FROM AXIOMATIC SYSTEMS TO REPLICODE, TO NON-AXIOMATIC SYSTEMS

June 2012

Ólafur Hlynsson

Master of Science in:
Software Engineering from Reykjavík University
Computer Science from University of Camerino

FROM AXIOMATIC SYSTEMS TO REPLICODE, TO NON-AXIOMATIC SYSTEMS

Ólafur Hlynsson
Master of Science
Software Engineering - Reykjavík University
Computer Science - University of Camerino
June 2012

School of Computer Science Reykjavík University
Faculty of Science and Technology University of Camerino

M.Sc. RESEARCH THESIS

ISSN 1670-8539

From Axiomatic Systems to Replicode, to Non-Axiomatic Systems

by

Ólafur Hlynsson

Research thesis submitted to the
School of Computer Science at Reykjavík University,
and the Faculty of Science and Technology at University of Camerino
in partial fulfillment of the requirements for the degree of
Master of Science in **Software Engineering** from Reykjavík University and
Master of Science in **Computer Science** from University of Camerino

June 2012

The undersigned hereby certify that they recommend to the School of Computer Science at Reykjavík University and the Faculty of Science and Technology at University of Camerino for acceptance this research thesis entitled **From Axiomatic Systems to Replicode, to Non-Axiomatic Systems** submitted by **Ólafur Hlynsson** in partial fulfillment of the requirements for the degree of **Master of Science** in **Software Engineering** from Reykjavík University and **Master of Science** in **Computer Science** from University of Camerino.

Date

Marjan Sirjani, Supervisor
Associate Professor, Reykjavík University

Kristinn R. Thórisson, Co-Supervisor
Associate Professor, Reykjavík University

Emanuela Merelli, Co-Supervisor
Associate Professor, University of Camerino

Pei Wang
Associate Professor, Temple University

Luca Tesei
Assistant Professor, University of Camerino

Date

Ólafur Hlynsson
Master of Science

From Axiomatic Systems to Replicode, to Non-Axiomatic Systems

Ólafur Hlynsson

June 2012

Abstract

This thesis is a two part study on axiomatic, and non-axiomatic reasoning systems in the context of systems that provide ambient assisted living.

The first part is focused on axiomatic systems that have a fully specified, predetermined functionality for any given time. We look at an axiomatic system called *ResourceHome* that was designed to detect dangerous spatiotemporal configurations and model *ResourceHome* in a system called $\mathcal{R}$*eplicome*. For that we use a programming language called Replicode which supports non-axiomatic reasoning, along with showing how to represent first-order logic constructs in Replicode.

The second part is focused on non-axiomatic systems. The functionality of these systems is dependent on previous experience as they adapt due to insufficient knowledge and resources. We start by look at a non-axiomatic reasoning system called NARS. We will then do a comparison study between NARS and Replicode to see what they have in common and what sets them apart. Then we will move on to formalize the functionality of Replicode with a general production relation and show how reasoning in Replicode is a special case of production. We will then conclude by giving an idea of how a non-axiomatic version of $\mathcal{R}$*eplicome* might look like.

There are several contributions made in this thesis; first we give a two-step translation algorithm from first-order logic to Replicode. Then we perform a comparison study between NARS and Replicode. Finally we give a partial formalization of Replicode as a production system by defining a general production relation, along with giving Replicode experience-grounded semantics.

Frá frumsetningakerfum til Replicode, til ófrumsetningakerfa

Ólafur Hlynsson

Júní 2012

Útdráttur

Þessi ritgerð er tveggja hluta rannsókn á frumsetninga-, og ófrumsetningakerfum meðhliðsjón af umlykjandi kerfum sem veita aðstoð við daglegt líf.

Fyrri hluti ritgerðarinnar einblínir á frumsetningakerfi sem hafa formlega lýsingu og fyrirfram ákveðna hegðun án tillits til tíma. Við skoðum frumsetningakerfi sem kallast *ResourceHome* sem var hannað til þess að þekkja hættur sem kunna að skapast vegna stöðu hluta í tíma og rými. Við notum svo *ResourceHome* sem grunn að kerfi er kallast *Replicome*. Við hönnun þess kerfis er notast við forritunarmál sem heitir Replicome, sem styður ófrumsetningarökleiðslur, ásamt þvi að sýna hvernig megi setja fram rökfræð i fyrstu stéttar í Replicode.

Seinni hluti ritgerðarinnar einblínir á ófrumsetningakerfi. Virkni þessarra kerfa er háð fyrri reynslu er kerfin aðlagast vegna ónógrar þekkingar og auðlinda. Við byrjum á því að skoða ófrumsetningarökleiðslukerfi er kallast NARS. Þá gerum við samanburðarrannsókn á milli Replicode og NARS til þess að sjá hvað þau eiga sameiginlegt og hvað skilur þau í sundur. Því næst gefum við formlega lýsingu á Replicode með almennu framleiðsluvensli, og sýnum hvernig rökleiðsla í Replicode er sérstakt tilvik af framleiðsluvensli. Við ljúkum svo seinni hlutanum með því að gefa hugmynd um hvernig ófrumsetninga útgáfa af *Replicome* gæti litið út.

Það eru nokkur framlög sem við leggjum fram í þessarri ritgerð; fyrst leggjum við fram tveggja skrefa algrím til þess að túlka fyrstu stéttar rökfræði í Replicode. Annað framlagið er samanburðarrannsókn á milli Replicode og NARS. Að lokum gefum við formlega lýsingu á Replicode sem framleiðslukerfi með því að skilgreina almenn framleiðsluvensl, ásamt því að gefa Replicode reynslubyggða merkinu.

This thesis is dedicated to my family; my mother and father, and two brothers.

Acknowledgements

This thesis is the result of a teamwork.

First I will give credit to my family. My parents: Ólöf Ingibjörg Einarsdóttir and Hlynur Höskuldsson, and my two brothers: Höskuldur Hlynsson and Hlynur Davíð Hlynsson. Without them, this thesis would never have been made.

Secondly, I will give credit to my supervisor, Marjan Sirjani, who gave me a chance to work on hard problems in cooperation with brilliant people.

For guidance and help in the construction of this thesis I would like to thank Kristinn R. Thórisson for the time and resources which he provided which were crucial for making the subject of the thesis a reality. I would also like to give thanks to Helgi Páll Helgason for the lessons on Replicode. Then I would like to thank Emanuela Merelli for the resources needed for the implementation of our case study. Then finally I would like to thank Eric Nivel for reviewing the parts of the thesis related to Replicode and assistance with practical problems related to Replicode, and Pei Wang for reviewing the parts of the thesis related to NARS and the formalization of Replicode.

Publications

Parts of this thesis will be submitted for a peer-reviewed conference.

Contents

Chapter 1

Introduction

This thesis is a journey from axiomatic reasoning systems, to non-axiomatic reasoning systems. We will explore the nature of these systems from the viewpoint of systems that provide *Ambient Assisted Living*, or AAL [AAL-Europe, 2012]. The main goal of AAL is to *assist with activities of daily living* through a combination of electronic devices and software that controls them.

Activities of daily living [MedicineNet, 2012], is by its definition a broad spectrum of activities, though it is mostly focused on providing care for people who have problems with doing the activities of their daily lives and to give them a normal life. As our case study we will use a system called *ResourceHome* [Cacciagrano et al., 2010], which is a framework that can monitor predefined situations like a potential danger or a forgetfulness situation, likes of which Alzheimer's patients have to deal with.

The functionality of *ResourceHome* is described by a formal model which can be verified to satisfy given properties and to function according to specifications. Being able to verify a system certainly has its benefits, such as being able to guarantee that a system will operate as specified in safety critical systems such as airplanes, trains and automobiles.

Axiomatic systems do however have their downsides in the context of *intelligent* systems. These downsides are mostly reflected in the fact that these systems are built to do a set of narrowly defined operations. To be specific, the situations which these systems can handle have been specifically designed and programmed for by a human being. These systems can only display limited, preprogrammed, *adaptation* to changes in the system's environment. Finally, though intelligently designed, these systems do not acquire new knowledge or manifest any *intelligent* behavior.

In this thesis, when we speak of *intelligent systems*, we mean systems that *adapt* under *insufficient knowledge and resources*. We borrow this working definition of intelligence from NARS [Wang, 2006], a Non-Axiomatic Reasoning System. What *adaptation under insufficient knowledge and resources* means is that we assume that the system has limited, and even incorrect, knowledge about the world, and it has limited amount of resources like memory and CPU power, which it will have to specifically manage.

The requirement that the system needs to be able to adapt under insufficient knowledge and resources gives rise to two concepts which will play a center role in this thesis. *Non-axiomatic reasoning*, which allows us to assign a varying *truth value* to statements instead of strict *true* or *false* values. And *experience-grounded semantics*, which means that the *truth value* that a statement has is dependent on the experience that the system has had in the past, rather than being static and derived from a set of axioms from an immutable set of knowledge.

The programming language which we will use in this thesis is a new programming language called Replicode [Eric Nivel, 2012], which is a functional programming language that supports bot axiomatic and non-axiomatic reasoning. Replicode is being designed to facilitate the implementation of an ongoing AI project called HUMANOBS [CADIA, 2012]. One of the main motivation for making Replicode was to build in the passage of time. This means that most objects and operations in Replicode have either a time-stamp or a timespan associated with them. Execution of code in Replicode is dependent on a temporally ordered sequence of objects, which means that Replicode is especially well suited for real-time computation.

As we will see in this thesis, Replicode is essentially a production system which contains production rules which can produce objects or change control values through execution of code. This process is performed by what is referred to as a *reduction mechanism* in Replicode.

1.1 Reasoning systems

One of the central themes in this thesis is the transition from axiomatic and semi-axiomatic systems towards non-axiomatic systems. Before we go on, a short overview of these different types of reasoning systems is in order.

In general, reasoning systems fall into three categories which we will give a brief overview of in order to highlight their differences.

Axiomatic systems: Axiomatic systems operate under the assumption that they have all the necessary knowledge of the domain, and they assume that they have sufficient CPU power and all the memory needed to solve a given problem. These systems therefore do not adapt to any extent.

An example of an axiomatic system is first-order logic where inferences are made upon a set of predetermined axioms with well defined deterministic inference rules. All conclusions that can be derived are based on knowledge that is present within the system which makes adaptation due to insufficient knowledge unnecessary. Similarly, all conclusions which can be derived in finite time will be reached eventually and no attempt are made to find a sufficient, sub-optimal, solution due to time constraints as first-order logic has no sense of time.

This has some serious consequences when considering the *constraint satisfaction problem*[1] which is considered to belong to the *NP complete* complexity class. The consequences are that there might not be enough time in the universe to solve certain problems given the present day computational power.

For this reason, we have chosen to use a system that is based on first-order logic as our case study.

Semi-axiomatic systems: Semi-axiomatic systems are designed under the assumption that knowledge and resources (e.g., memory and available CPU power) is insufficient in some aspects. So they need to adapt to a certain degree in order to solve a given problem.

Many examples of semi-axiomatic systems can be found in machine learning. An example of a semi-axiomatic system that allows for variation in the truth value of statements are systems that learn through decision trees. Decision trees are usually used for data mining on some set of data to estimate a solution to a given problem. Although decision trees allow for variation in the truth values of conclusions, a search through a knowledge base to reach a conclusion is essentially a constraint satisfaction problem, and therefore suffers the same limitations as explained above. Heuristics can be added to the search in order to shorten the time it takes to reach a conclusion, with the side effect of probably reaching a sub-optimal solution. This does however not make the system non-axiomatic, as the knowledge base does not change with experience as hypothesis are tested or conclusions reached based on actual experience.

[1] In short, the constraint satisfaction problem is a search through a knowledge base for a set of atoms which satisfy an equation with certain constraints.

Non-axiomatic systems: Non-axiomatic systems are designed by the principle that they have to adapt due to insufficient knowledge and resources, and that the system has to operate normally under such conditions.

Non-axiomatic systems assume neither sufficient knowledge nor resources. They manage to *bypass* the state explosion problem by providing a special definition of truth and intelligence. These definition give us a frame in which we can objectively reason about what can be considered an intelligent property of a system. Resource and knowledge management are provided by a special control mechanism that makes sure that the system does not use more resources than are available to solve a given problem within a given deadline. Knowledge in these systems can change over time as the systems *experience* more data, as the truth values of statements changes over time, both because the system has learned through experience and if the systems have had more or less resources available to reach a certain conclusion. NARS [Wang, 2006] and HUMANOBS [CADIA, 2012] are examples of non-axiomatic systems.

This overview reveals that these systems differ in the way that they adapt to the constraints under which they must operate, either by design or by the hardware in which they operate.

1.2 Defining intelligence

From the comparison above we see that the difference between these systems lies in the degree to which these systems *adapt* to the constraints that the physical world places on them and to the knowledge that they contain. From that we can be confident in adapting the working definition of intelligence from NARS [Wang, 2006, p. 39], which is as follows.

Definition 1 *Intelligence is the capacity of an information system to adapt to its environment while operating with insufficient knowledge and resources.*

1.3 Contributions

The contributions of this thesis will be argued for in a series of chapters that take the trajectory from axiomatic reasoning systems, to non-axiomatic reasoning systems. We do so by describing two systems that provide ambient assisted living: an axiomatic implementation of *ResourceHome* called *Replicome*, and a description of a non-axiomatic version of *Replicome*.

In this thesis we argue for the following propositions:

Thesis 1: Non-axiomatic reasoning systems can make more intelligent systems that provide ambient assisted living compared to axiomatic reasoning systems.

This proposition guides the structure of the thesis. We start by exploring the nature of axiomatic reasoning systems, their benefits and limitations, and then we move on to explore non-axiomatic reasoning systems. This proposition is important for the reason that if non-axiomatic reasoning systems do indeed provide more intelligent systems that provide ambient assisted living, then research should be guided in that direction. We do not provide empirical support for this proposition, but we do argue in favor for it by showing the limitation of axiomatic systems in general, and how non-axiomatic systems are the next logical step in the progress of creating intelligent systems.

Thesis 2: Replicode's *reduction* mechanism can be described by a *general production relation*.

Replicode is of a relatively new breed of programming languages. It is in core a functional programming language, which has support for both axiomatic and non-axiomatic reasoning. Replicode allows for generation, self-inspection and modification of code at runtime, and has a highly versatile and complex scoping mechanism. Replicode, unlike most programming languages, was not built from a fully formalized theory of operation, and it is therefore of most importance to formalize the core functionality of Replicode such that the capabilities and the limitations of Replicode can be fully explored.

Thesis 3: Replicode can have *experience-grounded semantics*.

The semantics of traditional programming languages are described by model theoretic semantics, where the meaning of statements of the language can be derived from a set of fixed, predetermined, axioms which do not change over time. Though model theoretic semantics can be applied to Replicode, it does little to help explain the nature of systems built in Replicode. As Replicode is designed to build intelligent systems that have their own understanding of the world, which might be different from our understanding of the world, we need a way to explain how an intelligent system came to a given conclusion. It therefore makes sense to say that the meaning of statements in the system can change with experience, instead of trying to prove it with a complicated set of rules and equations. This is why we give Replicode experience-grounded semantics.

Throughout this thesis we present several contributions. First we show how any system specified in first-order logic can be translated into Replicode by using a two-step translation algorithm that translates first from first-order logic to conjunctive normal form, and then a translation from conjunctive normal form to Replicode, where the second step is our original contribution. Then we do a comparison study between a Non-Axiomatic Reasoning System, NARS, and Replicode. There we show that Replicode has the same qualities that are found in NARS which give Replicode *experience-grounded semantics*. Then we move on to formalize parts of Replicode. We give a conceptualization of its grammar by removing all predefined objects from the language such that we can see the simplicity of legal statements in Replicode. We will describe Replicode in terms of being a production system and present a notion for that description called *general production relation*. We will then show how reasoning in Replicode is essentially a special case of a production. Finally we will give Replicode *experience-grounded semantics* by defining *truth*, *knowledge*, and *experience* in Replicode.

1.4 Overview of the thesis

In part one we explore axiomatic systems which are based on first-order logic and give a representation of them in Replicode. We start by giving an introduction to *ResourceHome*, which will be our running case study, as well as introducing the programming language Replicode in Chapters 2 and 3 respectively. Then we show in Chapter 4 how to represent first-order logic constructs in Replicode, along with giving a translation algorithm from first-order logic to Replicode. This translation will help us understand how to build simple systems in Replicode along with giving the option to algorithmically translate all systems specified in first-order logic into Replicode. We will then end the first part by giving an implementation of *ResourceHome* in Replicode in Chapter 5.

In part two we turn to non-axiomatic systems. We start by giving an introduction to NARS in Chapter 6. Then we do a comparison study of NARS and Replicode in Chapter 7 where we map the major theoretical and technical commonalities and differences between the two systems. In Chapter 8 we formalize some aspects of Replicode in terms of being a production system and give Replicode experience-grounded semantics. We finally close the thesis by describing what a non-axiomatic version of *ResourceHome* might look like.

Part I

From Axiomatic Systems to Replicode

Chapter 2

Ambient Assisted Living

The domain of the case study

Ambient Assisted Living (AAL) [AAL-Europe, 2012], provides assistance with daily living through automated software and electronic devices. These automations can range over domestic heating and air conditioning, indoor and outdoor lighting, to audio systems, security systems, robots and much more. These devices are usually controlled by one centralized control and its main goal is to improve the general standard of living of elderly people. AAL can increase the standard of living by providing personalized environmental settings, schedule and execute certain activities automatically, increase domestic security and provide assistance to the elderly and the sick.

In this chapter, we will use *ResourceHome* as our case study, which is a radio-frequency identification (RFID) based framework which can detect and prevent dangerous spatiotemporal configurations. We will also look at a derivation of *ResourceHome*, called ADEOTRS (AssisteD EnvironmenT for Objects Research & Safety control) [Laura Aureli, 2011], which is an implementation of *ResourceHome*.

These systems are essentially axiomatic in nature as they are dependent on a constant set of irreversible premises which can be either be true or false, or multi-valued. Having a system that is based on sound logic allows for building systems that can be statically verified to be safe and function according to the specifications.

2.1 ResourceHome

ResourceHome [Cacciagrano et al., 2010] is an attempt to build a ubiquitous system that can statically detect and prevent dangerous interactions between objects that carry certain properties. These properties are encoded in an ontological knowledge base that can describe properties that can cause dangerous interactions when interacting with other objects that carry certain properties. These objects also carry spatiotemporal information, which is their location in space and time.

The design of the model described in *ResourceHome* enables easy porting to model checkers such as the *Alloy Analyzer*, which allows for analysis on quantitative and qualitative properties, temporal and modal properties, and the topological distribution of safe and unsafe configurations.

2.1.1 Purpose

The purpose for designing *ResourceHome* was to give a framework that can be used to build systems that provide environmental safety control. It can be used both in domestic and industrial contexts which uses sensors and actuators, and is suitable for safe environment design.

The *ResourceHome* architecture is applicable in many different types of environments, e.g., in busy work environments which may contain dangerous objects that move frequently. The architecture is also suitable for environments which range over large areas and can be difficult to track, and for environments where the inhabitants are not fully capable of taking care of themselves due to an illness or a disease like the Alzheimer's disease.

ResourceHome provides safety control by running silently in the background and notifies only if it detects potentially dangerous configuration of the environment. It relieves the inhabitants from the work of tracking the movements of potentially dangerously interacting objects, thereby increasing the standard of living by tracking the overall safety of the environment.

2.1.2 Architecture

The *ResourceHome* architecture is roughly split into two layers; the hardware layer and the software layer.

2.1.2.1 Hardware layer

The hardware layer is composed of five distinct components which are necessary to achieve the implementation of *ResourceHome*. The hardware layer is based on RFID technology that provides a cheap solution to track the movement of objects through special RFID readers.

2.1.2.1.1 Tagged objects

All objects that the system will monitor are tagged with a thin RFID label that can be sticked to objects. These tags carry a unique identification number which guarantees that there is a one-to-one mapping from a RFID number to an object. These objects need to be manually fed to the system.

2.1.2.1.2 Two-dimensional RFID grid

The environment which is to be monitored is equipped with a grid of passive UHF RFID tags that are placed on the ceiling of the environment. The structure of the grid determines the resolution in which the environment can be monitored. It is recommended to build an hexagonal mesh where each tag is placed in the middle of a hexagon which has a diameter of one meter.

2.1.2.1.3 RFID reader

The RFID reader is a prototype of a portable RFID localizer that was engineered at University of Camerino. It consists of a small UHF RFID reader, a RS232-to-WiFi converter, a multiplexer and a five radially distributed directional antennas and has a range of up to two meters. The reader samples the environment, i.e., reads every RFID tag in the environment at a regular frequency and transmits the information through WiFi to wireless access points.

2.1.2.1.4 Wireless access points

A wireless access point is the interface through which the RFID readers relays the information which it extracts from the environment, e.g., the locations and states of objects, and passes on for storage and analysis.

2.1.2.1.5 Personal computer

Finally, there must be a single personal computer on which all the *ResourceHome* software modules can be run.

