfaction problems related to picture processing area [20, 19]. Today's propagation engines are still mainly based on the original algorithms from [20]. The general idea is to reason locally by taking each constraint into consideration in turn. Each constraint reacts to modifications of the variables under its scope, reducing the domains of the other variables of its scope if needed. For instance, the constraint $x \neq y$ will react to an instantiation of the domain of x to a value a by removing this value from the domain of y. The algorithm maintains a queue containing the variables that have been recently modified or the constraints depending on these variables.

An efficient algorithm that removes some values from the domains of the variables with the guarantee of never deleting any solution can be used in place of or in conjunction with constraint propagation. The deduction rules used for pruning part of constraint solvers can be seen as closure operators which help with:

- Narrowing: the operators reduce the domain of the variables.
- Monotonicity: the smaller the initial domains are, the smaller the domains obtained after application of the operators will be.
- Optionally idempotence: applying the operator twice gives the same result as applying it once.

As explained before, detecting a conflict and backtracking from it not only helps the solver not to be eternally stuck in a search space, it also helps avoiding redundant computations and preventing future inconsistencies.

A naïve approach: The simple chronological backtracking technique mentioned before [16] is presented in Figure AE.4 and Figure AE.5. In this algorithm, variables are incrementally instantiated with values. Once the current variable (*v[i]* in the algorithm) is assigned a value, a backward consistency checking is performed against all the past variables. If this consistency check fails, another value is selected for *v[i]* and the consistency check is performed again. If no value can be found for *v[i]* which is consistent with the past variables, then we detect a conflict and therefore the variable that is immediately before *v[i]* (*v[h]* in the algorithm) is uninstantiated and a new value is assigned to it [Figure AE.5]. As explained before, this is a naïve approach since *v[h]* may not have any role at all in the conflict detected.

```
1       FUNCTION bt-label(i, consistent): INTEGER
2       BEGIN
3               consistent ← false;
4               FOR v[i] ← EACH ELEMENT OF current-domain[i] WHILE not
                        consistent
5               DO BEGIN
6                       consistent ← true;
7                       FOR h ← 1 TO i - 1 WHILE consistent
8                       DO consistent ← check(i, h);
9                               IF not consistent
10                              THEN current-domain[i] ← remove(v[i], current
```

```
                                    domain[i])
11                      END;
12              IF consistent THEN return (i + 1) ELSE return(i)
13      END;
```

Figure AE.4 – BT labelling algorithm

```
1       FUNCTION bt-unlabel(i, consistent): INTEGER
2       BEGIN
3               h ← i - 1;
4               current-domain[i] ←domain[i];
5               current-domain[h] ← remove(v[h].current-domain[h]):
6               consistent t current-domain[h] ≠ nil;
7       return(h)
8       END;
```

Figure AE.5 – BT unlabelling algorithm

A more intelligent approach: Conflict-driven back-jumping (or CBJ) adopts a more intelligent approach towards detecting conflicts [16]. Using an example from [32] we attempt to show the CJP approach:

Consider variables $x_1, \ldots, x_5 \in \{0, 1\}$ and the following constraints:

C_1: $x_4 \neq x_5$

C_2: $x_2 + x_3 + x_5 \geqslant 2x_1$

C_3: $x_1 + x_4 = x_5$

Assigning value 0 to x_1 will have the contradictory consequence $0 + x_4 = x_5$ (by C_3) and by C_1: $x_4 \neq x_5$ which is an inconsistency. This inconsistency, however, is not necessarily obvious to propagation-based solvers because each of these constraints separately impose no inconsistency. A solver will be able to detect the

inconsistency once x_4 or x_5 are instantiated. For instance by assigning value 0 to x_4, by C_3 the solver will deduce that x_5 also has to be instantiated to value 0 which leads to a contradiction by C_1.

Constraint solvers can sometimes perform the same deductions several times, leading to unnecessary repetitions in the branches of the tree. In our case, the solver detects the inconsistency by successively assigning values 0 and 1 to x_4, leading in each case to a failure. Failing to detect the inconsistency involving x_4, the exploration of the 2 values for x_4 is repeated for every branch corresponding to an assignment of other variables.

This conflict is independent of the values assigned to variables x_2 and x_3, for instance. The constraints that are violated are C_1 and C_3, which do not involve variables x_2 and x_3. As an analysis of the conflict, therefore, the choice $x_1 = 0$ has to be reconsidered (i.e. any branch involving this choice will be inconsistent). The idea of CBJ is to keep track of the cause of conflicts by maintaining a conflict set that contains the variables involved in the conflicts (*conf-set[i]* in Figure AE.6).

```
1     FUNCTION cbj-label (i, consistent): INTEGER
2     BEGIN
3           consistent ← false;
4           FOR v[i] ← EACH ELEMENT OF current-domain[i] WHILE not
            consistent
5           DO BEGIN
6                 consistent ← true;
7                 FOR h ← 1 TO i-l WHILE consistent
8                 DO consistent ← check(i, h);
9                 IF not consistent
10                THEN BEGIN
11                      pushnew(h- l, conf-set[i]);
```

```
12                          current-domain[i] +- remove(v[i], current-domain[i])
13                           END
14                    END;
15          IF consistent THEN return(i + l) ELSE return(i)
16    END;
```

Figure AE.6 – CBJ labelling algorithm

```
1     FUNCTION cbj-unlabel (i, consistent): INTEGER
2     BEGIN
3           h ← max-list(conf-set[i]):
4           conf-sett[h] c-remove(h, union(conf-set[h], conf-set[i]));
5     FOR j ← h + l TO i
6     DO BEGIN
7           conf-set[i] ← {0};
8           current-domain[i] ← domain[i]
9     END;
10    current-domain[h] ← remove(v[h], current-domain[h]);
11    consistent ← current-domain[h] ≠ nil;
12    return(h)
13    END;
```

Figure AE.7 – CBJ unlabelling algorithm

Figures AE.6 and AE.7 (from [16]) show algorithms for implementing CBJ. CBJ maintains a conflict set (*conf-set[i]* in the algorithm) for every variable. *conf-set* is set to be *{0}* initially. Whenever a consistency check fails between *v[h]* and *v[i]*, *h* is added to the set *conf-set[i] (*line 11 of Figure AE.6 - i.e. *conf-set[i]* is a subset of the past variables in conflict with *v[i]*). If all values are tried in *current-domain[i]* and fail the consistency check, then CBJ jumps back to the deepest

variable *v[h]* where *h* is a member of the set *conf-set[i]* – line 3 of Figure AE.7 where *max-list* function returns the largest integer in a set of integers. When jumping back from *v[i]* to *v[h]*, CBJ carries the information in *conf-set[i]* and the set of variables in conflict with *v[h]* and *v[i]* (*conf-set[h]* becomes *conf-set[h]* $\cup$ *conf-set[i]* - *h*). If further backtracking takes place from *v[h]*, CBJ jumps back to *v[g]* where *v[g]* is the deepest variable in conflict with *v[h]* or *v[i]*.