2.1.2.2 Software layer

The software layer of the architecture is roughly made up of two closely related levels; the low-level for the tagged objects, and the higher-level for dynamically tagged environments. The architecture also details certain processes, or phases, which are meant to help the system achieve optimal performance of the system. The innovative part of *ResourceHome* is the software layer which combines state-of-the-art ontological knowledge management and SAT solver-based model checking techniques which guarantees that the system will monitor safety even though the objects in the systems are changing at runtime.

2.1.2.2.1 Low-level model for tagged objects

The resource semantics of the environment are encoded into an ontology-based model. Two ontologies are described, first one that categorizes the object and the latter is for modeling the environment. These ontologies can be extended by the user to model domain specific ontologies as required by the user. The quantity of objects and their property values can vary over time, and must the system's behavior be independent of these variations. [Cacciagrano et al., 2009, p. 3].

2.1.2.2.2 High-level model for tagged objects

The uppermost level of the *ResourceHome* is described by a first-order logic-based model for dynamic tagged environments. This model can be used to represent any environment with heterogeneous objects as a finite set of resources equipped with a finite set of properties. This model can be verified in the Alloy Analyzer. We will explain this model in detail later in the chapter.

2.1.3 Processes

ResourceHome defines three internal, possibly parallel, phases in which the system operates.

First the systems goes through a *learning phase*.[1] By that we mean that the system has to gain some information about the objects that have been registered into the environment, e.g., what objects share elements in the $Interaction$ set. These informations need to be supplied extrinsically. The system needs also to be put into learning phase if the topology of the environment has changed.

After having gone through the learning phase, the system can go to the *active phase*. In this phase, the server application stores and elaborates information about the environments and the objects that are within it.

The third phase is the *filtering phase*, which updates the server application with new knowledge about the environment. As mentioned above, the system periodically updates the knowledge it has about the environment so we need to make sure that all new objects are accounted for, that redundant objects have been removed, and that the system has the right topology of the environment.

2.1.4 First-order logic-based model for dynamic tagged environments

The uppermost level of *ResourceHome* is formally defined as a four-tuple which contains the universe of the domain, the set of configurations, initial configuration and the transition relation. The model is specifically defined for a set of tagged objects, a set of two-dimensional space coordinates, a set of object properties to be enabled/disabled and a set of (dangerous) property interactions to be monitored

This system is essentially a nondeterministic automation which has no final states. States are defined by the location and the values of the properties of the objects. Transitions are said to take place iff at least one object changes place or its properties change.

2.1.4.1 The RFID^2-model

The RFID^2-model is formally defined as:

Definition 2 *An RFID^2-model $\mathcal{M}$ is a tuple $\langle \mathcal{U}(\mathcal{M}), \mathcal{C}(\mathcal{M}), c_0, \delta \rangle$, where:*

- $\mathcal{U}(\mathcal{M})$ *is the* universe $\mathcal{M}$,
- $\mathcal{C}(\mathcal{M})$ *is the set of* states *or* configurations*, of* $\mathcal{M}$,
- $c_0 \in \mathcal{C}(\mathcal{M})$ *is the initial configuration.*
- $\delta \subseteq \mathcal{C}(\mathcal{M}) \times 2^{\mathcal{C}(\mathcal{M})}$ *is a transition relation.*

[1] The learning phase is a predefined stage in ResourceHome, it does not learn in the same sense as when intelligent systems learn, which involves creating new knowledge, whereas the learning phase is more of a preparation stage for the active phase.

Next we respectively present the definitions for $\mathcal{U}(\mathcal{M})$, $\mathcal{C}(\mathcal{M})$ and $\delta(c_i)$.

Definition 3 *If $\mathcal{M}$ is a RFID^2-model, then $\langle\mathcal{U}(\mathcal{M})$ is the tuple $\langle Pos, Pr, O, Int\rangle$, where:*

- *Pos is a finite* Point *set: $\forall p \in Pos, p.x \in [0...n]$, $p.y \in [0...n]$ $(n, l \in \mathbb{N})$.*
- *Pr is a finite* Property *set: $\forall P \in Pr$, $P.range \in \mathbb{N}$ is the circular aura (scope) ray of P.*
- *O is a finite* Object set*: $\forall o \in O$, $o.position \in Pos$, which represents the o's position in the plan; $o.true$, $o.false, \in 2^{Pr}$, such that $o.true \cap o.false = \varnothing$, which represents the set of enable and disable properties of o,*
- *Int is a finite $Interaction$ set: $\forall I \in Int$, $I \subseteq Pr$ and $I \neq \varnothing$. I represents a set of interacting properties. An example of such a set could be the set $I = \{"flammable", "flame"\}$.*

Definition 4 *If $\mathcal{M}$ is a RFID^2-model, then*
$\mathcal{C}(\mathcal{M}) = C_O(o_1) \times C_O(o_2) \times ...C_O(o_m)$, where:

- *$\{o_1, ..., o_m\} = O$ and*
- *$C_O(o_k) = \{c(o_k) = \langle x_k, y_k, T_k, F_k\rangle \in Pos \times 2^{Pr} \times 2^{Pr}, T_k \cap F_k = \varnothing\}$, for any $k \in [1..m]$.*

Definition 5 *If $\mathcal{M}$ is a RFID^2-model and $c_i = \langle c_i(o_1), ..., c_i(o_m)\rangle \in C(M)$, then*
$\delta(c_i) = \{c_j = \langle c_j(o_1), ...c_j(o_m)\rangle \in C(M) | \Delta(c_i, c_j)\}$ is the set of M's configurations which are reachable from c_i, where:

- $\Delta(c_i, c_j) = \begin{cases} true\ \ if\ \ \exists k \in [1..m] | c_j(o_k) \in \delta_O(c_i(o_k)) \\ false\ \ otherwise \end{cases}$
- $\delta_O(c_i(o_k)) = \{c_j(o_k) = \langle x_j, y_j, T_j, F_j\rangle \in C_O(o_k) | (x_i \neq x_j)$ *or* $(y_i \neq y_j)$ *or* $(T_i \neq T_j)$ *or* $(F_i \neq F_j)\}$

2.1.4.2 Safe and unsafe configurations

Having defined the RFID^2-model formally, we can now partition the set of all configurations into two distinct subsets; the set of safe configurations $Sf(M)$ and the set of unsafe configurations $Usf(M)$. Having those sets defined will help us in giving some sense for the general safety level of the environment, along with helping us define the *Trend2Safe* function in the next section. Note that the location of objects is discrete which allows us to cover the entire environment in finite number of states.

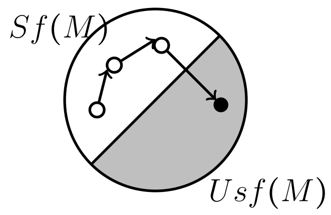


Figure 2.1: Safe and unsafe configurations.

Let us now define the the set of safe configurations, and the set of unsafe configurations.

Definition 6 *If $\mathcal{M}$ is a RFID^2-model then*
$Sf(M) = \{c \in C(M) | Safe(c)\}$*, where:*

1. $Safe(c) = \begin{cases} true \ \ if \ \ \cap_{o \in I, I \in Int, P \in (I \cap_{o.true})} A(0, I, P) = \emptyset \\ false \ \ otherwise \end{cases}$ *, where*
   $A(o, I, P) = Pos \cap (X(o, I, P) \times Y(o, I, P))$
   $X(o, I, P) = [(o.position.x - P.range)...(o.position.x + P.range))]$
   $Y(o, I, P) = [(o.position.y - P.range)...(o.position.y + P.range))]$
2. $Usf(M) = \{c \in C(M) | \neg Safe(c)\}$

Now it is easy to define the global safety level, *G-Safety*, by finding the ratio of the number of safe configurations in the system to the number of configurations in the system. This is essentially the ratio of the number of safe configurations to the total number of configurations.

Definition 7 *G-Safety:*

$$G - Safety(M) = \frac{|Sf(M)|}{|C(M)|}$$

Having a number to describe the global safety level of the system can give us an indication of how *risky* the environment can be, but it does little to help us when operating within the system as this is a static number for a given environment. This leads us to a new function, *Trend2Safe*, which does factor in the current location of an object, and determines the local safety level.

2.1.4.2.1 Trend2Safe

Trend2Safe is a function which gives the ratio of number of safe configurations to the number of all configurations reachable one transition from a given configuration c_i.

Definition 8 *Trend2Safe:*

$$Trend2Safe(c_i) = \frac{|Sf_\delta(c_i)|}{|\delta(c_i)|}, \text{where } Sf_\delta(c_i) = \delta(c_i) \cap Sf(M)$$

A more accurate safety metric for a given state would be to define the transitive closure of *Trend2Safe*. This gives us the composite probability of all safe configuration paths in M, starting from a safe configuration c_i.

Definition 9 *Trend2Safe*:*

$$Trend2Safe^*(c_i, I) = \begin{cases} 0 \ \ if \ \ (Sf_\delta(c_i) = \varnothing) \wedge (|I| = 0) \\ 1 \ \ if \ \ (i \in I) \vee ((Sf_\delta(c_i) = \varnothing) \wedge (|I| > 0)) \\ \frac{1}{|\delta(c_i)|} * \sum_{c_j \in Sf_\delta(c_i)} Trend2Safe^*(c_j, I \cup \{i\}) \ \ otherwise \end{cases}$$

This function can determine if there are any safe successors left, or if we have already visited a given safe configuration. This leads us to the safety function, *Safety(M)*, which is associated to the whole RFID^2-model:

Definition 10 *Safety:*

$$Safety(M) = \begin{cases} 0 \ \ if \ \ c_0 \notin Sf(M) \\ Trend2Safe^*(c_0, \varnothing) \ \ otherwise \end{cases}$$

2.1.5 ResourceHome in Alloy

Having formally defined the upper and the lower level of the software layer, we can analyze these two layers separately.

For the lower level we can reason on *specific automata*, which includes quantitative properties like the number of safe and unsafe configurations in the model, or qualitative properties like how each object determines the unsafety of a given configuration. We can also analyze temporal and modal properties of the models, like the reachability of safe and unsafe configurations, or the topological distribution of the configuration.

The uppermost level allows us to reason on *classes of automata*, which includes properties like the partial ordering on automata where the safety/unsafety levels are fixed but varying classes like the interaction or property sets. Then we can determine the possible existence of inherently safe or unsafe automata.

If the environment changes, we need to go into the learning phase again and run the model through the Alloy Analyzer.

2.2 ADETORS

ADETORS is an implementation of *ResourceHome* with one new type of situation defined, *forgetfulness situation*, and a special *utility function* which gives a given configuration a *utility value*. We will borrow the idea about forgetfulness situations and the concept behind the utility function in Chapter 5 where we implement our version of *Resource-Home*; *Replicome* v.1.0.

ADETORS also defines additional objects and concepts for the implementation of the system which will not be covered here. For more details on ADETORS please refer to Laura Aureli [2011]. The purpose of ADETOR, its architecture and formalism is otherwise the same as in *ResourceHome*.

2.2.1 Utility function

The utility function gives a utility value $u \in \mathbb{R}^+$ which represents how safe or unsafe a given configuration is, where low value of utility indicates low levels of *danger*.

If we define a *danger threshold*, dgr_thr, above which all utility values will be considered to place the system into an unsafe configuration, we can see that this is similar to the idea behind the *Trend2Safe* function in the sense that it gives us an idea of the safety level of the current configuration, without the overhead of exhaustive search in the state space.

The main difference between these values is that the value from *Trend2Safe* is based on complete knowledge about its environment, whereas the utility function is calculated from values in $U(M)$ in the RFID^2-model. The utility value is inversely related to the *Trend2Safe* function; where a high level of utility indicates a possibly unsafe configuration, and a low value from *Trend2Safe* indicates a relatively unsafe configuration.

This is how we define an unsafe configuration with our utility function:

Definition 11 *Unsafe configuration:*

$$uc = \begin{cases} true \;\; if \;\; u > dgr\_thr \\ false \;\; otherwise \end{cases}$$

Although the value from Trend2Safe and the utility function are different in nature, the utility function can *emulate* the *Trend2Safe* function neatly if we choose the right properties for the right kind of utilities.

2.2.1.1 Dangerous interactions

Objects that are possibly *dangerously interacting* are given a specific *property value* which indicates the level of possible danger they will contribute to a given interaction. An example of this would be that the danger caused from exploding 110lbs gas cylinder is a lot greater than the danger from an exploding lighter. This difference is manifested in the objects' property values.

For dangerous interactions, the value of utility, or the *danger value*, is defined to be the product of the property values divided by the distance between the objects. For the special case when the distance is zero, we use a *max* function in the divisor, which either returns a special division factor which is larger than zero or the actual distance between the objects.

2.2.1.2 Forgetfulness situation

Forgetfulness situation is essentially a binary situation, either something is being forgotten, or not. To make that fit into our notion of the utility function, we will formally define a forgetfulness value.

For forgetfulness situations we can define the utility value as: $u_F = forgetfulness * dgr\_thr + 1$, where $forgetfulness$ is a boolean value indicating whether forgetfulness situation has been detected. We do this so whenever something is detected to be forgotten, the utility value will be big enough to set the system into an unsafe configuration.

Note that we assume that a forgetfulness situation is essentially a unsafe configuration.

2.3 Summary

ResourceHome is an axiomatic framework for modeling safe environments with respect to the model that is given for that environment. It gives us a good basis to build a system that can be verified to be safe and operate according to specification by using the Alloy Analyzer.

Before we move on to describe *Replicome*, let us look at an introduction to Replicode, the programming in which we will build our system.

Chapter 3

Replicode

The programming language

Replicode [Eric Nivel, 2012] is a functional programming language that is being developed as a part of the HUMANOBS [CADIA, 2012] artificial general intelligence project at Reykjavìk University. The HUMANOBS project centers around building a system that is capable of learning socio-communicative skills by observing people. The goal of HUMANOBS is to provide a general framework for machines to learn by association by observing its environment. Replicode allows for the implementation of highly modular, fine-grained, reactive and data-driven systems that handle time in an explicit manner and does not make a fundamental distinction between data and code (i.e. all code can be viewed as data).

Replicode is built in the nature of Constructivist AI methodology [Thorisson, 2009]. This methodology has its roots in Piaget's theory of cognitive development, or Piaget's Constructivism [Piaget, 2012], which defines four developmental stages:

1. **Sensimotor stage** where object awareness is developed and motor-capabilities of the body explored. At latter stages they learn how to combine schemes, or *models*, to reach a *goal*, like reaching for a toy.

2. **Preoperational stage** is characterized by *magical thinking*.

3. **Concrete operational stage** where logical thinking begins.

4. **Formal operational stage** where abstract reasoning develops.

Replicode has a syntax which is well suited for mimicking the sensimotor stage. This stage requires a mechanism which can build *cause-effect models* of the world, which can then be used to make predictions about what the result of a given *action* will be. We need to make predictions because the models that we build aren't necessarily correct. This gives the opportunity to adjust, change or throw away the models that aren't satisfactory.

Replicode can also facilitate other aspects of Piaget's theory. Replicode provides a pattern extractor which allows for learning, or automatic model building. This can also allow for *assimilation* given that we have an I/O stream which gives a detailed enough description of the environment. *Accommodation* can then be achieved by by a combination of model building, varying truth values of models through experience and simulation. And finally, by providing a control mechanism, Replicode can also allow for sophisticated adaptation under insufficient knowledge and resources.

3.1 Overview

Replicode is composed of several elements that can be used to build a multitude of miscellaneous systems. A high-level distinction can be made between those elements and can they be split roughly into three different categories.

First there is the basic functionality that should be familiar to all programmers. This is executable code that produces some output or some changes in the state of the system. Objects and values that hold information about the system. And control statements that determine the path of execution and support for I/O devices.

Secondly, Replicode allows for both axiomatic and non-axiomatic reasoning. Replicode can take objects which carry values and use them to reason about them based on the existing knowledge of the world.

Thirdly, there is the possibility of building self-growing systems. This is per se not something that Replicode does automatically, but the necessary framework is in place as a pattern extraction library is provided. Self-growing systems utilize model acquisition and simulations on existing knowledge, they produce hypothesis and assumptions that are either proven to be true or false based on actual experience. Building such a system is one of the goals of the HUMANOBS project.

3.2 How Replicode works

3.2.1 Building domain knowledge

Domain knowledge in Replicode is built through markers. Instances of entities, ontologies and classes and atoms are used in a marker which build a structure of objects which Replicode can pattern match against.

3.2.1.1 Entities

Entities are like entities in the real world. They are used as a symbol to specify to what a certain knowledge applies. Entities can be what ever the designer of a Replicode system wants it to be, whether it is a concrete object like a key or a person, or concepts which represent a qualitative property of an object. Though entities and ontologies can both be used for the same purposes, as they are only labels, it is recommended to use entities for concrete objects and ontologies for something that assigns a value to an entity.

This is how one would define an entity in Replicode:

```
subject:(ent 1) |[]
```

3.2.1.2 Ontologies

Ontology is a concept provided by Replicode to aid in building domain knowledge. Ontology literally means existence, and is used to build a formal representation as a set of concepts within a given domain.

Ontologies can be used to describe classes, relations, attributes, axioms and much more. The concept is used in Replicode in a similar fashion as it is used in information science. In our example, an ontology for the entity `subject`, could be a `position` attribute. This attribute could be defined in terms of a two or three dimensional vector which gives the entity a location in Euclidean space.

An example of how one would define a `position` ontology is:

```
position:(ont 1) |[]
```

3.2.1.3 Markers

Markers are used to build relations between objects. The relations that are built using markers is the foundation of the knowledge that a Replicode system has. If the system is to be aware of any complicated structure, the system has to know the objects that are within the structure and what the relationship between the objects are.

An example of a marker would be to give an entity a location in space. If one would want to give the subject we defined above the location $(2, 3)$, it can be done like so:

```
subject_pos:(mk.val subject position (vec2 3 4) 1) |[]
```

3.2.2 Programs

Programs in Replicode are predefined objects with special functionality. Programs can react to events that occur both internally as a result of a reduction, and externally in the world through input devices. Programs can also be used alone to produce change in the state of a Replicode system by *producing* more objects or change the control values of some objects. When used to react to events, patterns in the program define the structure and constraints on the objects that they are to be reacted to. When there are no constraints that must be satisfied for a program to run, the patterns are empty.

Programs can create other programs, they can provide a predefined response to predefined events, add to preexisting knowledge and make modifications to related or unrelated objects in the system. Objects can not be modified directly in Replicode, old objects are rather replaced by new or improved objects. Programs are seen as executable knowledge in Replicode.

All programs in Replicode have the same structure, they contain several placeholders which serve different purposes. Let us walk through these placeholders such that we can get an idea of how programs work. Programs contain a placeholder for a set of *template patterns* which the program can be instantiated with. When there are some objects in this placeholder, the program must be instantiated with objects that match this pattern. This serves the purpose of being able to execute programs with minor variations without actually changing the program. Then programs defines a placeholder for a set of *input patterns* which the program can react directly to. This, along with the template patterns serve as the inputs of the program. After that we have a set of *guards* that can range over both the template and input patterns. Finally there is a placeholder for executable code, or the *productions* of the program. This is the code which is executed given that the template and input pattern of the program have been successfully matched against and the guards satisfied. As with most other objects in Replicode, programs have a propagation of saliency threshold which we will discuss later in this chapter.

It is also worth mentioning that there are no iterators in programs, nor are there any AND nor OR operators. But there are ways to achieve conjunction and disjunction in Replicode as we will see in Chapter 4.

The following is a structure of a program:

Code Listing 3.1: Program structure.

```
(pgm
[template-patterns]
[input-patterns]
[guards]
[productions]
psln_thr
)
[view]
```

This is a program template to be exact, and can be instantiated many times with different parameters. Template code is similar to class definitions in object oriented languages. All template codes needs to be instantiated in order to be executed by the *executive*.

If there are no template patterns nor input patterns the program will execute when instantiated. The following example shows the structure of the standard *Hello World* example.

Code Listing 3.2: Hello world.

```
hello_world:(pgm |[] |[]
[]
    (inj []
        reply:(str "Hello world!" 1)
        [SYNC_FRONT now 1 1 root nil]
    )
1) |[]

ihello_world:(ipgm hello_world |[] RUN_ALWAYS 0us NOTIFY 1) []
    [SYNC_FRONT now 1 forever root nil 1]
```

3.2.2.1 Instantiations

As mentioned above, template code need to be instantiated in order for them to ‘do’ something. Objects like programs, composite states and models need a special constructor, or instantiation, that has the prefix ’i’ in front of their respective constructor for them to become active. One can for example define a program that injects the string "Hi" into a certain view, but that string would not be injected until the program has been instantiated. This can prove useful when testing programs with small variations in the input of the program to find the best outcome of executing the program.

3.2.3 Patterns

Replicode makes heavy use of pattern matching. All inputs to a program are defined as a pattern, and the executive utilizes pattern matching to perform reductions behind the scenes in order to interpreted the world. Patterns can describe a total and partial structure of objects, along with defining precise and imprecise values on the attributes of objects.

In this section we will see what object structures are in Replicode and how they relate to patterns. We will see how to create a pattern on a specific object structure, then we will conclude this discussion on patterns by explaining what pattern matching is in Replicode.

3.2.3.1 Object structure

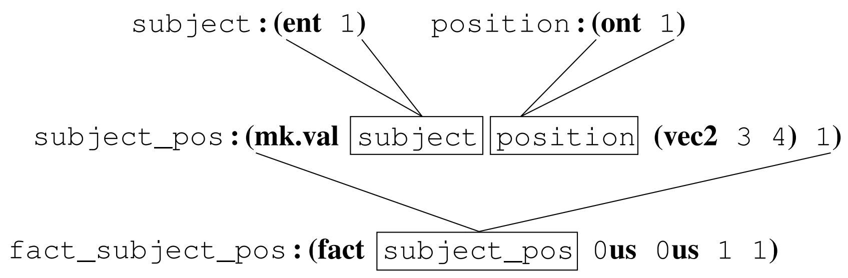
Object structures encode some specific domain knowledge which Replicode can pattern match against. It is best to think about patterns in terms of what they are supposed to match; a specific set of objects which may point to other objects or atoms. Let us take a quick look at what we mean by object structures.

Code Listing 3.3: Object structure.
```
!class (vec2 x:nb y:nb)

subject:(ent 1) |[]
position:(ont 1) |[]
subject_pos:(mk.val subject position (vec2 3 4) 1) |[]
fact_subject_pos:(fact subject_pos 0us 0us 1 1) |[]
```

When Replicode executes the code above, it will build a graph for the labels in the objects which point to other objects. In Figure 1 we can see an illustration of how this is in Replicode; the last statement, `fact_subject_pos` contains a label, `subject_pos`, which is a pointer to the marker object. The marker object then contains two labels: one that points to an instance of the entity class and one that points to an instance of the ontology class.

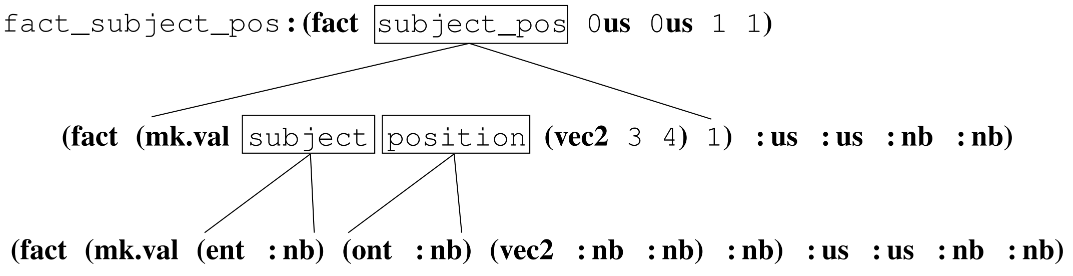
Figure 1 Construction of the `fact_subject_pos` object.


subject:(ent 1)
position:(ont 1)
subject_pos:(mk.val subject position (vec2 3 4) 1)
fact_subject_pos:(fact subject_pos 0us 0us 1 1)

Understanding this underlying structure of objects in Replicode is a prerequisite to understanding patterns in Replicode.

Now, if we look at the object which `fact_subject_pos` points to, we can see that it is made up of many different objects. We can say that it has a specific object structure. This becomes more apparent when we look at Figure 2. If we expand the structure of the object which the label points to, we can see the internal pattern, or the structure of the object.

Figure 2 Expansion of the `fact_subject_pos` object's structure.


fact_subject_pos:(fact subject_pos 0us 0us 1 1)
(fact (mk.val subject position (vec2 3 4) 1) :us :us :nb :nb)
(fact (mk.val (ent :nb) (ont :nb) (vec2 :nb :nb) :nb) :us :us :nb :nb)

The last line in Figure 2 is not a legal skeleton for a pattern in Replicode as it is only a bare-bone structure of the object. If we place labels at appropriate places, we can however turn it into a pattern. Next we will look at how to create a pattern for a given object structure.

3.2.3.2 Creating patterns

Patterns are best explained by an example. Let us create a pattern that matches any entity that has been marked as to be positioned at $(3, 4)$ and is pointed to by a fact. This is the same as the object structure that we just explored.

1. First we want to discard all the elements of the fact object that we are not concerned about **:us :us :nb :nb**, which represents the valid-after and valid-before time-span, the confidence and the propagation of saliency threshold, respectively. We do so by replacing the elements that we want to discard with the wildcard "**: :**".

   Now our original structure:
   (fact (mk.val (ent :nb) (ont :nb) (vec2 :nb :nb) :nb) :us :us :nb :nb)
   becomes:
   ((fact (mk.val (ent :nb) (ont :nb) (vec2 :nb :nb) :nb) ::).
2. Then we discard the propagation of saliency threshold for the marker:
   (fact (mk.val (ent :nb) (ont :nb) (vec2 :nb :nb) ::) ::).
3. We do not care what the entity is that has a position, so we just say that we expect an object of type entity as the first element in the marker class:
   (fact (mk.val #ent (ont :nb) (vec2 :nb :nb) ::) ::).
4. Now we want to place the constant `position` instead of the ontology class because we are looking for all markers where position is the second element:
   (fact (mk.val #ent: `position` **(vec2 :nb :nb) ::) ::)**.
5. Since we are looking for entities at a specific location, $(3, 4)$, we want to declare label for the atoms in the **vec2** object so we can access them, we don't need to declare the type of the atoms:
   (fact (mk.val #ent: `position` **(vec2** `x: y:`**) ::) ::)**.
6. Now we have the skeleton of the object that we want to match. To create a pattern we just need to plug the skeleton into a pattern object:
   (ptn (fact (mk.val #ent: `position` **(vec2** `x: y:`**) ::) ::)** |[]).
7. The last thing we need to do to finish our pattern is to add the guards that specify that $x = 3$, and $y = 4$:
   (ptn (fact (mk.val #ent: `position` **(vec2** `x: y:`**) ::) ::)** [(= x 3) (= y 4)]).
8. We can simplify this guard by setting a constant value to the atoms in the **vec2** class, this way we do not need to use guards:
   (ptn (fact (mk.val #ent: `position` **(vec2** `3 4`**) ::) ::)** |[])

This brief, non-exhaustive, overview of object structures and patterns highlights the basics behind these concepts in Replicode.

3.2.3.3 Pattern matching

Now it is easy to see what pattern matching in Replicode is; finding a set of objects which match the object structure defined in the pattern and assigning the values from the objects to the corresponding elements in the skeleton such that the objects or their values can be further evaluated in guards or used in the executable code.

This process is however only one part of the *reduction mechanism* in Replicode which we will take a closer look at later in this chapter.

3.2.4 Groups

Groups hold a set of views on some objects and serve the purpose of grouping things down into workspaces. The grouping of objects also serves the purpose of restricting the scope of objects where the objects are place in a sub-system of some sort.

Another way to understand groups is by looking at how the mind works. The mind has both a conscious and an unconscious parts which could be looked at as two separate groups. We have no knowledge of what is happening in the unconscious, but despite that, what happens in the conscious can affect the unconscious and vice versa.

This indirect communication between the conscious and the unconscious can be looked at like projections between groups. That is, some objects can be made available to other groups at runtime by placing them in other groups by giving the objects views. With that being said, groups are not totally isolated units.

3.2.4.1 Views

Views hold a set of control values on objects. Views are used to place objects into groups and thereby making them eligible reduction in said groups. The control values that that the views carry control the visibility of that object to another objects in the same group.

One way to understand views is if you look at some random object in your line of sight. You could say that it has a view that is projected onto your conscious group. Some objects are close to you and are easily recognizable and within your reach, while other might be far away and will not become available unless you move closer to them. Finally we can have some objects which might be naturally decaying, like fruits, and can only be available for eating for some period of time. In a nutshell, these are similar to the control values that are in the view.

3.2.4.1.1 Saliency

Saliency is a value which specifies roughly if an object is reliable enough to be considered as an input to a reduction. Each group defines it own saliency threshold, which is a lower bound on the saliency value for objects to be considered as inputs to a reduction. Saliency is a control value, and can therefore be modified by programs.

Another important concept relating to saliency is the propagation of saliency. Let us take for example a fact which is not salient enough to be considered to be an input to a program because the object which it points to is not salient enough. If the object which is pointed to gains saliency, it would be normal to expect that the fact that points to the object would also gain saliency, as they are related. When this happens, i.e., when a fact becomes more salient because the object which it points to gained saliency, it is called propagation of saliency.

3.2.4.1.2 Resilience

Resilience controls for how long an object lives in a view. It basically acts as a decay on the life of object, after witch, the object will be removed from the group. Objects can be set to live forever in a group, or they can be set to live for some predetermined time. The resilience of a view can be adjusted by programs at runtime.

3.3 The Executive

There is a lot of work going on behind the scenes when a Replicode system is running which deserves a quick word in order for the examples to be clear. The system that performs the main work is referred to as the “executive”. The executive is composed of two systems: rCore and rMem.

rCore has the role of executing programs and perform reductions, i.e., performing pattern matching on incoming objects and running all the programs where the input patterns are matched and the guards satisfied. rCore also builds reduction markers, which are notifications that a successful reduction has occurred.

rMem on the other hand has the role of monitoring changes in object values and updating them, detecting programs that have become viable for execution, eliminate duplicate values, and much more.

3.3.1 Reductions

Every group in Replicode is monitored by rMem which creates an instance of rCore which performs reductions on that group. Replicode roughly performs two kinds of reductions; reductions on programs, and reductions on models and composite states. The latter two types of reductions are essentially a special case of the first and are used for reasoning as we will see in Chapter 8.

Reductions in Replicode generally follow the steps listed here below.

Definition 12 *Operational semantics of a reduction* [1]*.*

1. **Pattern match:** *A pattern match is successful if there exists a set of objects O which matches the object structure defined in the input pattern of an instantiated program* [2]*.*
2. **Check guards:** *Guards are satisfied if they all evaluate to true.*
3. **Execute:** *The code which is specified in the set of productions will be executed.*

When a reduction is performed, the executive provides notifications of which programs were instantiated with what objects, and which objects were produced at what time.

This is a somewhat simplified definition of what happens when a reduction is performed, but it does give some idea of how the reduction mechanism works in Replicode.

3.3.2 Execution overlays

The executive can make many reductions on the same program from a set of objects. This is because the executive tries to execute a program with any combination of objects which are available to a program at a given time and match the pattern structures of the program.

[1] This is the general model described in the Replicode specifications paraphrased to match the terminology used in the thesis.

[2] We refer to the input pattern of the program which is pointed to by an instantiated program as its own input patterns. This is a major simplification but will be sufficient for our purposes in this thesis.

3.3.3 External devices

Device functions are the systems means of achieving some goals in the real world. One can connect a device such as a motor control for limbs or any other device that the designer wants to be controlled by the system. When such a device is connected to a Replicode system it can be communicated to and controlled by commands that are executed. Those devices then relay back to the system a notification of a successful or unsuccessful execution of that command. This serves the purpose of providing feedback between a Replicode system and external devices that are necessary for the accomplishments of goals that need to be executed and finished in a specific order. Devices are Replicode objects which typically call out to C++ code (DLL's) to perform commands.

There are no limitations of what kind of devices the system can communicate to. All that needs to be in place is the definition of a given device in the system and the device needs to be able to recognize the commands that are going to be executed through so called device functions.

3.4 Reasoning

Replicode is a programming language which allows for both axiomatic and non-axiomatic reasoning. By non-axiomatic reasoning we mean that truth does not come in absolute *true* or *false* values, but rather with a certain degree of confidence.

For Replicode to be able to reason about its environment there needs to be in place a minimum amount of prior knowledge that Replicode can reason about. This knowledge can be the awareness of objects and actions in the environment, the spatial and contextual relation between objects and actions, or what ever the designer of a Replicode system wants it to know about. Replicode can be used to specify knowledge and algorithms which can be evaluated and reasoned with.

3.4.1 Facts

Facts are used in high-level reasoning and are pointers to other objects. When an object is declared as a fact it carries along with it information about how certain the fact is. No objects are considered for reasoning unless they have been stated to be a fact. Facts, and the reasoning on them, are time dependent. That is, if a model is fired forward or backward because of a fact that will become true after some amount of time, the results of that firing will be considered after that amount of time.

3.4.2 Composite states

As already mentioned, there are no AND or OR operators in Replicode. Composite states were introduced to build states that are *true* if and only if all elements within the composite states are *true*. Composite states are primarily used in conjunction with models when building knowledge.

3.4.3 Models

Models encode a bi-directional knowledge about the world, where each model has a left-hand side and a right-hand side. These models are essentially a Generalized Modus Ponens which allow for forward and backward chaining.

Models do not encode absolute truth about the world, rather it encodes a cause-effect relation about the world. This relation can be enforced or weakened though successful or unsuccessful predictions made on the basis of the knowledge encoded in these models. We will take a better look at how this works in the remainder of this chapter.

3.4.4 Inference

Replicode can perform two kinds on inferences: forward chaining and backward chaining, which are similar deduction and abduction in traditional logic, respectfully. As hinted at earlier, inference in Replicode is essentially a special kind of reduction. There is one important difference between how reductions on models different from reduction on programs, and that is a control algorithm that selects what models will actually produce code.

Definition 13 *Operational semantics of a reasoning reduction.*[3]

1. **Pattern match:** *A pattern match is successful if there exists a set of objects O which matches the object structure defined on either input pattern of the model.*
2. **Check guards:** *Guards for the side which was matched in step 1 are satisfied if they all evaluate to true.*

[3] This definition is based on the descriptions of forward and backward chaining, and how simulation works in the Replicode specifications.

3. **Simulate:** *Simulate the outcome of executing the models and select the ones with the best outcome.*
4. **Execute:** *The code which is specified in the set of productions will be executed.*

Besides the fact that this definition covers both forward and backward chaining, there is one new step which makes this kind of reduction different from reductions on programs, and that is simulation. Also, the objects which are produced in a reasoning reduction carry special *truth values* which are calculated by the executive. We will formalize this notion in Chapter 8.

3.4.4.1 Forward chaining

If some objects are successfully pattern matched against the left-hand side of a model, and the forward guards of the model are satisfied, then a reduction will be performed and an object of type prediction will be produced. This prediction then points to a fact which contains a confidence value based on values in the model and the input facts. This fact then points to the predicted object structure in the right-hand side of the model

The fact that points to the prediction that is produced has a confidence value that is calculated as the product of the success rate of the model and the confidence value of the input fact.[4] This is essentially the truth value of the forward chaining which was performed on the model.

3.4.4.2 Backward chaining

If a goal which points to the right-hand side of a model is successfully pattern-matched against, and the backwards guards of the model are satisfied, then a reduction will be performed and an object of type goal will be produced. The goal which is produced points to the object in the left-hand side of the model which holds the desired object structure.

The same holds for the fact which point to the goals which are produced and the facts that point to the prediction, the confidence value is the product of the success rate of the model and the confidence value of the input fact. The confidence value is interpreted as the likelihood of the goal succeeding.

[4] This is explained better in Section 8.4.2.

3.4.5 Summary

We have just barely scratched the surface of Replicode's functionality. Replicode systems can be immensely complex and require one to think of multiple parallel processes interacting together to produce an emergent intelligent behavior, which is non-trivial.

Some concepts which were not mentioned include Primary and Secondary groups. Primary and secondary groups as essentially a control mechanism for models. They provide a way to manage models that lose saliency in the primary groups and become inactive. Then the model is ejected to the secondary group where it is monitored in case it starts performing well again.

Then there is the concept of Drives which are meant to act like innate drives, think of the lowest level of Maslow's hierarchy of needs.

Finally, the functionality of groups have only been mentioned, but they provide over 30 control values which affect the way they operate. They contain some interesting properties which have yet to be studied in depth.

CHAPTER 4

An Axiomatic Approach to Replicode

In this chapter we will see how to represent first-order logic constructs in Replicode and give a two step translation from first-order logic to Replicode. We do this because our case study, *ResourceHome*, is based on first-order logic-based models for dynamic tagged environments.

We will look at various ways of representing concepts from first-order logic (FOL), from representing knowledge bases to complex formulas. We finally conclude the chapter by giving a two-step equisatisfiable translation from FOL to Replicode via an intermediary format of logically equivalent formulas in conjunctive normal form (CNF).

It is recommended to take a quick look at Appendix A before reading this chapter where we outline some constructs which are used in this chapter, though it is not necessary.

4.1 First-order logic representation in Replicode

This section is dedicated to show how to represent various concepts from FOL in Replicode. As a byproduct of that, we will see how some concepts and constructs simply can not be represented in Replicode.

Some parameters in the examples in this chapter have been omitted for the sake of clarity. Full listing of executable examples from this chapter are available in Appendix B.

4.1.1 Alphabet

Replicode defines a set of characters which are called literals. In general, we have at our disposal the following character-set: A, ...Z, a, ..., z, 0, ..., 9, _, |.

There are however some restrictions on how the last two characters can be used. For further information on these restrictions please refer to Section 2.6 in Replicode specifications [Eric Nivel, 2012].

4.1.2 Constants

Constants in FOL are essentially names of objects or properties in the given domain. In FOL, constants can be negated, but because Replicode has no sense of boolean negation, we assume *negation* as failure of the given constant being evaluated to *true*. Therefore we do not declare negated literals when building a knowledge [1].

Constants in Replicode can be declared either as an entity or as an ontology, as there are no semantics behind entities and ontologies in Replicode, they are essentially just labels. A good rule when designing programs is to define all concrete objects as entities and relations or concepts as ontologies.

When constants are declared, they are considered to be *true*, as there is no way of declaring a *false* constant in Replicode, but we can declare it to be non-existent by using an anti-fact, which is essentially the same as not declaring it at all, we will elaborate on this a bit later.

Let us look at an example. If we represent the FOL constant *p*, which is interpreted to be true, by using the entity *p* and relating it to an instance of the rFOL class by using the *is* ontology. Then this can be done by using the marker value class (mk.val), or simply a marker, and by pointing a *fact* to it.

This is how one would define the constant *p* in Replicode:

Code Listing 4.1: Constant declaration.

```
p:(ent 1)
is:(ont 1)

p_is_true:(mk.val p is (rfol true)); v=true
fact_p_is_true:(fact p_is_true)
```

[1] We can point an anti-fact to the literal to get similar results, but we will keep it simple for now.

Note that we do not necessarily have to point facts to the marker in these examples to execute a program, but they are required for reasoning within Replicode, so we will point facts to them.

4.1.3 Variables

Variables can not be directly converted into Replicode without the usage of programs, models or composite states. Variables that do appear in formulas can be found via *resolution*, which is essentially done by patter-matching.

4.1.4 Functions

In FOL, functions are n-ary, but in Replicode the default markers are 3-ary and there are restrictions on how objects can be placed within the marker. One can define arbitrary classes in Replicode, including markers that accept any given number of arguments. But markers can not be overloaded such that it can accept either one, or two, or n-number of arguments. One can also define an argument as a set, but that excludes the possibility of referencing one particular argument. This means that we can not represent all valid FOL functions in Replicode by using only one type of marker, that is why we use programs.

If we want to represent the fact that p is a letter, we can do it by simply defining an ontology called $letter$ and relate p to that ontology by using our is relation.

This pseudo-code shows how this is done:

Code Listing 4.2: Relating *p* to a *letter*.

```
p:(ent 1)
is:(ont 1)
letter:(ont 1)

p_is_letter:(mk.val p is letter)
fact_p_is_letter:(fact p_is_letter)
```

When dealing with arities larger than three, we need to turn to programs. In programs we can define a set of *input patterns* that allow us to further process the arguments.

Here is an example of how one would define a program that executes if the constants p, q, r and s are true:

Code Listing 4.3: Declaring functions with higher arities.

```
run_if_p_q_r_s:(pgm |[]
[]
    (ptn (fact (mk.val p is (rfol true) ::) ::) |[])
    (ptn (fact (mk.val q is (rfol true) ::) ::) |[])
    (ptn (fact (mk.val r is (rfol true) ::) ::) |[])
    (ptn (fact (mk.val s is (rfol true) ::) ::) |[])
|[]
[]
    (inj []
        a:(answer "Found p, q, r and s.")
    )
)

irun_if_p_q_r_s:(ipgm run_if_p_q_r_s); instantiation
```

Note that the program needs to be instantiated, `irun_if_p_q_r`, in order to run and it will output the string "Found p, q and r." if they have been declared to be true.

4.1.5 Equality

As for functions that take more than three arguments, equality checks must be done by using programs.

Code Listing 4.4: Equality check.

```
run_if_p_is_q:(pgm |[]
[]
    (ptn f1:(fact (mk.val p is (rfol true) ::) ::) |[])
    (ptn f2:(fact (mk.val q is (rfol true) ::) ::) |[])
[]
    (= f1 f2)
[]
    (inj []
        a:(answer "p equals q?" 1)
    )
1) |[]

irun_if_p_is_q:(ipgm run_if_p_is_q)
```

This program will obviously never execute because $f1$ and $f2$ are not the same. Readers with keen eyes will probably wonder why two true statements are not the same. This is because the entities used in the marker are not the same. If one would want this to evaluate to true, one would have to compare variables the rfol class, but not the fact.

One can in a similar fashion check for inequalities by replacing "=" with "<>".

4.1.6 Negation

As mentioned earlier, Replicode has no negation in a logical sense except in patterns. Instead, Replicode defines an anti-fact, |fact, which can represent the absence of some fact. Anti-facts can point to other facts, but the result is not that we end up with the original value negated, as we will see.

In this example, p is declared to be true, a fact points to that declaration, and then we have an anti-fact pointing to the fact.

Code Listing 4.5: Example of an anti-fact on a fact.

```
p_is_true:(mk.val p is (rfol true))
fact_p_is_true:(fact p_is_true)
fact_p_is_not_true:(|fact fact_p_is_true)
```

If we had a program that had one input pattern as shown below, that program would not execute given the knowledge base in code listing above.

```
(ptn (|fact (mk.val s is (rfol true) ::) ::) |[])
```

If we would on the other hand put the structure of that anti-fact in an input pattern, it would execute.

```
(ptn (|fact (fact (mk.val s is (rfol true) ::) ::) ::) |[])
```

This means is that the double negation elimination inference rule from FOL is non-existent in Replicode.

To end this discussion on negation, let us look at what happens if we declare a fact, and an anti-fact on the same marker:

Code Listing 4.6: Contradicting facts.

```
p_is_true:(mk.val p is (rfol true))
fact_p_is_true:(fact p_is_true)
fact_p_is_not_true:(|fact p_is_true)
```

This is completely legal within Replicode as it allows contradictions like this. In terms of FOL, we are allowed to have non-satisfiable equations in Replicode. This means that a direct translation from FOL to Replicode does not preserve satisfiability, i.e., an unsatisfiable equation in FOL could be satisfied in Replicode.

A quick example of such equation is $p \wedge \neg p$, if we place that in a guard for a program with the knowledge base in code listing 4.6, that program will execute.

There is no way around this problem other than making sure that the knowledge base does not contain any contradictions.

4.1.7 Binary connectives

Replicode does not have an AND nor an OR operator, so we need to find a way to represent conjunction and disjunction with other means. This turns out to be a non-issue with respect to conjunction, but things get a bit more complicated when dealing with disjunctions.

4.1.7.1 Conjunction

Conjunction is actually an implicit property of the set of inputs to a program. As we saw when we discussed functions, the program does not execute unless all of the inputs to the program are present at the *same* time.

This means that we can represent a cunjunction on the form: $p \wedge q \wedge ...$ as:

Code Listing 4.7: Conjunction

```
conjunction:(pgm |[]
[]
    (ptn (fact (mk.val p is (rfol true) ::) ::) |[])
    (ptn (fact (mk.val q is (rfol true) ::) ::) |[])
    ...
```

4.1.7.2 Disjunction

In order to achieve disjunction we need to combine three or more functions to disjunct two or more literals by using a method that I refer to as *program weaving*, see Section A.4. What we do when we want to disjunct terms is to create one program per term, where the term is in the input of that program, and let all the programs that make up the disjunction produce a result class which all have the same identification number. Then we create one program that has that identification number in a result class in the input guard of that program, which restricts the execution of that program to a successful match against any of the disjunct terms.

This is best explained with a simple example. Consider the disjunction: $p \vee q$. This can be represented in Replicode by using three programs; two that execute if p or q is noticed, and one that executes if one of these programs executed.

Code Listing 4.8: Disjunction

```
run_if_p:(pgm |[] []; guard on p
    (ptn (fact (mk.val p is (rfol true) ::) ::) |[])
|[] []
    (inj []
        p:(result 0); id for this program
    )
)

run_if_q:(pgm |[] []; guard on q
    (ptn (fact (mk.val q is (rfol true) ::) ::) |[])
|[] []
    (inj []
        q:(result 0); same id as in the program above
    )
)

run_if_p_or_q:(pgm |[] []
    (ptn (result id:) []; guard on the result class,
        (= id 0); id must equal 0
    )
|[] []
    (inj []
        a:(answer "Found p or q." 1)
    )
)
; instantiations omitted
```

Notice that the condition on the input in the last program, `run_if_p_or_q`, is not in the same place as in the last program in code listing 4.4. Where the guard is placed now is intended for conditions on the given fact, whereas the other guard can range over multiple input patterns.

4.1.7.3 Implication relation

The implication relation can not be represented by a simple construct in Replicode. The reason for this can be found in the logical equivalence of the implication relation, which is a disjunction of two terms. And as we have seen, representing a disjunction of terms can be achieved by program weaving, so we can at best simulate the implication relation.

A formula on the form: $p \Rightarrow q$ can be rewritten as $\neg p \vee q$, which is easily representable in Replicode as we have seen.

4.1.8 Quantifiers

Representing quantifiers in Replicode has its own set of challenges as they contain variables which can be contained within a complex FOL formula.

As we saw in the section on disjunction, we need $n + 1$ number of programs to achieve a disjunction of n terms. But because variables in the disjunction might be bound by a quantifier, we need to be able to carry the bound variable to the input of other programs. This can not be done with program weaving, but luckily for us, Replicode allows us to nest programs within programs and produce them on runtime.

4.1.8.1 Existential quantifiers

In practice, existential quantifiers are simply represented as an input pattern for the program. The input can then be used for further evaluations within the program. Below is an example of how one can represent the existentially quantified formula $\exists p, p = p$.

Code Listing 4.9: Existential quantification

```
if_p:(pgm |[] []
    (ptn p:(fact : ::) |[])
[]
    (= p p)
[]
    (inj []
        a:(answer "Found some fact which equals itself.")
    )
)
```

4.1.8.2 Universal quantifiers

Universal quantifiers are implicit in Replicode through overlaid execution. All objects that match a given input are checked against a program, model or a composite state to see if a reduction can be performed.

There is an alternative to performing a reduction on all viable objects, and that is to represent a logically equivalent negation of the universal quantifier. If we look at the formula: $\forall p, p = p$, which means that all instances equal themselves. If we negate that formula we get: $\nexists p, p \neq p$, which means that there does not exist an x such that p does not equal itself. This can be represented easily in an anti-program as we see below.

Code Listing 4.10: Negated existential quantification

```
if_p:(|pgm |[] []
    (ptn p:(fact m: ::) |[])
[]
    (<> p p)
[]
    (inj []
        a:(answer "Found no fact which does no equal itself.")
    )
)
```

The interpretation of this program is that it will execute, if the executive does not find a x does not equal itself within the given time-scope of the program.

4.1.9 Inference

So far we have not discussed models which are Replicode's natural way to reason about facts. A model is essentially a Generalized Modus Ponens mechanism which allows for forward and backward chaining, or deduction and abduction respectively. The reason why we have not represented anything by using models is because they have implicit truth values that can change if contradictory evidence is presented. This would not be in the spirit of FOL, as it is axiomatic.

Truth values in FOL are either *true* or *false* and nothing in between. So if we want to be able to represent FOL in Replicode without the risk of truth values changing, we must use programs, and not models. Please refer to Section 9.2.2.1 for an example of how models can be used.

Despite these limitations, we can still find some of the allowed inference rules in FOL within Replicode. The rest of this section is dedicated to exploring some of the inference rules which are present, some of the inference rules which can be simulated, and some of the rules that do not exist in Replicode.

4.1.9.1 Universal elimination

Universal instantiation can be achieved through either template arguments or by declaring variables in the input pattern of a program. The following is a simple example of universal instantiation. We place an extra guard, $(<> \ m1 \, m2)$, to make sure that the markers are not the same.

Code Listing 4.11: Universal instantiation

```
ui:(pgm |[] []
    (ptn (fact m1:(mk.val v1: ::) ::) |[])
    (ptn (fact m2:(mk.val v2: ::) ::) |[])
[]
    (= v1 v2)
    (<> m1 m2)
[]
    (inj []
        a:(answer "Found two different markers with the same ↙
    object")
    )
)
```

4.1.9.2 Universal generalization

Universal generalization can not be done in Replicode. Refer our discussion on universal quantification for further details.

4.1.9.3 Existential elimination

Replicode does not allow for existential instantiation. New variables, or Skolem constants, need to be explicitly declared by the programmer.

4.1.9.4 Existential generalization

Existential generalization is implicit in Replicode.

4.1.9.5 Reductio ad absurdum

Reductio ad absurdum can not be represented in Replicode. One could try to change the saliency value of the original fact such that it will not be considered as an input to a program. But that could only work if we temporally sequence the evaluations of FOL formulas, because at the same time when the saliency value is changed, the original fact has already be considered as an input to a program.

4.1.9.6 Double negation elimination

As mentioned above, double negation elimination can not be done within Replicode. But if we want, we can define programs that take facts that are wrapped in two instances of anti-facts and produce the original fact, thereby achieving the inference. But this suffers the same limitations as in Reductio ad absurdum.

4.1.9.7 Modus tollens

This is implicit within Replicode if we assume negation as failure.

4.1.9.8 Modus Ponens

Modus ponens can be achieved by placing the left-hand side of the implication relation in the input of a program, and make the program produce the right-hand side of the relation.

4.1.10 Summary

As you might have noticed, some of these methods are neither efficient, nor guaranteed to work for every formula in FOL. That is why we will introduce a two-step algorithmic translation from FOL to Replicode with a pit-stop in conjunctive normal form. That algorithm guarantees to give a logically equivalent representation of any satisfiable equation in FOL in Replicode.

4.2 Two-step translation from FOL to Replicode

In this section, we present a two-step algorithmic translation from first-order logic sentences to Replicode. This algorithm was originally designed to translate the danger table presented in Laura Aureli [2011] into guards on program. Translating simple non-quantified FOL expressions into Replicode is trivial, and can be done algorithmically. For more complex FOL expressions we need to do some simplifications to give a logically equivalent representation in Replicode. It is neither guaranteed that this translation is complete nor sound. The only guarantee that we can give is that it is equisatisfiable.

As explained above, FOL is not directly representable within Replicode, though some aspects of FOL can be represented. In order to give a more reliable translation we have chosen to first convert a FOL formula into its logically equivalent conjunctive normal form. I do this because all formulas in FOL can be converted into CNF and the translation preserves satisfiability.

4.2.1 Step 1: Translation From FOL to CNF

There already exists an algorithm that have been proven to give a correct translation from FOL to CNF in Russell and Norvig [2003], and is the following algorithm borrowed from that work.

1. **Eliminate implications:** Implication relations are eliminated by using the logical equivalence of $p \Rightarrow q$, which is $\neg p \vee q$.
2. **Move $\neg$ inwards:** All negations need to be compliant with CNF, which allows only negations on atoms. The following are rules that can be used to achieve that goal.

   Note that this step might need to be applied several times in order to properly move all negations inwards.

$$\begin{array}{rll} \neg(p \vee q) & \text{becomes} & \neg p \wedge \neg q \\ \neg(p \wedge q) & \text{becomes} & \neg p \vee \neg q \\ \neg\forall x, p & \text{becomes} & \exists x \neg p \\ \nexists x, p & \text{becomes} & \forall x \neg p \\ \neg\neg p & \text{becomes} & p \end{array}$$

3. **Standardize variables:** If a variable is used twice in different qualifiers in the same sentence, change the name of one of the variables to avoid confusion.
4. **Move quantifiers left:** Now all quantifiers can be moved to the left in the order in which they appear.
5. **Skolemize:** This involves removing all existential quantifiers by elimination.
6. **Drop quantifiers:** After Skolemization all quantifiers can be dropped, as all variables must be universally quantified.
7. **Distribute $\vee$ over $\wedge$:** This last step ensures that we have a conjunction of clauses.

After having followed these steps correctly, we should be left with a syntactically simple representation of sentences in first-order logic.

There are a some side-effects from this translation which are worth mentioning. The process of Skolemization changes the meaning of the original formula as it removes all quantifiers. Though the formulas in CNF are logically equivalent to the original formula, they have lost some information. Given that all quantifiers are gone, we do not know the quantitative relation between atoms. This means that this process may not be suitable for performing model checking on knowledge bases.

Now we need only to find a translation to Replicode from conjunction of clauses that contain disjunctions of literals to complete our algorithm.

4.2.2 Step 2: Translation From CNF to Replicode

After having translated a formula from FOL to CNF, the translation of the formula becomes relatively simple.

If we are converting a formula that has a negated literal within it, we can not omit it as we can when building knowledge bases. We must represent them as anti-facts, otherwise a program might execute if the said fact is existent.

The following is a pseudo-code for translating formulas in CNF to Replicode.

1. **Identify all the clauses:** Give all clauses of disjunct literals a unique identification number.
2. **Weave the clauses:** Weave all the disjunct clauses together as shown in Section 4.1.7.2 and let the topmost program output the unique identification number for that clause in a result class.
3. **Conjunct the clauses:** Place all the identification numbers from step 1 in a corresponding result class in the input of a program which evaluates the conjunction.
4. **Notify of success:** Let the program from step 3 output an instance of an answer class with an appropriate message.

This algorithm preserves satisfiability, given that the knowledge base does not contain any contradictions. Thus the two-step translation algorithm as a whole is equisatisfiable. This process also preserves the axiomatic nature of the FOL expression as the guards are essentially Boolean expressions.

4.3 Summary

In this chapter we saw how to represent basic FOL constructs within Replicode. We also gave an algorithmic translation for any valid FOL formula to Replicode through an intermediary format, CNF.

This means that we can take any system specified in FOL and translate it into Replicode by using the two-step translation algorithm presented.

For avid Replicode fans, this process might seem a bit cumbersome, and even somewhat counterproductive. The process does however show that axiomatic systems can be represented within Replicode, which is essentially a non-axiomatic system. This means that axiomatic systems are essentially a subset of non-axiomatic systems.

In next chapter, we will show how to model ResouceHome in Replicode in a system called *Replicome*, which is a first-order logic-based model for dynamically tagged environments.

Chapter 5

Replicome v.1.0

An axiomatic implementation

Replicome v.1.0 [1] is a system that is inspired by *ResourceHome* and ADETORS. In this version of *Replicome* we show how the system is implemented by using only programs. The reason for choosing only programs for this version is to be in the spirit of axiomatic systems, i.e., so we can guarantee that a system that has been modeled and verified to be correct by using Alloy as shown in [Cacciagrano et al., 2010, p. 5], can be represented in Replicode and does not change while operating.

We will use some of the techniques we developed in previous chapter to translate constructs from *ResourceHome* to Replicode, but we will not show a step-by-step translation from *ResourceHome* to Replicode.

As in previous chapter, the code in this chapter has been simplified considerably for clarity, but full listing of executable code can be found in Appendix B.

5.1 System description

The system should be able to recognize objects that carry information about location and a set of dangerous interaction properties that are to be monitored.

These properties are monitored to detect two different types of scenarios:

[1] This system is currently in development and will be available in the summer of 2012 and will be available at: http://cadia.ru.is/svn/repos/ReplicodeOther/Replicome/.

1. **Dangerous interactions between objects**
   Dangerous interactions can be between exactly two objects, which when at the same place and at the same time can be potentially dangerous.
2. **Forgetfulness situations relating to objects**
   Forgetfulness situations should include all cases where the subject forgets an item, or an action, and involve spatiotemporal properties.

The system should be able to recognize dangerous configurations like when a candle is at the same place as a curtain, or if the subject is about to forget his keys before leaving the monitored environment.

5.2 System interface

The interface language of the system is composed of marker values and facts which contain measured information about the environment such as the location and states of objects, which are the system's inputs. In response, the system outputs an alarm about the type and location of the danger if the system is in an unsafe configuration.

All legal statements that the system understands must be legal statements in Replicode. This is to save us the trouble of creating an intermediary language between Replicode and an external system. This requirement does not restrict us in any sense, because Replicode is very flexible when it comes to extending the system.

5.2.1 System inputs

The system's inputs encode two different types of information; the location of objects and the interactions they can have with other objects, e.g., interaction properties.

The system can understand two types of information about the environment.

1. The location of objects through a `position` relation on a **vec2** class that encodes an object's position in two-dimensional space.
2. The interaction between objects in the environment through a `danger` relation on an **interaction** class that encodes and labels interaction between two objects, where one object is a *reagent* and the other is a *reactant* [2].

[2] Reagent is a substance that brings about a chemical reaction, and a reactant is a substance that is consumed in that reaction.

5.2.1.1 Representing locations of objects

As mentioned above, we relate an instance of a **vec2** class to a object to denote its location. The **vec2** class has two members, $x, y \in \mathbb{N}$, that represent the location of objects in a two-dimensional plane.

An example of how objects are given a position is as follows [3]. We assume that the entity `obj` and the ontology `position` have been declared.

Code Listing 5.1: Object location.

```
obj_pos:(mk.val obj position (vec2 3 4))
fact_obj_pos:(fact obj_pos)
```

5.2.1.2 Representing interactions between objects

In this version of *Replicome*, we consider only interactions between two objects for simplification. Later, in Section 5.4, we will show how the system can be extended to take into account interaction between more than two objects.

An example of a tyoe of dangerous interactions can for example be fire, explosion or any other interaction which involve two objects. These types of interactions are represented in the `type` property of the **interaction** class declared below.

We borrow terminology from chemistry when we talk about the role that an object plays in an interaction with another object. Objects are either labeled as being a *reagent*, which are substances that bring about a chemical reaction, or as being a *reactant*, which is a substance that is consumed in a chemical reaction. This labeling is done in a special **interaction** class that also indicates what type of danger is posed from the two objects. An example of a *reagent* can be a match that when lit causes a fire hazard when it comes into touch with the *reactant* oil. The role that an object plays in an interaction is represented in the `role` property of the **interaction** class.

The property value, `propval`, is an arbitrary number in $\mathbb{N}$ that is assigned to a given object to denote the potential danger that an object poses when interacting with other objects. Some objects simply pose a greater danger than other objects, and is that represented in the property value of the object. This is the declaration of the interaction class:

```
!class (interaction type:st role:st propval:nb)
```

[3] These objects would actually be injected into Replicode from an external I/O device.

This design also enables us to monitor potentially dangerous interactions between objects in one simple program as we will see in Section 5.3. But before we move on to describe the system's outputs, let us look at a simple example of how *reactants* and *reagents* are declared.

Code Listing 5.2: Object interaction properties.

```
; Reagent
match_type:(mk.val match danger (interaction "fire" "reagent" 10))
fact_match_type:(fact match_type)

; Reactant
oil_type:(mk.val oil danger (interaction "fire" "reactant" 10))
fact_oil_type:(fact oil_type)
```

This design enables us to declare the same object as a participant in many different *types* of interaction, and it also allows us to declare an object to have an opposing *role* in a different kind of interaction.

This can be expanded to allow for enabling and disabling properties, e.g., to allow for cases when the match is lit, or if the oil is securely contained, but this essentially captures the spirit of the *danger table* from ADETORS so we will let this design to be sufficient for now.

5.2.2 System outputs

The system defines one function which is used to notify external devices about potential dangers.

5.2.2.1 The Alarm function

The **alarm** function takes two arguments; the `what` argument of type **st** that gives us information about what is happening within the system, like a danger/forgetfulness situation, and the `where` argument of type **vec2** that tells us where a given event is occurring.

The declaration of the alarm function is straight forward:

```
!dfn (alarm : :); arg0: what:st, arg1 where:vec2
```

5.3 Programs

Program in Replicome are the same as programs in Replicode, object of type *program* which execute code in response to something in the environment.

We will first show how to represent dangerous interactions in a simple, intuitive but a general way. Then we will show how to represent forgetfulness situations that can be monitored for specific events.

5.3.1 Dangerous interactions

Because of how we defined interactions between two objects, by labeling them as either a *reagent* or a *reactant* for a given type of event, defining a program that monitors potential dangerous interactions between any such pair of objects becomes a breeze. We need only to define one program, `dangerous_interaction`, which can monitor all interactions between objects that have been marked by the **interaction** class.

The `dangerous_interaction` program defines four input patterns and five guards on the inputs. The first two input patterns say that we are looking for two objects where one is specifically a *reagent* and the other is a *reactant*. The latter two objects say that we will also need information about the location of the objects such that we can estimate if they pose any danger at all.

The guards that we place on the objects are twofold; on one hand we are doing identity checks on the object themselves, and on the other hand we are doing an ontological check on the objects.

First we must make sure that the two objects that are being tested as an input to the program are not the same. Then we must make sure that the same objects are used for a given **interaction** and **vec2** class. Due to how Replicode *overlays* the execution of programs, i.e. tries all viable objects to the input of the program and executes all combinations that are successfully matched, we need to place these precautionary guards on the input of the program.

After we have made sure that we have two different objects, we check if they are of the same type and play opposing roles in the interaction. For this we use two guards, one to check if they have the same *type* of danger, and another to check for their spatial proximity.

Code Listing 5.3: Declaration of the dangerous_interaction program.

```
!def dgr_thr 99; danger threshold = 100
!def div_factor 0.9; division factor = 0.9

dangerous_interaction:(pgm |[]
[]
   (ptn (fact (mk.val o1: danger (interaction t1: "reagent" pv1:) ::) ↙
   ::) |[])
   (ptn (fact (mk.val o2: danger (interaction t2: "reactant" pv2:) ::) ↙
   ::) |[])
   (ptn (fact (mk.val o3: position p1:(vec2 : :) ::) ::) |[])
   (ptn (fact (mk.val o4: position p2:(vec2 : :) ::) ::) |[])
[]
   (<> o1 o2)
   (= o1 o4)
   (= o2 o3)
   (= t1 t2)
   (> (/ (* pv1 pv2) (max div_factor (dis p1 p2))) dgr_thr)
[]
   (cmd alarm [t1 p1] 1)
1) |[]

idangerous_interaction:(ipgm dangerous_interaction) |[]
```

There are several labels declared in the input pattern of the program. All the `o1, ...,` `o4` are labels that are used for the identification checks on the objects which constitute the first three guards.

- (<> o1 o2)
  Ensures that the same object is not used for both roles in the interaction.
- (= o1 o4) and (= o2 o3)
  Ensure that the same objects belong to the `position` and `danger` relation.

The other labels, `t1`, `t2` and `p1`, `p2` are used to identify the type of danger that the two objects pose and their location, receptively. These labels are used in the latter two guards that complete the constraints that are placed on the inputs.

- (= t1 t2)
  Ensures that the two objects that play an opposing role in the interaction belong to the same type of danger, e.g. fire hazard.

- (> (/ (* pv1 pv2) (***max*** div_factor (***dis*** p1 p2))) dgr_thr)
  This guard seems a bit cumbersome, but this simply says that the guard evaluates to true if the product of the two property values, $pv1 * pv2$, divided by the maximum of the distance between the objects and the division factor, is greater than the given danger threshold [4].

If all guards are satisfied, the program executes and calls the **alarm** function with information about the type and the location of the danger.

Note that the `dangerous_interaction` program uses two non-standard operators in Replicode, the ***dis*** operator that has been extended to recognize the **vec2** classes, and the ***max*** operator which returns the maximum of two numbers. It will be shown in Annex C how these operators are implemented. Please refer to Annex C for implementation details of these operators.

5.3.2 Forgetfulness situation

We handle forgetfulness situations in a bit different manner than dangerous situations. Instead of defining a general method for identifying forgetfulness situations, we define one program per forgetfulness situation that must be monitored.

For the simple example of notifying if the subject is leaving the monitored environment without its keys, we only need to use the position relation.

Code Listing 5.4: Declaration of the remember_keys program.

```
remember_keys:(pgm |[]
[]
   (ptn (fact (mk.val subject position p1:(vec2 x: y:) ::) ::) |[])
   (ptn (fact (mk.val keys position p2:(vec2 x: y:) ::) ::) |[])
[]
   (= p1 exit)
   (<> p1 p2)
[]
   (cmd alarm ["Forgetting keys" p2] 1)
1) |[]

iforgetfulness:(ipgm remember_keys) |[]
```

[4] Our divisor here is a bit different from the ADETORS implementation, but we can not allow division by zero here, nor anywhere else, in case the distance between the objects proves to be zero!

This example only needs to know the current location of the two objects in order to notify of a forgetfulness situation. The guards are straight forward, the first guard checks whether the subject is at the exit, while the other guard checks if the keys are not in the same location as the subject.

Notice that we use the entities `subject` and `keys` directly instead of defining them as labels. This way we do not have to have special checks on the identity of the objects.

5.4 Extending the system

If we want to add new functionality to the systems, we need only to add or remove programs to *Replicome*. Replicode is designed with code generation and modification in mind, which means that all Replicode compliant objects, such as programs and models, can be injected to the system at runtime. This means that the system can be extended at runtime, and it does not matter whether we want to expand the ontology that we have or if we just want to add a monitor for a new forgetfulness situation.

5.5 Summary

Replicome mirrors both the general spirit of the *ResourceHome* and its functionality to a certain extent. The design of the dangerous interaction program is simple and expressive for many types of common dangerous situations like explosions and fires. The knowledge base of the system can be increased and decreased, and the functionality of the system can be extended at runtime.

But this design is not without its limitations. We might for example need three different objects to cause an interaction. This would have to be dealt with by creating a program per special case, as we saw in the implementation of forgetfulness situations. Another drawback is obviously that there are many other types of dangerous situations that this design can never handle, like when overreaching from the topmost step of a poorly fixed ladder, with the danger being to fall and seriously injuring oneself.

The design of the forgetfulness situation is neither without its flaws. The keys might be in the same location as the subject, but the subject might not necessarily have to physically have the keys. The exit location could also be in the pathway from the living-room to the kitchen, which would throw many false alarms and undoubtedly become irritating quite fast.

We see that we need a new way of thinking about smart systems if we want these systems to be *smart*, or even just remotely intelligent. We need more than simple interfaces and parameter tweaking. In Chapter 9 we will show how we can build smarter system. A systems that can interact with the environment and learn how they works by the way the environment responds to their actions. System that can learn by observing the environment, systems that can learn by observing itself. To do that, we need to leap from axiomatic systems to non-axiomatic systems…

Part II

To Non-Axiomatic Systems

Chapter 6

NARS

An introduction

Non-Axiomatic Reasoning System, or NARS, is a fully formalized intelligent reasoning system presented in a series of publications.[1]

What NARS can do is to take sentences in natural language that carry a special relation [2], or a *couple*, and evaluate the truth of the sentence based on prior knowledge. The logic behind NARS is Non-Axiomatic Logic, or NAL, which is a logic that is composed of eight layers from NAL-1 through NAL-8 which range from low level to high level reasoning. NARS also has a formal language, Narsese, which has a well defined grammar and semantics.

NARS differs from traditional reasoning systems in the way it performs inference. Traditional reasoning systems are axiomatic in nature, that is, the truth value of statements are fixed or can be derived from a set of predefined rules and do not evolve over time. Statements are either considered to be true or false. In non-axiomatic systems on the other hand, truth values are assigned to statements which indicate to what degree the system considers a given statement to be true with respect to its prior *experience*. These values are not fixed, but change as the system evolves or if the system has different amount of resources to answer the question within a certain deadline.

NARS is a composition of several components. It has a Non-Axiomatic-Logic, NAL, which is described by a multi-layered formal language called Narsese. The semantics given for NARS is described by *experience-grounded semantics*, and finally it describes

[1] The most comprehensive overview of the theory is in the book, *Rigid flexibility [Wang, 2006].*

[2] The relation is denoted with a special symbol which corresponds to a word in natural language

several processes which manage knowledge and resources. Let us take a closer look at what makes a non-axiomatic reasoning system.

6.1 Term logic

To give a good sense of what kind of reasoning system NARS is, let us start by looking at the basics; *term logic*. Term logic, sometimes referred to as Aristotelian logic or traditional logic, was the dominant way of doing logic until the introduction of the sound, and complete, predicate logic. Predicate logic got rid of the ambiguity in reasoning, which makes it suitable for mathematically rigorous reasoning. Formal axiomatic systems as predicate logic, have largely failed to bring about *intelligent* machines.

Term logic is essentially made up of three things; major premise, minor premise, and syllogistic rules which allow to draw conclusions from a set of sentences. Let us look at a classic example of syllogistic inference.

All men are mortal
All Greeks are men
∴ *All Greeks are mortal*

The minor premise states that all men are mortal, the major premise states that all Greeks are men, and the conclusion which is reached by a deduction is that all Greeks are mortal.

The premises are more or less based on experience and knowledge about the world. They are also categorical in the sense that they place category of things that are considered men in the category of things that are mortal, and it places the category of Greeks in the category of men.

If we take a closer look at the minor premise, it is actually composed of a quantified (all) subject (men), a verb (are), and an object (mortal). That is, the subject and the verb are connected by a *coupla*, or an *inheritance relation*, to give the sentence a meaning. The meaning comes from the *intensions* and the *extensions* of the terms in the statement. *Men*, are in the *extensions* of being *mortal*, i.e., among the subjects which are mortal, men are included. Similarly for *mortal*, which is in the *intensions* of *man*, i.e., of all the things men are, being mortal is one of them.

But what if we would find a Greek that turned out not to be mortal? How would that affect our model of the world? Should we modify the premises to include all but the special case to keep the logic sound? These kinds of problems can be solved with non-axiomatic logic.

6.2 Non-axiomatic logic

NAL is an eight layer logics that iteratively defines new grammar and inference rules for high-level reasoning. NAL is essentially a an extension of term logic, which allows for reasoning on categorical sentences which are structured in a specific way, as defined by its grammar.

There exists an ideal version of NAL, NAL-0, which is both axiomatic and sound, which means that *truth* has only two forms; *true* or *false*. In NAL on the other hand, *truth*, is actually two values which are assigned to a statement; *frequency* and *confidence*.

If we continue our example of mortal Greeks, if we have two *evidences* of an men that have died, and none that has still not died within the normal age for a human, our minor premise in NARS could look like this:

All men are mortal <1 0.5>

Given that NARS has 1 confirmed case of someone who died, and no case of anybody that old to be considered immortal, the frequency will be 1 and the confidence 0.5 [3]. If we would however discover a Greek which has been verified to be at least 2000 years old, we would have to factor that into our knowledge of men. With one known exception to the rule that all men are mortal, we see the truth value of the sentence change. Specifically, the frequency of the statement decreases and the confidence increasing.

All men are mortal <0.5 0.67>

This is essentially the *non-axiomatic* part of the logic, the *<frequency confidence>* quantification for the truth value. What this allows us to do is to evaluate how accurate a statement is based on the *experience* of the system. An example of a non-axiomatic syllogism can then be as follows.

All men are mortal <0.5 0.67>
All Greeks are men <1 0.99>
∴ *All Greeks are mortal <0.5 0.33>*

[3] Frequency is defined as the number of positive evidences for a statement, divided by the total number of evidences (both positive and negative), or $f = \frac{w^+}{w}$. Confidence is defined as, $c = \frac{w}{w+k}$, we set $k = 1$.

This conclusion would be reached by applying the F_{ded} rule defined in NAL [4]. What the confidence value reveals is that if you have weak evidence for the minor premise, but a relatively strong major premise, the conclusion will reflect that weak premise by giving it a lover confidence.

6.2.1 Defining truth

Truth in NARS is defined by the truth value that a given statement hold, or formally:

Definition 14 *The truth value of a statement consists of a pair of real numbers in [0, 1]. One of the two is called frequency, computed as $f = w^+/w$; the other is called confidence, computed as $c = w/(w + k)$, where k is a positive number.*

6.3 Inference

There are roughly three types of inferences that can be made in NARS; deduction, abduction and induction. Let us have a quick look at these types of inferences:

Deduction is the inference where a conclusion is shown to necessary follow from a major and a minor premise. Or simply to find the conclusion from a set of statements. An example of this could be:
If A implies B, and B implies C, then A implies C.

Abduction is the inference where a probable cause if found for a given statement and a rule. An example for this would be:
Given the fact that the window is closed, and the fact that windows can be closed by humans, a probable cause for the fact that the window is closed is because somebody closed it.

Induction is a bit different from the other two where we have a known rule. Induction is a generalization from a set of facts where the rule for a given relation is found. An example of an induction could be:
If most A's are B's, and all A's are C's, then most B's are C's. This essentially allows to reason about a probability or an uncertainty. This is understandably, the most elaborate part of the inferencing mechanism in NARS.

[4] The frequency is calculated as $f = f_1 * f_2 = 0.5 * 1$, and the confidence as $c = f_1 c_1 * f_2 * c_2 = 0.5 * 0.67 * 1 * 0.99 = 0.33$.

For more information on the types of inferences that are allowed in NARS, please refer to [Wang, 2006].

6.4 Experience-grounded semantics

Given the fact that truth is defined as two values that have the range $[0, 1] \in \mathbb{N}$ which are a function of the system's experience, it makes little sense to use traditional model theoretic approach to define the meaning of a sentence, even though these truth values can be described as a function of a given input stream (experience).

The difference between the traditional model-theoretic semantics, and an experience-grounded semantics is that models are static, i.e., they do not change over time, whereas experience stretches out over time. Model is also a complete description of some universe, where experience is a partial description. Models must also be consistent, where experience can be inconsistent.

This is why NARS has *experience-grounded semantics*, but not model-theoretic semantics. What EGS does is to give us more theoretical flexibility, but it does so in a mathematically rigid way.

6.5 Learning

Learning, in general, is the acquisition of new knowledge. Learning in NARS is achieved in five different ways.

1. The system can learn new things by performing inferences on sentences. When an inference is made, the system generates new beliefs which are added to the knowledge base of the system.
2. If the beliefs produced in the previous step are already existent in the knowledge base, the system merges the new beliefs with the old ones and modifies the truth values.
3. The meaning of a term in NARS is determined by the beliefs in which the terms appear, as new beliefs are generated and old beliefs are forgotten, the system learns the meaning of those terms according to its experience with the terms.
4. The meaning of compound terms can be dissected into new beliefs as the system evolves with experience.

5. Finally, by adjusting priority values in terms, tasks and beliefs, and throwing out the useless ones, the system learns what is important and what is irrelevant. This learning is achieved through a special control mechanism.

These items can further be grouped into three different categories, learning through inference, semantic learning and controlled learning. So far we have covered the first two categories, so let us conclude this introduction to NARS by taking a brief overview of the control mechanism in NARS.

6.6 Control mechanism

The control mechanism makes sure that NARS adapts under the assumption of insufficient knowledge an resources. That is, given a task with a deadline, the control mechanism make sure that the beliefs with the highest degrees of truth gets selected to finish the task within the given deadline. The control mechanism also adjusts these truth values in accordance with its experience such that the beliefs selected to finish a task can vary over time.

This is of course a very high-level and simplified overview of the control mechanism in NARS, but it gives an idea of the purpose that it serves.

6.7 Summary

As we have seen, NARS is quite different from traditional axiomatic reasoning systems. It has a pair of truth values which determine the degree of truth that a sentence carries. And the meaning of truth in the system, or its semantics, is determined by the system's experience, rather than a static, immutable, model.

Even though the mathematical rigor has been somewhat relaxed in NARS, it is still based on sound non-axiomatic logic, NAL, where the semantics of the system can be derived from a set of inference rules in NAL and an input stream, or *experience*.

In next chapter, we will look at the similarities and differences between Replicode and NARS with the purpose of applying some of the formalisms described in this chapter to Replicode.

Chapter 7

Replicode and NARS

A comparison study

In order to see how applicable *experience-grounded semantics* (EGS) is to Replicode, we need to have an understanding of how Replicode and NARS [1] are alike and how they differ. This chapter is dedicated to measuring Replicode [2] up against NARS [3].

7.1 General overview

NARS is a system that is created from work done on *the new theory of intelligence* and is as such a full implementation of an AI system, or a reasoning system. Where Replicode is a programming language that was designed to facilitate the implementation of the HUMANOBS architecture, which is a full implementation of an AI system for a humanoid project [CADIA, 2012].

With that being said, a comparison study between a programming language and a reasoning system might seem odd. But when considering the end goal of the comparison study, to apply EGS to Replicode, it will become clear by the end of this chapter that this study

[1] In this chapter, NARS is used interchangeably with the system proposed in [Wang, 1995], and the thesis in which it was presented.

[2] In NARS, the word "model" is used in the traditional model-theoretic sense, i.e., as a complete description of the environment, but a partial model of the world is called *experience*. In Replicode, "model", as presented in the Replicode specifications [Eric Nivel, 2012], describes a specific high-level construct that encodes an *implication* relation between facts.

[3] In the Replicode specifications [Eric Nivel, 2012], the word "induction" is used for *backward-chaining*. This terminology is often used when discussing backward and forward chaining, and is similar to what is traditionally referred to as *abduction*. To minimize confusion I will refer to backward-chaining as abduction in this chapter.

compares the theory on which NARS is built, to the theory on which Replicode is built and in that context the comparison study makes perfect sense.

The major commonalities and differences between Replicode and NARS are outlined in table 7.1. This high-level overview reveals that they share a common theoretical ground, i.e., NARS is a *non-axiomatic reasoning system* by definition, and Replicode supports *non-axiomatic reasoning*. Replicode is not an *intelligent* reasoning system in and of itself in the same sense as NARS is, but Replicode does provide the platform on which intelligent reasoning systems can be built. This means that Replicode can support various notions of truth, as the one that is presented in NARS.

A major difference between the systems is that through NAL, NARS can infer new knowledge from existing knowledge through its inferencing mechanism. Replicode can however not acquire new knowledge as it comes out of the box, but it can facilitate learning, as shown in the HUMANOBS architecture, where model acquisition is achieved through an extension of the *executive*. It should be noted though that a *patter-extractor* is provided with the distribution of Replicode which tries to make sense of an input-stream, but that is not a part of the core of Replicode.

Another difference between the systems is in their interface languages. In short, *sentences* in Replicode are just structures of objects and no meaning is placed on that particular structure (excluding models). Where in NARS, sentences are given meaning through the *inheritance relation* that they carry. This inevitably leads to some differences in the semantics of the languages.

7.2 Requirements made by NARS

This section explores the general requirements for a system to be considered as a *non-axiomatic reasoning system*, as described by NARS, and compares these requirements against Replicode.

The purpose of this section is to see if Replicode provides the necessary foundation for systems built in Replicode to be considered to be intelligent reasoning systems, as defined by NARS.

7.2.1 Characteristics of a reasoning system

NARS defines some general characteristics that describe a *reasoning system*.

	Replicode	**NARS**
Primary motive for design:	To meet the need for high-performance, distributed logic, where time, induction and abduction are first-class citizens. Also to provide a simple syntax for self-inspection and modification of code.	To create a general-purpose thinking machine.
Working definition of intelligence:	Based on Constructivist AI assumptions [Thorisson, 2009]. Otherwise in complete agreement with NARS' working definition of intelligence.	Intelligence is the capacity of an information-processing system to adapt to its environment while operating with insufficient knowledge and resources.
Inferencing capabilities:	Can perform deduction and abduction. But can be extended to allow for different types of abductions.	Can perform all of the major three: induction, deduction, abduction. Then there are defined more inference rules such as: comparison, analogy, simplification, resemblance, implication, equivalence, etc.
Reasoning framework:	Pattern-matches on inputs and outputs of models, for deductions and abduction respectively.	Has a non-axiomatic logic (NAL) to support all its reasoning.
Interface language:	All facts in Replicode that can be reasoned about point to other objects, which eventually point to marker values (mk.val), which has a entity-ontology-object syntax. There are no semantics encoded in the entities and the ontologies, so one could equally use just entities or ontologies.	Knowledge in NARS is represented by a subject term and a predicate term which are related by a *coupla.* in a *term logic* fashion.

Table 7.1: A high-level comparison of different aspects of Replicode and NARS.

A reasoning system must have:

1. **Formal declarative language** for communication internally by the system, or between the system and a user.

2. **Semantics for the language** that determines the meaning of the words and the truth values.

3. **Set of inference rules** that can be used to reason about current knowledge and derive sub-questions from sections, and so forth.
4. **Memory** that can store questions and knowledge.
5. **Control mechanism** for resource management and selection between competing knowledge.

Replicode is a declarative language which can be used for communication internally within the system and between the system and a user.

The truth values in Replicode can be determined and will a formalization of how truth values are derived be given in next chapter.

Replicode does define two inference rules, deduction and abduction, which can be used to match facts against and to derive sub-goals.

Replicode does have an elaborate memory or workspace architecture, called *groups*, which store knowledge in the form of models, and a memory management mechanism to accompany it.

Replicode does provide a crude resource management mechanism without defining any additional control mechanism. The control is achieved through saliency manipulation by the executive. Replicode also provides a model selection mechanism through *simulation*. So Replicode does provide a control mechanism that allows the system to *adapt under insufficient knowledge and resource*, but not in an optimal fashion as that would need additional control mechanisms.

7.2.2 System restrictions

NARS needs by definition of intelligent reasoning systems to be able to adapt under insufficient knowledge and resources. This means that the system must operate normally under the following restrictions.

1. **Finite**
   There must be a constant information processing capacity. That is, we can not expect more CPU power or memory when needed.
2. **Real-time**
   There must be time constraints on all tasks, i.e., we can not assume infinite amount of time to solve a given task.

3. **Open**
   No constraints can be on the content of the knowledge that the system can accept (given that the knowledge can be represented in the interface language).

Replicode *is not* finite. Replicode was designed with distributed computation in mind so the information processing capacity can be increased or reduced. Replicode can however operate normally on a constant information processing ability, so it can be finite.

Replicode *is* real-time. All *goals* and *predictions* have a deadline, or a time-interval, attached to them in which the goal should be achieved or the prediction should become true. This means that the system has to pick the best *known* model for a given task, albeit not necessarily the best.

Replicode *is* open. There are no domain dependencies on the knowledge that can be represented in Replicode. Replicode is also open in the sense that it handles knowledge that is inconsistent with current knowledge. Replicode also fails gracefully if goals can not be achieved with the given knowledge or resources.

7.3 Interface languages

The way in which a humans or computers, communicate with Replicode and NARS differs substantially.

NARS accepts knowledge in the form of statements that are much like natural language and allow for the categorization of things. The knowledge part of statements in NARS are represented in *extended term logic*, which is the relational framework behind NAL and is both intensional and extensional. In NARS, statements about the world and questions about them are stored in memory until it is decided by a control mechanism that they are not needed anymore. This allows for powerful inferencing on a limited amount of knowledge, and allows NARS to generate all truths that can be derived from the system at a given time with respect to the inference rules of NAL.

Replicode on the other hand has an elaborate syntax which allow for real-time device interaction. But with respect to reasoning, statements, or objects, can roughly be reduced to predictions, goals, models and composite states which are pointed to by facts. These are predefined objects that carry truth values which the inferencing mechanism in Replicode understands. Knowledge in Replicode is therefore all objects that the inferencing mechanism in Replicode understands. Facts are ephemeral, which means that they live for only a limited amount of time and are not stored for later uses. The lifetime of facts is also dependent on the *resilience* of the facts, which might skew the given time interval a bit. One can however store facts about the world in an external system which can then be used for further processing.

7.4 Reasoning

NAL, the logic behind NARS, is composed of the language *Narsese*, inference rules from *extended term logic* (ETL), plus semantics [Wang, 2010]. NAL allows for categorical reasoning, and has therefore a relation to Aristotle's syllogistic logic, as for every valid syllogism in Aristotle's logic there is a corresponding (valid) inference rule in ETL [Wang, 2006, p. 59].

Replicode has only two inference rules which allow deduction and abduction on knowledge [4]. These models encode an *implication* relation between two terms, $A \rightarrow B$. The equivalent to Replicode's implication relation in NARS is found in NAL-7, and can be expressed as A =/> B. The interpretation of this statement is as follows: *"If A is noticed, B is expected."*

The interpretation of this is in Replicode is that when A is noticed, given that all guards are satisfied, the system will output a *prediction* that it will encounter B. If B is encountered, the prediction was successful and the model is reinforced as the success rate of the model is increased. If B is not encountered, the prediction failed and the success rate of the model is decreased.

These models can also be used in reverse, that is, given the desired state B, one can output a goal for reaching state B, and the model can be used to achieve A, which will inevitably result in B, as specified by the model. There is one thing that needs to be emphasized here, and that is that in order for Replicode to actually be able to *do* something that will achieve the goal, A must be a device action which is at the system's disposal.

7.5 Non-axiomatic truth

Both systems are non-axiomatic in nature, so they have a *fuzzy* notion of truth. As a simplification, truth in a non-axiomatic system is the system's best current estimation at a given time.

[4] We will define exactly what knowledge means in next chapter.

Statements in NARS are on the form: $S \sqsubset P < f, c >$. That is, a statement of two terms related by a an inheritance relation, $\sqsubset$, followed by the frequency of the judgment and its confidence. The frequency and the confidence values essentially encode the non-axiomatic part of the statement, which is used to evaluate how accurate judgments are that use that given knowledge in its reasoning.

7.5.1 Frequency

The frequency of a judgment in NARS is defined as the number in (0,1) of positive evidences, $+w$, divided by the total number of times the inheritance relation/model is checked, w. Or simply, $f = +w/w$.

In Replicode, this number is equivalent to the *success rate* of the model.

7.5.2 Confidence

The confidence of a judgment in NARS is the number w, as defined above, divided by $w + k$, where k is a positive constant. Or simply, $c = w/(w + k)$. The constant k is intended to act as a reliability threshold of some sort, giving only high confidence to judgments that have a well established, or stable, frequency.

Replicode has a corresponding confidence value encoded in facts, but it is calculated in a different manner. For facts that come from I/O devices, the confidence value is always set to one. For facts that point to goals and prediction as a result of a backward or forward chaining, the confidence value is calculated as the product of the success rate of the model and the confidence value of the input fact. The confidence value of the fact that the goals and the predictions hold is interpreted by the executive as the saliency value of the fact that holds the goal. This means that some goals or predictions are not executed because their saliency is lower than the saliency threshold of the group.

Even though the way of calculating the confidence value differs between Replicode and NARS, they essentially serve the same purpose; to allow the system to adapt under insufficient knowledge and resources.

7.5.3 Replicode specific values

Models do encode some additional values that are not directly present in NARS so here is a quick overview of those values.

7.5.3.1 Strength

This number indicates if the model has performed well in the past. This number is managed automatically by Replicode.

7.5.3.2 Number of evidences

This number corresponds to w, i.e., the number of times a knowledge has been tested, in NARS.

7.5.3.3 Derivative of the success rate

This is simply the success rate of the model prior to last evidence. This number, along with the success rate, is used to track the progress of the success rate of the model. This number is supplied to control learning, so it has no effect on how truth is interpreted in the system.

7.5.3.4 Template arguments, patterns, forward/backward guards, output groups

These additional arguments are used when the models is executed or evaluated for execution. So far we have assumed empty sets in all but the output groups, where we want the output to be our *workspace*.

When the other arguments are used, along with a clever organization of groups, we step into the realm of *contextual non-axiomatic reasoning*, which is a whole category on its own, and will be explored further in Chapter 8.

7.5.4 Is “truth” in NARS the same as “truth” in Replicode?

Yes, and no. Some values are calculated in a different manner between the systems, so if Replicode and NARS were *seeded* the *same* knowledge, given the same experience, the systems could evaluate the same truth with a different confidence. If we look at the fact that truth can be time-dependent, both Replicode and NARS respect that notion, but Replicode has a fine-grained methods of handling the relationship between truth and time. But the truth is the same in Replicode and NARS in the sense that they are both derived from *experience*.

7.6 Semantics

NARS receives statements, or facts, that carry some *semantics*, or a meaningful relation, about the world. When NARS receives more information about the, the truth values of other statements are updated given that the statements are related to via inheritance relation.

Replicode also receives statements about the world, usually through I/O devices, but they carry no semantics, i.e., they are just a pattern of objects. In Replicode, all knowledge is encoded in bi-directional models, and the objects that they produce, which the system can use for deductions and abduction upon. Knowledge in Replicode is also encoded in composite states, predictions and goals, as they contain, in combination with models, meaningful information about the world. The truth values of knowledge, or models and the objects which they produce, is updated in accordance to the system's experience.

So one can fairly say that knowledge in Replicode can be experience-grounded, given priori knowledge at birth and an input-steam that contains some feedback about previous outputs.

7.7 Summary

This comparison shows that the two systems are in many respects quite different from one another. They were designed with a different purpose, they have different capabilities, and different architecture.

But, there is one deciding factor which determines the applicability of EGS to Replicode, and that is how *truth* is derived. Knowledge, or truth, in NARS is developed through *experience*, which means that truth is subject to revision as the system acquires new knowledge. This is also true for Replicode, truth values can change over time on the basis of the results of the predictions that models produce. Models are also evaluated and selected based on their performance in a *simulation*. Which means that truth can change when given different amount of time to find a way to achieve a goal.

Given these observations, we can say that a Replicode system, which was provided knowledge at start-up and evolves over time by adjusting the success ratae of models according to predefined rules in the system, we can say for certain that knowledge in Replicode is dependent on previous experience. Therefore, Replicode systems *can* have an *experience-grounded semantics*, as defined in [Wang, 2004].

CHAPTER 8

Formalizing Replicode

AN ATTEMPT

In this chapter we will formalize some aspects of Replicode.

The formalization of Replicode does not come without its complications. Due to the special predefined objects that are part of the language, Replicode has a quite complicated syntax and semantics. Therefore, a BNF description of the language is quite complex and cumbersome to read through.

Some predefined objects in Replicode can trigger the *executive* in such a way that new objects are created, or produced, on the fly. This is called a *reduction* in Replicode.

In addition, Replicode is also a reasoning system. This means that in order to give a proper formalization of Replicode, we must find a way to describe Replicode in such a way that it captures both the reduction mechanism, and the inference mechanism.

Replicode also has a rather complicated scoping mechanism. This becomes apparent when studying the groups that are provided and the ways in which they can interact. What this means is that when we try to give the language semantics, we have to factor in the visibility of objects to each-other, as values of the objects in Replicode decide if they can be used in reductions.

As we have well established throughout this thesis, when a reduction is performed then a set of instructions are executed and objects can be produced.[1] This essentially makes Replicode a production system as defined by OMG [OMG, 2009]. There are however some differences from what OMG defines and how Replicode operates. We will show that

[1] This includes all objects, whether they be custom domain dependent objects or objects that are a part of the language axioms, i.e., objects that execute some instructions.

these differences are irrelevant for the purpose of formalizing Replicode as a production system.

8.1 Overview

We will start this chapter by giving a brief introduction to production systems as defined by OMG in Section 8.2. There we will go through definitions relevant to our purposes of formalizing Replicode as a production system.

Then in Section 8.3 we will move on to define Replicode in terms of being a production system. We will define a reduction by giving a general production relation which captures the operational semantics of a production rule. Then we will give a detailed definition of all the constituent and dependent definitions upon which our production relation depends.

After having given the production relation, we will show how inference in Replicode is essentially a special case of a production in Section 8.4. This will allow us to to give the inferencing rules for forward and backward chaining in terms of the notation that we have established.

We will then move on to give Replicode experience-grounded semantics, by defining knowledge, experience and give the meaning to statements within Replicode in Section 8.5.

We will then finally conclude by looking at an experimental version of the grammar for Replicode, along with looking at the possibility of describing a model as a special case of a causal model in Sections 8.6 and 8.7 respectively. These last two sections are not complete and are in their preliminary stages, but they do serve the purpose of illustrating the underlying simplicity of the syntax of Replicode, and that *models* belong to a class of well defined structures.

8.2 Production rule system

OMG has given specifications for *production rule representation* along with some necessary definitions. These include definitions for production rules for forward chaining inference, procedural processing, along with a general representation for rule conditions and expressions. This specification from OMG for production rule representations is non-exhaustive and does not give definition for production rules for backward chaining inferences, but they do give us a framework to operate within.

Let us take a quick look at the main concepts from OMG's specifications.

8.2.1 Production rule

A production rule is essentially a rule which states that some action should be taken given that some condition is satisfied. These rules are normally represented in the form of: if [condition] then [action-list]. The definition of a production rule is as follows.

Definition 15 *A production rule is a statement of programming logic that specifies the execution of one or more actions in the case that its conditions are satisfied. Production rules therefore have an operational semantic.*

8.2.2 Production ruleset

When these rules are placed in a container which can hold one or more rules and a set of objects, then that container is referred to as a *production ruleset*. The purpose of the ruleset is to group rules together with a common business process, or for it to behave like a functional unit. The objects which are at the rules' disposal are referred to as *data source*. The definition for a ruleset is simple:

Definition 16 *The container for production rules is the production ruleset* .

8.2.3 Rule variable

There are essentially two types of variables defined for production rule systems: *standard variable* that has possibly some initial expression and are defined at the ruleset level, and a *rule variable* which has a type and an optional domain specified by filtering the data source. These variables can either be defined at the ruleset level or at the rule level. At runtime these variables behave in a specific way.

8.2.3.1 Semanitcs of rule variables

The semantics for variables at runtime are defines as follows.

Definition 17 *Standard variables are bound to a single value (that could itself be a collection) within their domain. The value may be assigned by an initial expression, or assigned or reassigned in a rule action.*

Definition 18 *Rule variables are associated with the set of values within their domain specified by their type and filter. Each combination of values associated with each of the rule variables for a given rule is a tuple called a binding. It binds each rule variable to a value (object or collection) in the data source.*

This means that a rule can be considered for execution for all variable values and might be executed more than once for a given datasource. This also means that we can represent the production rule on the form: for [rule variables] if [condition] then [action-list].

8.2.4 Semantics of production rules

The semantics of production rules are defined as follow.

Definition 19 *The operational semantics for production rules of a forward chaining:*

1. **Match:** *the rules are instantiated based on the definition of the rule conditions and the current state of the data source.*
2. **Conflict resolution:** *select rule instances to be executed, per strategy defined for conflict resolution.*
3. **Act:** *change state of data source, by executing the actions of the instances of the selected rule.*

8.2.4.1 Operational semantics for forward-chaining production rules

The ruleset for forward chaining is defined without giving the algorithm for selection of the rules to execute. The following rules are applied for all instantiated rules.

Definition 20 *The operational semantics of forward-chaining production rules are defined as:*

1. **Match:** *bind the rule variables based on the state of the data source, and then instantiate rules using the resulting bindings and the rule conditions. A rule instance consists of a binding and the rule whose condition it satisfies. All the rule instances are considered for further processing.*
2. **Conflict resolution:** *the rule instance for execution is selected by some means such as a rule priority, if one has been specified.*
3. **Act:** *the action list for the selected rule instance is executed in some order.*

8.2.5 PRR-Core

The Production Rule Representation Core, or PRR-Core [OMG, 2009, p. 9], is a conceptualization of a production rule system as seen by OMG. This system is described by a set of UML diagrams and specifications which show how the architectural construction of production rules, rule variables, rulesets, and the components needed to build a production rule system.

8.2.6 Summary

This short and somewhat simplified overview of the specification of production rule systems defined by OMG are much in the nature of what we have discussed so far with Replicode. Production rules in Replicode are either models or programs which are executed if a condition on their input is satisfied, as we saw when we discussed the Replicode executive in Section 3.3. Production rulesets in Replicode are simply groups. The operational semantics which govern the execution of production rules in Replicode are almost identical to the operational semantics we defined for reductions in Definition 12 and for a reasoning reduction in Definition 13. Finally, the PRR-Core describes a system which is analogous to the rCore in Replicode. Given these similarities, it is only logical to formalize Replicode in terms of a production rule system.

There are however some drawbacks to these definitions. It does not provide a platform to reason about production rules in general, as it does not allow us to reason about production rules for forward and backward chaining inference as a special case of a general production. As a consequence, these definitions do not allow us to define inference rules in terms of production rules.

In next section we will give a generalization of a production rule with a *general production relation*, which can be used to describe the execution of a production rule in Replicode (a process called reduction), and the inferences which Replicode is capable of performing.

8.3 Replicode as a production rule system

As we saw above, a production rule system is essentially a system that facilitates the implementation of production rules. In Replicode, production rules are predefined objects of type model or program state which are collectively referred to as *executive objects*. The execution of these objects is managed by the *executive* of Replicode, which a component which is analogous to the PRR-Core in OMG's specifications and executes executive objects.

What we will attempt to formalize in this section is the executive of Replicode. To be specific, we will describe the scoping which takes place through *groups* and are managed by rMem, and the *reductions* which are performed by rCore. We will do so by giving a general production relation, and all the definitions which are needed to do so.

8.3.1 Executive object

As defined by OMG, a production rule is a statement of programming logic that specifies the execution of one or more actions in the case that the guards on the inputs are satisfied. In Replicode, this is essentially an executive object of type program, which provides a skeleton for creating a general production rule in Replicode. All objects that can be represented in Replicode can be used as an input pattern for programs. Programs can also produce any legal objects in Replicode by executing an injection command, and programs can change control data of objects [2].

Let us refresh how the structure of a program looks like:

Code Listing 8.1: Structure of program with instantiation.

```
pgm0:(pgm [tpl-patterns] [input-patterns] [guards] [productions])
ipgm0:(ipgm pgm0 [actual-tpl-arguments]) [set-of-views]
```

Programs in Replicode contain a set of instructions, `[productions]`, which are executed by the executive in a reduction. Productions are essentially a set of instructions which can inject new objects into groups, change control values of existing objects in a group, and execute device functions, to name a few. But in other objects like *models*, only prediction or goals on specific objects can be created. Both of these objects manifest the behavior defined for the production rules above, so we can give a localized version of the definition of production rules for Replicode.

Definition 21 *An executive object, denoted with "$\rightarrow$", is an object of type program or model that can trigger the executive to execute a set of instructions specified in the executive object, given that its input patterns [3] have been matched and guards on the inputs satisfied.*

[2] Control data are variables like saliency and activation, for more information about control data, please refer to the Replicode language specifications.

[3] We make no distinction of template patterns and input patterns defined for programs, models or composite states in Replicode. We simply treat them both as a constraints on the structure of input objects.

The operational semantics behind executive objects is the same as we defined in Definition 12 in Chapter 3.

8.3.1.1 Production relation

Given the definition that we just gave for an executive object in Replicode, we can introduce a production relation that captures the core of the operational semantics behind a reduction in Replicode.

Definition 22 *A general production relation is defined as:*
$I_{t,gr_{\rightarrow}} \xrightarrow{G(I_{t,gr_{\rightarrow}})} P_{t,Gr_{\rightarrow}}$,
where:

- $I_{t,gr_{\rightarrow}}$ *is the set of visible input objects from group* $gr_{\rightarrow}$ *that match the object structure of the input patterns of the executive object at time* t*,*
- $\rightarrow$ *is the executive object which is an object of type program or a model,*
- $G(I_{t,gr_{\rightarrow}})$ *is a set of guards on the set of input objects* $I_{t,gr_{\rightarrow}}$*,*
- $P_{t,Gr_{\rightarrow}}$ *is a set of objects to be produced and injected into a set of groups,* $Gr_{\rightarrow}$*, at time* t*.*

The interpretation of this is as follows. A reduction will be performed which produces a possibly empty set of objects $P_{t,Gr_{\rightarrow}}$[4], which are injected into the set of groups $Gr_{\rightarrow}$, at time t, given that the following two conditions are satisfied.

Firstly, the executive must have found a set of objects, $I_{t,gr_{\rightarrow}}$[5], in group $gr_{\rightarrow}$[6] which match the object structure defined in the input pattern of the executive object at time t. We refer to the objects which match this criteria as being *visible*. This is a great simplification as there are many control values which are used to decide if an object can be used as an input to an executive object. We will elaborate on this later in this section.

Secondly, the values of the objects in $I_{t,gr_{\rightarrow}}$ must satisfy the possibly empty set of guards, $G(I_{t,gr_{\rightarrow}})$[7].

This production relation is general in the sense that it covers all possible reductions which Replicode can perform. It is also context dependent in the sense that it is sensitive to both the *visibility* of input objects, and their values which are used in the guards of the relation.

[4] `[productions]`, from Code Listing 8.1.
[5] `[tpl-patterns]`, `[input-patterns]`, from Code Listing 8.1.
[6] `[set-of-views]`, from Code Listing 8.1.
[7] `[tpl-patterns]`, `[input-patterns]` and `[guards]`, from Code Listing 8.1, where we include the guards which are provided in the pattern objects of the first two.

There are some concepts in this definition which require some more explanation and the rest of this section will be dedicated to defining and establishing the terminology upon which this definition depends.

8.3.2 Groups

An executive object must live in a group in order for Replicode to perform a reduction on that object. This is analogous to the ruleset which was defined by OMG, we will however stick to Replicode terminology and refer to them as groups.

Definition 23 *A group is a set of objects:*
$gr = \{o_{gr}, ..., o_{gr}\}$*, where* o_{gr} *denotes an object in group* gr*.*

When an object is in a group, we say that the object has been *projected onto* the group. This is done in Replicode by injecting an object which has a *view* in its placeholder for views. All objects can have one or more views which can project them onto one or more groups.

Given that there can be many, possibly interacting, groups in which objects can live, it can be convenient to refer to these groups as Replicode's environment.

Definition 24 *An environment,* Env*, is the set of all groups that have been defined:*
$Env = \{gr_0, ..., gr_n\}$*, where* gr_i *is a group that has been defined.*

One final notion that we will formalize for the environment is its life time. Given that time is a first-class citizen in Replicode, we will have to include time in our definitions.

Definition 25 *The lifespan of the environment,* Env_{LS}*, is defined as the set:*
$Env_{LS} = \{0, .., n\}$*, where* n *is the time of shut-down, and* $n \in \mathbb{N}$*.*

We essentially define the lifespan of the system to be the internal time counter in Replicode which starts at zero, and continues counting until the system shuts down. This value is in microseconds.

8.3.3 Visibility of objects

We use the word *visible* to characterize objects which have all the requirements to be pattern matched against the input pattern of an executive object.

There are essentially two factors which decide if an object can be used in a reduction. First the object must be alive, or *visible in time* when the reduction takes place. Then the object must be *visible in memory*, which means that the object must live in the same group as the executive object and have control values that make the object viable.

When an object can be used in a reduction, whether that be the executive object or an input to an executive object, we will refer to these objects as being *visible*.

8.3.3.1 Visible in time

Objects are only considered for a reduction if they are visible in time to the executive object. To formalize this notion, we will define the lifespan of an object, and the time in which a reduction takes place.

What we mean by objects being visible in time is that the object must have been alive, with some margin of error, when a reduction takes place. We are given a special time tolerance, time_tolerance, which allows the system to look forward and backwards in time.

For simplification, we will assume that a reduction is instantaneous, i.e., happens in zero time at time t. But we can split this time into three different timespans; the timespan t_{pm} which denotes the time which takes to successfully pattern match against the input of an executive object, the timespan t_{ge} which denotes the time which takes to evaluate the guards of the executive object, and the timespan t_{pr} which denotes the time from which the first object that was produced becomes alive, to the time which the last object dies, or becomes invisible with respect to time.

Let us start by defining the lifespan of an object.

Definition 26 *The lifespan of an object, o_{LS}, is defined as the set of numbers:*
$o_{LS} = \{t_0, ..., t_n\}$, where t_0 is the injection time of the object, and t_n is the time at which the object does not longer exist in Replicode, and $o_{LS} \subseteq Env_{LS}$.

Now we can formally define a visible object in time.

Definition 27 *A visible object in time, o_t, is the object which is alive within the time tolerance given when a reduction takes place.*

This definition is rather loose, as some objects have injection times and other objects have a valid-after and valid-before timespan defined, but we will let this suffice for our purposes for now.

8.3.3.2 Visible in memory

Objects that are visible to an executive object are the objects which live in the same group as the executive object. Not only do they need to live in the same group, but they also need to have a saliency value above the saliency threshold of the group. We will treat this as a guard on the production as this value can be a function of the objects that participate in a production, so we do not handle visibility due to changes in saliency specifically here [8].

Now we define the, possibly empty, set of groups, defined by a set of *views*, in which an object lives:

Definition 28 *The set of groups, Gr, in which an object, o, lives is:*
$Gr_o = \{gr_i \mid gr_i \in Gr \quad o \in gr_i\}$

Given the fact that objects are only considered as an input for an executive object if they all live in the same group, $gr_o \in Gr_o$, this also defines the set of all objects that can be considered as an input for an executive object.

This is why we will next define the set of all objects that live in a group, gr:

Definition 29 *The set of objects that live in the group, gr, is defined as:*
$O_{gr} \subseteq O = \{o_i \mid o_i \in O \quad o_i \in gr\}$*, where O is the set of all allowed objects in Replicode.*

8.3.3.3 Visible objects

Now we can finally define the set of visible objects to an executive object that can be used in a reduction by joining two of the definitions for visible objects above.

Definition 30 *The set of visible objects to the executive object, $\rightarrow$, which lives in group, gr, at time t is defined as: $O_{t,gr_{\rightarrow}}$*

After having filtered out the objects which can be used as an input for the rule, the operational semantics behind the binding of the variables are similar to the one defined in Definition 17 and 18.

Now we have defined exactly the conditions which need to be met in order for objects to be used as an input for an executive object. Next we will define the guards on these objects which must be satisfied for the executive object to execute.

[8] Either the saliency value is read directly from the view of an object, or it is calculated specifically from the values of the objects which participate in forward and backward chaining.

8.3.4 Guards

Guards are used to limit the reduction to some specific objects and their values. These guards can be evaluated by a Boolean expression which may contain function on the inputs which evaluates to true or false. Input objects can also be copied either fully or partially to the code which is produced.

Let us now define the, possibly empty, set of guards which must be satisfied in order for Replicode to perform a reduction on an executive object.

Definition 31 *A set of guards on a set of input objects, O, which must be satisfied in other for a reduction to take place on an executive object is defined as:*

$$G(O) = \{f_1(O), ..., f_n(O)\} = \begin{cases} \textit{true} \text{ if } \; f_1(O) \wedge ... \wedge f_n(O) \; \text{ evaluate to true} \\ \textit{false} \text{ otherwise} \end{cases}$$

Where G is a set of zero or more functions on one or more objects in the set O that evaluates to either true or false, and the empty guard is always true.

Note that the guard is actually defined as a function or an expression in Replicode. This is because guards are not simple values, but can be a function of some values found in the input objects of the executive object.

8.3.5 Summary

As one can see, there are considerable similarities between the production rule systems defined by OMG and the way that we define Replicode as a production system.

Table 8.1 gives a mapping from the concepts used to describe production rule systems and concepts we use to describe Replicode.

Replicode production system	**OMG's production rule system**
Executive object (models or programs)	Production rule
Executive (rCore and rMem)	PRR-Core
Reduction	Execution of a production rule according to the operational semantics of the production rule
Group	Production ruleset
Input objects	Standard/rule variables
Visible objects	Datasource

Table 8.1: A mapping from OMG's specifications of a production rule system to Replicode production system.

Some concepts, like a production rule and rule variables are naturally analogous to their Replicode counterpart, where as the mapping from input objects to standard and rule variables are not as accurate.

In next section we will build upon our definitions and show how inferencing in Replicode is essentially a special case of a production relation.

8.4 Replicode as a reasoning production system

Replicode can reason on specific types of predefined objects. Knowledge in Replicode is encoded in bi-directional objects of type *model*, and reasoning takes place when the the forward or backward guards on the model are matched, and predictions and goals are produced with a specific truth value. At a closer look, models can be simulated[9] by two programs that calculate these truth values and produce objects that contain them. One program that simulates forward-chaining, and one program that simulates backward-chaining.

Let us look at how a structure of a model with instantiation looks like:

Code Listing 8.2: Structure of model with instantiation.

```
mdl0:(mdl [tpl-args] [lhs rhs] [fwd-guards] [bwd-guards] [↙
   ouput-views] s noe sr dsr)
imdl0:(imdl mdl0 [actual-tpl-arguments]) [set-of-views]
```

Where `s` is the strength of the model, `noe` is the number of evidences, `sr` is the success rate and `dsr` is the derivative of the success rate.

What happens when Replicode makes a deduction, or performs forward-chaining, is that when the left-hand side of the model has been pattern matched against, and the forward guards satisfied, the an object of type fact which points to a *prediction* is produced. This prediction contains a pointer to a fact which points to the predicted object structure with a given confidence value. Based on the outcome of the prediction, the model will either be reinforced or weakened through its success rate.

Conversely, when Replicode makes an abduction, or backward-chaining after the right-hand side of the model has been pattern matched against, and the backward guards satisfied, then an object of type fact which points to a *goal* is produced. This goal then points to a desired object structure which Replicode should try to achieve. Failures to achieve the goal does not effect the success rate of the model.

[9] Programs can mimic models at least conceptually, if not literally.

8.4.1 Defining the truth

Before we go on to give a formal description of the types of inferences allowed in Replicode, let us start by defining what *truth* is.

Definition 32 *Truth in Replicode is a visible object of type fact and carries the truth value confidence, which is a number in* $[0, 1] \in \mathbb{R}$.

Defining truth as this is a natural extension of truth being either true or false, or in $\{0, 1\}$, as has been shown [Wang, 2006, p. 67].

Let us now define what can be considered a truth value in Replicode. The difference between truth and truth values in Replicode is that truth is an object of type fact that contains a confidence value, and truth values are used to derive the confidence value of the fact.

Definition 33 *Truth values in Replicode are the values:*

1. *Confidence, which is denotes the confidence of the object.*
2. *Success rate, which denotes the number of positive evidences divided by the total number of evidences.*

The reason for defining these two values collectively as truth values will also be useful when we give Replicode experience-grounded semantics in Section 8.5.

If we now look at models as the knowledge in Replicoode, or at least as a part of it, then we can say that reasoning takes place when a fact is produced with a specific confidence value where the confidence values are defined by a truth-value function. These values are calculated in the same manner for both forward chaining and backward chaining, or deduction and abduction respectively. But they are interpreted in a different manner.

8.4.2 Truth-value function

Let us now look at how the truth-value function looks like for both forward and backward chaining inference in Replicode. This function defines exactly what the truth value will be from an inference on a model

Definition 34 *Truth-value function for deduction and abduction is defined as:*
$T(lhs, mdl) = fact_{rhs}.conf = fact_{lhs}.conf * mdl.sr,$
where:

- $fact_{rhs}.conf$ *is the confidence value of the produced fact which points either to a goal or a prediction,*

- *$fact_{lhs}.conf$ is the confidence value of the input fact,*
- *mdl denotes the model,*
- *$mdl.sr$ denotes the success rate of the model.*

This function is built into Replicode, and is called any time a forward or backward chaining is performed. It accepts the model and the fact that points to a goal or a prediction and sets the confidence value of the fact which is produced.

8.4.3 Defining a model

Models define a relation between two object patterns which are the left-hand side and the right-hand side of the model. Due to the bi-directional nature of the models, they must be represented with at least two production relations.

Definition 35 *A forward chaining production relation is defined as:*

$lhs \xrightarrow[mdl]{G(fwd)} \mathcal{P}(rhs)$,

where:

- *lhs is a set of objects from $gr_{\underset{mdl}{\longrightarrow}}$ which matches the object structure defined in the left-hand side of the model at the time of reduction,*
- *rhs is a set of objects from $gr_{\underset{mdl}{\longrightarrow}}$ which matches the object structure defined in the right-hand-side of the model at the time of reduction,*
- *$G(fwd)$ is the forward guard which on the set of objects, fwd, which are specified in the model,*
- *$\mathcal{P}(rhs)$ is a shorthand notation for a prediction of the appearance of set of objects, rhs, which match the object structure of the right-hand side of the model: $\{fact_p, pred, fact_o, obj\}$,*
  where:
  - *$fact_p$ is an object of type fact which points to $pred$,*
  - *$pred$ is an object of type prediction which points to $fact$,*
  - *$fact_o$ is an object of type fact which points to obj,*
  - *obj is any object that is expected to appear,*
- *mdl is just a label for the executive object.*

One trick that we use next when we define the backward chaining production relation is to use the the set rhs on the left-hand side, and vice versa. We do this instead of turning the arrow around.

Definition 36 *Backward chaining production relation is defined with the following rule:*
$\mathcal{G}(rhs) \xrightarrow[mdl]{G(bwd)} \mathcal{G}'(lhs)$,
where:

- $G(bwd)$ *is the backward guard which on the set of objects,* bwd*, which are specified in the model,*
- $\mathcal{G}(rhs)$ *is a shorthand notation for the input of the model:* $\{fact_g, goal, fact_o, obj\}$*, where:*
  - $fact_g$ *is an object of type fact and points to* $goal$*,*
  - $goal$ *is an object of type goal which points to* $fact_o$*,*
  - $fact_o$ *is an object of type fact which points to* obj*,*
  - obj *is any object which is wanted to appear,*
- $\mathcal{G}'(lhs)$ *is a shorthand notation for the subgoal which is produced to achieve the object structure which matches* lhs*, where the set is defined as above but with the exception that the* obj *is the object that appeared,*

Now we have defined the production relation for models in Replicode. Integrating reasoning into this production mechanism will prove to be quite natural, as we will see in the next section.

8.4.4 Inference as a production

Inferencing in a production system is a bit different from normal inferencing. Not only does Replicode support non-axiomatic reasoning, but its reasoning is the result of the production of object with specific truth values.

8.4.4.1 Inference rule for forward chaining

In order to attach a truth value to the produced objects from the reduction we just need to add the truth-value function to our forward chaining production relation. By doing so we can formally define what a reasoning production relation for forward-chaining is. We can skip the notation which defines the visibility of objects in memory, as the model only operates within one group. For clarity we also drop the notion of time.

Definition 37 *A forward chaining inference on a model is the production of objects which reflect the object structure in the right-hand-side of the model and contains a fact which points to a prediction which points to a fact which points to a desired object specified in* *the right-hand side of the model and carry a specific truth value.*

$\{fact_{lhs}, ...\} \xrightarrow[mdl]{G(fwd)} \{fact_{rhs}, ...\}$,
where:

- $mdl.sr$ *is the success rate of the model,*
- $fact_{rhs}.conf = fact_{lhs}.conf * mdl.sr$ *is the confidence value of the fact which points to the produced prediction.*

This is essentially just a production relation where there is a specific object structure in the set of inputs for the model and its productions. This also defines exactly what the truth-value in the produced fact from the inference should be; the product of the confidence value of the input fact and the success rate of the model. Now let us define the inference rules for backward chaining.

8.4.4.2 Inference rule for backward chaining

In order to give the inference rules for backward chaining we need to make similar modifications to the backward chaining production relation that we defined earlier:

Definition 38 *A backward chaining inference on a model is the production of objects which reflect the object structure in the right-hand-side of the model and contains a prediction which points to a fact which points to a desired object specified in the right-hand side of the model and carry a specific truth value.*

$\{fact_{lhs}, ...\} \xrightarrow[mdl]{G(bwd)} \{fact_{rhs}, ...\}$,
where:

- $mdl.sr$ *is the success rate of the model,*
- $fact_{rhs}.conf = fact_{lhs}.conf * mdl.sr$ *is the confidence value of the fact which points to the produced goal.*

8.4.5 Learning through inference

Before we can move on to the next section where we give Replicode experience-grounded semantics, we need to define how learning is achieved through inference. Let us look at an informal description of what learning through inference in Replicode is.

When a model produces a prediction which points to a fact as a result of deduction, a success object will be produced after the deadline for that fact has expired which will either point to a fact if the prediction was successful or an anti-fact if the prediction was not successful. If the prediction was successful then the number of positive evidences, and count, will both be incremented by one which increase the success rate of the model. If the prediction was unsuccessful, then the number of positive evidences will remain unchanged, but count will be incremented by one, which will reduce the success rate of the model.

We see that this is actually a two-step process, first the model performs deduction, then the model is updated. This has to be represented as two different rules which are both a composition of production relations, where the rules are chosen based on the result of the prediction.

Definition 39 *Learning through inference is defined as:*

1. $\{fact_{lhs}, ...\} \xrightarrow[mdl]{G(fwd)} \{fact_{rhs}, ...\} \cap \{success_{fact}\} \xrightarrow[mdl]{} \{mdl_S\}$,
2. $\{fact_{lhs}, ...\} \xrightarrow[mdl]{G(fwd)} \{fact_{rhs}, ...\} \cap \{success_{|fact}\} \xrightarrow[mdl]{} \{mdl_f\}$,

where:

- $\xrightarrow[mdl_{s/f}]{}$ *is the reduction which takes place given a success or a failure of the prediction, respectfully.*
- $success_{fact}$ *is notification of a successful prediction,*
- $success_{|fact}$ *is notification of an unsuccessful prediction,*
- mdl_s *is the modified model as the result of a successful prediction where* $mdl_s.sr = \frac{mdl.positive\_evidence+1}{mdl.count+1}$ *is the success rate of the modified model.*
- mdl_f *is the modified model as the result of an unsuccessful prediction where* $mdl_f.sr = \frac{mdl.positive\_evidence}{mdl.count+1}$ *is the success rate of the modified model.*

Though this could be joined with the inference rules defined previously, they are kept separate for simplification. Also note that we do not specify from where the fact or the anti-fact comes from, but that should be easy to do if we define rCode, or the executive as an executive object which produces the success object when the deadline has expired.

Having properly defined the reduction mechanism in Replicode, given the inference rules and defined how learning is achieved through inference, we can now move on to the next section where we will give Replicode experience-grounded semantics.

8.5 Semantics

We have defined the general transition relation for Replicode, the allowed inference rules and inference through learning. Now we can define what knowledge, and experience is in Replicode.

Definition 40 *Knowledge is the set of objects of the type model and the objects of type fact which can be used in the reduction for those models.*

Note that both models and facts carry the truth values we defined earlier.

If we look at *models* with the glasses of *term logic*, we can borrow terminology and concepts from NARS to give Replicode semantics, as a *model* is analogous to the implication relation defined in NAL-7.

Now we can define *experience* in Replicode.

Definition 41 *An experience, $K_{t,Gr}$, is a non-empty set of models, and a possibly empty set of objects in the set of groups Gr, that carry some combination of the truth values which are derived by learning through inference at time t.*

This definition covers essentially two objects in Replicode, a fact that is either used as an input for an executive object, a fact that is produced as the result of a reduction, and the executive object *model*.

Now we can move on to give the *meaning* of valid objects in Replicode.

Definition 42 *Given an experience $K_{t,Gr}$, the meaning of a non-empty set of visible objects, $O_{t,Gr}$ to one or more models, $M_{t,Gr}$, in the set of groups Gr is the extensions and intensions of the models in $K_{t,Gr}$ which is evaluated at time t.*

Having given these definitions, we have given Replicode experience-grounded semantics. The semantics for Replicode has not been fully given, as we have not defined the truth-value functions and other lower levels of the semantics, but we will make this suffice for the purpose of giving Replicode experience-grounded semantics.

8.6 Grammar

There are several ways in which one can look at the syntax for Replicode. One can just describe all rules that are valid with respect to all the predefined objects with an EBNF notation as was attempted in Appendix D for Replicode v1.1a. Another approach would be to give a slightly simplified version of the syntax such that it can be represented in a meaningful way. Third approach could be to describe the syntax with respect to the kind of statements that the inferencing mechanism in Replicode understands.

In this chapter we will give a pseudo-grammar for general statements in Replicode and for the inferencing mechanism. This is merely intended for illustration purposes.

8.6.1 High-level grammar

If we try to describe the grammar of Replicode with respect to its functionality in the language, the task becomes a lot more simpler. By removing all the constraints that predefined objects place on the grammar, we can give a conceptual representation of the grammar in a simplified, pseudo EBNF, manner.

What this grammar reveals is that the syntax of Replicode can roughly be split into four, possibly conjunct, categories of sentences. By doing so we introduce immense ambiguity and erroneous rules which can generate illegal statements in Replicode.

But what also happens when we do so is that we see that, at the core, Replicode has a really simple syntax which allows us to declare classes, instantiate objects and provides a connection to C++ code for extension.

```
Sentence ::= ClassDecl | ObjectDecl | Definition | MapToRCode/RAtom
ClassDecl ::= "!class" " " Object
ObjectDecl ::= (Label ":")? Object " " SetOfViews
Object ::= "(" className (" " Parameter)+ ")"
Definition ::= "!def" " " name name
MapToRCode/RAtom = ("!class" | "!def" | ...) " " (name | Object)
Parameter ::= (atom | memberDecl | Object)
```

A valid sentence in Replicode is roughly:

1. **ClassDecl:** A class declaration is usually on the form: **!class** (`name val_name`**:**`type`).
2. **ObjectDecl:** This rule is context dependent, the label can either be a pointer to an object or it can be a variable. It is a pointer when it is declared in a group but it is a variable when it is used in a guard. An example of the former could be `str1`**:**(**str** `"Hello"`) **|[ ]**, and for the latter we might have `sum`**:**(+ `1 1`).
3. **Object:** This is also context dependent rule, but the difference here is that the `Parameter` can either be a member declaration when it appears in a class declaration, or it can appear as a label or a variable.
4. **Definition:** Definitions are simple and on the form **!def** `name value`.
5. **MapToRCode/RAtom:** These are classes, operators, devices, etc, that are also internal Replicode structures.
6. **Parameter:** The context dependency here is the same as for `Object`.
7. **atom/memberDecl/className/name:** Atoms and member declarations are constructs from Replicode, and className and name are simple labels.

This grammar is definitely not suitable for parsers, but it does explain in a neat manner how a valid statement in Replicode generally looks like.

8.6.2 Grammar for the inferencing mechanism

Another natural way to give grammar would be to describe what kind of sentences the inferencing mechanism understands. In the current version of Replicode, version 1.2, the only relation upon which inferencing can be made, are bi-directional *models*. These models contain structured information which are pattern-matched against when inferencing.

Given that, we can give another pseudo EBNF for Replicode which describes grammar for the inferencing mechanism.

```
Sentence ::= Model
Model ::= LeftHandSide -> Object
LeftHandSide ::= (Command | Object)
Command ::=  (Label ":")? "(cmd" name " " "[" (Parameter)+ "])"
Object ::= "(" className (" " Parameter)+ ")"
Parameter ::= (atom | memberDecl | Object)
```

Technically, a command is an object also, but only in this special case above, $className \neq$ "*cmd*". Also, this grammar totally disrespects the syntax of Replicode, but it illustrates that a model has a left and right-hand side which have only one object each. That is, models essentially build one-to-one models of the world, but they can also accept multiple objects through instantiated composite states.

A sentence that the inferencing mechanism understands is roughly:

1. **Model:** When models are successively pattern-matched against, they are instantiated and a reduction is performed. The truth value that is produced can either be encoded within a *prediction* or a *goal*.
2. **LeftHandSide -> Object:** This is mainly used for restricting the commands to the left hand side of the model.

Models in Replicode also encode a special context dependent Replation between A and B. It is not only a higher-level syntax, which allows composite sentences, but its evaluation is also dependent on the values that have been placed as a result of a pattern-matching from current knowledge.

This does not complete a full grammatical description of Replicode, but it does give us an idea about statements which the inferencing mechanism understands.

8.7 Knowledge as Causal models

Causal models, as represented in Pearl [2009], are essentially directed graphs which represent a causal relation which can have a given Bayesian probability. In short, a causal link in a causal diagram represents a pair of facts, A and B, where B must follow from A, and if B would not have been, if it were not for A. That is, causality is defined in the context of counterfactuals. This idea of defining causation in terms of counterfactuals is not new, and was used by David Hume and David Lewis in their attempt to define causation [Menzies, 2009]. In recent times, *causality* has bee fully mathematized by using a calculus of counterfactuals [Pearl, 2009, p. xv].

Causal models assume that the probability distribution over the model is known, they provide algorithms which can be used to reason about a given causal model both in terms of actions on the causal model via the *do* operator, or to infer a likely result, and they allow us to do statistical inferences on the model. What this means is that we can answer questions about a given model with some degree of certainty, which is essentially *null hypothesis testing*.

These models are axiomatic, where causal relations encode truth with a given probability distribution. Causal models do not evolve in time, the probabilities are actual probabilities and not measurements, such that they give an actual representation of a given causal model. This might at first glance seem a bit restrictive compared to what we have been discussing in the latter part of the thesis, but we can argue that knowledge, as it is represented in NARS and Replicode at a given point in time, is in essence a special case of causal models.

Though models in Replicode have no notion of probability, we saw in our discussion on *confidence* values in Replicode that they are *similar* to probabilities. If we have a model that says that when you toss a coin, the result will be that the head will come up, the confidence of the prediction that is made when the coin is tossed will eventually converge to 0.5, the expected value from a coin toss, with large number of trials given that we have a fair coin. We can then argue that given a *fair sample* and a stable environment, the confidence value of a prediction will converge to the actual probability of the outcome with increasing number of trials. Saliency then acts *like* a p-value of some sort in hypothesis testing, that is, we place a lower bound on the confidence value in order to accept the hypothesis that, e.g., that when you toss a coin, the head will come up. I'm probably on slippery slope with this analogy, but it illustrates the point that high-level knowledge, or structured knowledge, in Replicode can bee seen as a special case of a Bayesian network.

The conclusion being that knowledge in Replicode at a certain point in time can essentially be treated as causal models, which puts us on a firm theoretical ground of causation. This could in theory be added to Replicode as an extension.

8.8 Summary

Formalizing Replicode in terms of a production system does seem to provide an accurate description of the internal processes of Replicode. The production relation that we gave might not be entirely accurate, as we assume instantaneous productions, and do not make a distinction between template and input patterns, but the definition does give room to expand it to account for real-time in productions and a more detailed handling of input patterns if needed. The fact that we can describe reasoning in terms of a production shows that our approach fits naturally to Replicode.

Concerning the grammar and the semantics of Replicode, there is still some way to go, as we gave only a partial formalization of those.

Chapter 9

Replicome v.2.0

A non-axiomatic implementation

As we have seen in the preceding chapters, non-axiomatic systems differ from traditional axiomatic systems in many aspects. This leads us to consider how a non-axiomatic version of *Replicome* would look like.

In this chapter, we will describe *Replicome* v.2.0, which draws inspiration from the systems that we have covered so far; ResourceHome, Replicode and NARS, along with a subject which has yet to be covered, a theory of *causality*[Pearl, 2009]. We will not give a complete description for this system, as designing such a system would be a master's project in and of itself. Rather we will give an insight into the design principles behind such a system, give a high-level description the architecture and some of its internal processes, along with ideas of implementation.

9.1 Overview

Replicome v.2.0 is a system that provides intelligent AAL. It should be capable of learning some of the behaviors of the inhabitants with the end goal being to increase the standard of living. If we go back and review some of the flaws we listed for *Replicome* v.1.0, we want the system to be able to learn to alarm only when the inhabitant is forgetting his keys and is actually leaving his home, instead of sounding the alarm when the inhabitant is at the door [1].

Replicome v.2.0 is in *intelligent* system in the sense that it adapts under insufficient knowledge and resources. It is capable of learning how itself works through observing the effects of that

[1] For demonstration purposes, we discard simple solutions to this problem like placing an extra sensor outside of the door which would notify only when the inhabitant is outside the door, as we would want to notify of a forgetfulness situation with some notice.

it can have on its environment and it can create models of the world by building cause-effect relationships of the input-stream it receives.

Finally, *Replicome* v.2.0 should be *safe*, meaning that when the system is deployed, it should not experiment with its devices in such a way that it can be potentially dangerous for the inhabitants.

We essentially want the system to manifest *embodied intelligence*, but not in the traditional sense that the system has a human like body, rather we look at the sensors and actuators like an extension of the brain which the software represents. This means that the system needs to be able to get feedback from its operations.

9.2 Architecture

Replicome v.2.0 is a composition of sensors, actuators and software to control them. Sensors provide information about the environment, actuators provide the system with ways to interact with the environment, and the software is the brain that makes sense of the input from the sensors, and uses the actuators to achieve some goals.

9.2.1 Sensors and actuators

The hardware layer of the system must be constructed in such a way that for every operation that the system can make, there has to be provided a feedback from its operations. If we would for example have actuators that can control the temperature of the habitat, there must be a thermometer for the area which is affected by that actuator. There are really no limitation on what kind of hardware can be connected to the system, as long as the system is provided an appropriate feedback for the operations on that hardware.

9.2.2 Software layer

The software layer will be composed of a domain knowledge, drivers for sensors and actuators, and a control mechanism. The control mechanism must contain safety measures to make sure that no harmful actions will be taken and that the system does not pursue goals that eliminate the possibility of achieving more urgent goals.

These goals can be like to make sure that the temperature does not fluctuate beyond a certain margin of error, or to make sure that no forgetfulness situations occur.

9.2.2.1 Forgetfulness situation

Let us take a quick look at an example of how to represent a forgetfulness situation in a non-axiomatic manner. Here we use a composite state to indicate that the subject is at the exit but does not have the keys. If we have a forgetfulness situation then that would indicate that the next time that we sample the environment that the subject has left the habitat.

We represent that with the model, which states that if there is a forgetfulness situation then that would imply the absence of the subject the next time we sample the environment.

Code Listing 9.1: Declaration of the dangerous_interaction program.

```
diff_pos:(cst |[] []
   (fact (mk.val subject position exit :) ::); subject at exit
   (|fact (mk.val keys position exit :) ::); keys not at exit
|[]
|[]
[stdin] 1) |[]

forgetfulness:(mdl |[] []
   (fact (icst diff_pos |[] [h: p0:] ::) t0: t1: ::)
   (|fact (mk.val subject position :) t0: t2: : 1)
[]
   t2:(+ t0 sampling_period)
[]
   t0:(- t2 sampling_period)
[stdin] 1 1 1 0 1) |[]
```

What would happen if the subject would not leave the habitat is that the success rate of the model would be reduced every time an unsuccessful prediction about the absence of the subject would be made. If the model makes repeated unsuccessful predictions, eventually the confidence value would be so low that the model would not be considered to be an accurate description of the world.

What we would need is a model acquisition module which would try to find a better model of a forgetfulness situation. We would also need a program that would signal that a forgetfulness situation had occurred. And we would need to have a top-level goal which states that the system should avoid forgetfulness situations.

9.2.2.2 Knowledge acquisition

There are many ways to build knowledge, or models, about the world. We would propose to try to use the IC or IC* algorithms proposed in [Pearl, 2009, p. 50 and 52]. Though *causal models* and models in Replicode are not identical, we showed in last chapter that the latter is essentially a special case of the former.

We will leave it as a project for the future to find a reasonable integration between the two, and a method of implementation.

9.3 Summary

The discussions in this chapter are lightweight, relatively non-specific and very general. We do not dive too deep into the design and implementation of the non-axiomatic version of *Replicome* as such a system needs components which are outside the scope of this thesis. What this chapter does show is that the functionality of the system is not specified by a formal model, but rather the functionality is determined by the experience that the systems has.

Chapter 10 Conclusions

There are several conclusions which have been reached in this thesis.

First we saw that there exists an algorithmic translation from FOL to Replicode. By using *ResourceHome* as a case study and translating that into Replicode, we saw that we can actually translate all first-order logic expressions into Replicode through the logically equivalent intermediary format of conjunctive normal form. The reason why we can not give a direct translation from first-order logic into Replicode is because not all inferences are available in Replicode which are available in first-order logic, such as double negation elimination. This translation is however not without its limitations as we saw in Chapter 4. First-order logic expressions that contain quantifiers lose qualitative relations between atoms which means that the translation does not preserve all of their semantics. We do not provide proof of the soundness nor the completeness of the translation, but we showed that the translation is equisatisfiable, which means that we are able to represent systems that are specified in first-order logic, such as *ResourceHome*, into Replicode.

Then we saw that NARS and Replicode share a common theoretical background and the reasoning framework within Replicode is essentially a small subset of NAL. This is important for two reasons; first for our attempt at applying experience-grounded semantics to Replicode, and secondly for future attempts at extending the reasoning framework in Replicode to allow for inferences such as those found in NARS. What this also suggests is that Replicode can facilitate the architecture needed to build intelligent systems as described by NARS.

Then we saw that NARS and Replicode try to solve two different problems within AI; formalization of categorical reasoning and a platform to facilitate architecture based on Constructivist AI assumptions. This means that the two systems complement each-other in such a way that inferences available in NARS are a viable option for future extensions of Replicode.

We saw that Replicode can indeed be formalized in at least two different aspects. First we argued that we can give a formal description to a *reduction* in Replicode, we did so by defining a general production relation along with defining the necessary components of that definition. We showed that the production relation can describe the operations of programs and models, where programs can be described by the general production relation and models can be described by two rules; forward chaining production relation and backward chaining production relation. We also showed the composition of two production relations as we showed how to described learning through inference. Secondly we argued for Replicode being able to have an experience-grounded semantics. We did so by first defining truth values in Replicode as the confidence and the success rate which objects hold. Then we defined knowledge as the objects that hold the truth values. Next we defined experience as the knowledge that have varying truth values as learning takes place through inference. Then finally we defined the meaning of statements within Replicode as the intensions and extensions of the models in Replicode as a given time.

Then we gave two informal grammar for Replicode, one general for legal statements in Replicode, and for the inferencing mechanism. This grammar can not be used to build parsers for Replicode, but it does serve the purpose of helping people getting a better sense of how legal statements in Replicode look like. The grammar also reveals that the underlying format of legal statements in Replicode are well structured, which is one of the main purposes of building Replicode; to simply syntax to allow for relatively easy self-inspection and modification code.

Another more subtle conclusion that we have reached, and that when building *intelligent* systems, non-axiomatic reasoning system heavily outweigh the traditional axiomatic reasoning systems. According to our working definition of intelligence, non-axiomatic reasoning systems are more flexible and can adapt both to insufficient knowledge and resources. This is crucial because it is hard, if not impossible, to have complete knowledge of a given domain. And even if we do have complete knowledge of the domain, we more often than not have to search though state spaces that grow exponentially as the state-space explodes. This well known problem hinders the real-time operation of systems, as we either exhaust memory before a solution is reached or we have to wait for an unacceptably long period of time as the CPU crunches its way to a solution. The way that non-axiomatic reasoning systems handle these problems is first by providing the system with the means to acquire new knowledge, modify existing knowledge and throw out bad knowledge. Secondly, non-axiomatic reasoning systems essentially bypass the state space explosion problem by selecting the best solution they can reach within a given amount of time by evaluating the confidence value of the solution.

There is however a trade-off cost associated with building non-axiomatic systems, and that is that they are way more complex in design and implementation. These systems are essentially composed of a multitude of small rules, from which an intelligent behavior emerges, but not an omniscience component which can answer everything. These systems are relatively new and the surface of their capabilities has barely been scratched.

Chapter 11

Future Work

There is still some future work to be done, both in terms of intelligent systems that provide AAL, and the formalization of Replicode. Here we outline a few of the possible future work that can be done.

1. **Design and implement *Replicome* v.2.0**
   Designing and implementing this system is a major challenge in and of itself.
2. **Find the limitations of the two-step translation algorithm**
   We did not fully explore soundness and the completeness of the translation algorithm we presented.
3. **Give a formal description of Replicode as a production system**
   There is still a lot of work that needs to be done in formalizing Replicode, some ideas of how to formalize Replicode as a production system can be found in the chapter on related work.
4. **Give a formal description of Replicode as a rewrite system**
   Replicode can also be formalized as a rewrite system. Maude [Maude, 2012] is based on rewrite logic and some inspirations can be found in the work that has been done on real-time Maude, along with some material in the chapter on related work.
5. **Extend the product relation with real-time**
   We assumed instantaneous productions for simplifications, but if one is to get a more accurate description of what is happening within Replicode, real-time must be added.
6. **Group interactions**
   Groups and their behaviors and interactions was left relatively untouched in this thesis. Their usability is probably not fully understood yet.
7. **Merge NARS and Replicode**
   Given how NARS and Replicode complement one another, it would make sense to implement NARS in Replicode. The most natural way of doing so would be to create new predefined objects like programs and models that allow for the same inference as is defined in NAL.

Chapter 12

Related Work

12.1 Ambient Assisted Living

There is a lot of work being done on systems that provide ambient assisted living, and we will not make any efforts to cover them all here. In the following subsections we will give a light overview of some of the work being done in the field.

12.1.1 HOPE

Smart Home for Elderly People (HOPE) [Project, 2012a] is aimed at helping people with Alzheimer's disease to use technology to allow for more independent life. The main aim of the project is to help people with Alzheimer's disease to help them perform the activities of daily life such that they can become more independent

An example of what HOPE hopes to bring patients is monitoring of appliances, such that they can be turned off if they are forgotten in an activated state, monitor the patient such that he doesn't leave the house, and talk to the patient is he is about to get a panic attack.

12.1.2 ROSETTA

ROSETTA [Project, 2012b] is aimed at prevention, early detection and efficient management of treatable psychosocial and physical consequences of chronic diseases that are the result of progressive cognitive decline.

The ROSETTA system can monitor the activities of elderly people with sensors and generate alarms when unexpected events occur. These events can include events such that a person falls, or if a person is immobile for extended periods of time which might indicate serious problems. The ROSETTA system is not entirely autonomous, as it relies on the input from caretakers, which helps to ensure unobtrusiveness of the system.

12.2 Smart Homes

Smart homes have been on the radar for a long time and is their realization somewhat linked to the realization of truly intelligent systems. Thought the latter has yet to be realized, some attempts have been made at the former. There are countless of examples of systems that aim to make homes smart, and will we provide an quick overview of some notable smart homes which are relevant to the theme of this thesis.

12.2.1 Axiomatic smart home

Policy-based Managing of Actors in Self-Adaptive Systems (PobSAM) [Khakpour et al., 2010] is a formal methodology designed to enable modeling and development of self-adaptive systems.

In the paper cited above, an example is given of a smart home which allows for automatic lighting control, automatic window control which includes adjusting blinds for the windows, and automatic heating control which allows for constant room temperature and energy saving.

As the name suggests, the PobSAM is an actor-based model which allows for limited predetermined adaptation. Though self-adaptive, the system does adapts only to environmental variables, but not insufficient knowledge and resources and is therefore not intelligent as we have defined.

12.2.2 Semi-axiomatic smart home

In the paper *Management of uncertainty and spatio-temporal aspects for monitoring and diagnosis in a Smart Home* [Augusto et al., 2005] was outline a smart home which is non-axiomatic in nature. It uses methodology called *Rule-base Inference Methodology using the Evidential Reasoning* (RIMER) which is extended with active database framework.

This system combines event-condition-action rule systems with uncertainty reasoning to estimate the likelihood of a given rule being the best explanation for a given event. It is semi-axiomatic in the sense that it assigns a confidence value to a given statement based on the available evidence. So the system does to a certain extent assume insufficient knowledge, but no efforts are made to control insufficient resources. That, along with the fact that the truth value of statements do not evolve over time, but are decided on a case-by-case basis makes the system semi-axiomatic rather than non-axiomatic.

12.2.3 Non-axiomatic smart home

No examples were found for non-axiomatic smart homes, so this one is up for grabs!

12.3 Formalization of production rule systems

There have been some attempts at formalizing production rule systems and will we provide an overview of some of them here.

12.3.1 Rewrite Semantics for Production Rule Systems

In the paper *Rewrite Semantics for Production Rule Systems: Theory and Applications* [Snyder and Schmolze, 1996] the authors formalized the the actions of production rules on a working memory of facts as a rewriting process which are subject to constraints. They described the system as a composition of three components: a memory of facts, a collection of production rules and an interpreter which applies the rules.

The production rules which they dealt with are on the form:

$C_1, ..., C_n \Rightarrow A_1, ..., A_m$, where:

1. each side of the rule is a set,
2. the conditions C_i are either atoms or negative literals,
3. each of the A_i is of the form "ADD A" or "REMOVE A",
4. each variable that occurs in an action A_i also occurs in some negated C_j.

This formalization of production rules is somewhat different from our formalization of the general production relation. Firstly they assume one global workspace for all production rules so they need not to consider rule scoping as we needed to consider when we formalized the production relation for Replicode. Secondly, these production rules have no sense of time so they are not applicable to the functioning of Replicode. Thirdly, the left-hand side of production rules can not be used as an input as can be done in Replicode with the template and input patterns. Finally, the formalization presented in that paper was for axiomatic systems based on first-order logic (with negation as failure) and would not have allowed us to represent non-axiomatic reasoning nor learning by inference.

This work can however provide some guidelines for future work, then as to formalize Replicode as a rewrite system.

12.3.2 Production Systems and Rete Algorithm Formalisation

In the paper *Production Systems and Rete Algorithm Formalisation* [Horatiu Cirstea, 2004] the authors presented a formalization of general production systems and Rete algorithm.

In that paper, a general production system was defined as follows:
$\mathcal{GPS} = (\mathcal{F}, \mathcal{P}, \mathcal{X}, \mathcal{L}, \mathcal{WM}, \mathcal{R}, \mathcal{S}, \mathcal{T})$, where

1. $\mathcal{F}$ is the set of function symbols,
2. $\mathcal{P}$ is the set of predicate symbols,
3. $\mathcal{X}$ is the set of variables,
4. $\mathcal{L}$ is the set of labels,
5. $\mathcal{WM}$ is the initial working memory,
6. $\mathcal{R}$ is the set of production rules over $\mathcal{F} = (\mathcal{F}, \mathcal{P}, \mathcal{X}, \mathcal{L})$,
7. $\mathcal{S}$ is the resolution strategy,
8. $\mathcal{T}$ is the matching theory.

They also define production rules as:
$[l]\ if\ p, c\ remove\ r\ add\ a$, which consist of the following components:

1. a name from the label set $l \in \mathcal{L}$,
2. a set of positive and negative patterns $p = p^{+} \cup p^{-}$ where a pattern is a term $p_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
3. a proposition c whose set of free variables is a subset of the pattern variables: $Var(c) \subseteq Var(p)$,
4. a set of r of terms whose instances could be intuitively considered as intended to be removed from the working memory when the rule is fired,
5. a set of terms whose instances could be intuitively considered as added from the working memory when the rule is fired.

Their definition of general production system might be applicable to Replicode with some modifications. First a modification needs to be made to allow for non-axiomatic logic, but they assume first-order logic (with negation as failure). They also assume one global workspace so one would need to factor in the scoping which takes place in groups in Replicode. One would also need to add the notion of time, as the definition of the production system has no sense of time.

The definition of the production rule is also not rich enough to apply them to executive objects in Replicode. As with the definition of the production system, no special scoping is assumed as we saw with groups in Replicode, the rules have no sense of time and they do not allow for non-axiomatic reasoning. They seem to formalize the production system as a rewrite system as was done in [Snyder and Schmolze, 1996]. We made no attempts at formalizing Replicode as a rewrite system, as we were mainly focused on formalizing the reduction mechanism and giving Replicode experience-grounded semantics.

12.4 Artificial general intelligence

The research on AI that has been cited in this thesis, i.e. NARS and HUMANOBS, fall under the category of artificial general intelligence (AGI), or strong AI [1]. There are other research in this field which are worth mentioning.

12.4.1 Cyc

Cyc [Cycorp, 2012a] takes the taxonomic approach to try to build common sense knowledge which is capable of performing inference on that knowledge. The project was started in 1984 and is still running. An open source version of the knowledge base has been released under the name OpenCyc [Cycorp, 2012b].

12.4.2 OpenCog

OpenCog [Foundation, 2012a] is an open source cognitive framework based on OpenCog Prime [Foundation, 2012b] which aims at builing AGI. This framework opens up the possibility of integrating various aspects of AI methodologies. There are essentially four key aspects of AGI design according to Hart and Goertzel [2008] which are: cognitive architecture, knowledge representation, learning and teaching methodologies. The OpenCog framework aims to help with all of these categories.

12.4.3 Numenta

Numenta takes the biological approach to AGI, where the aim is to build intelligent systems based on the way that the neocortex operates [Numenta, 2012]. The work done at Numenta Inc. is mostly centered around hierarchical temporal memory [Jeff Hawkins, 2012] which is a machine learning model which as shown interesting results in image recognition.

[1] Note that AGI hasn't been realized yet, but these projects aim at creating general intelligence, as apposed to restricted, or narrow, intelligence as seen in expert systems and alike.

Appendix A

First-order logic helpers

To be able to properly convert FOL sentences into Replicode, we need to define some constructs and methods that will be used throughout this chapter.

A.1 The rFOL class

The rFOL class is a simple class that enables us to declare something to be true in a knowledge base.

The declaration of the class is as follows.

!class (rfol v **: bl)**

As we saw in Chapter 4, we do not actually need to specify the boolean value property v in the class because there is no way of negating formulas in Replicode such that the negation is carried into the formula as in the traditional logical sense [1]. It is there simply to comply with the structure of marker values and to keep a similarity to regular FOL terms.

A.2 The Result class

The result class is a simple class that carries an identification number to signal that a program has executed. This class is necessary when dealing with evaluations on nested FOL formulae, as it allows us to build more complex expressions than is possible without it. Below is a declaration of the result class.

[1] This is not absolutely true because one can create programs that simulate negation, that approach however does not guarantee that FOL system would be evaluated correctly because other programs are executed simultaneously. So one would need to temporally order the evaluations of the formulas.

!class (result `id`**:nb)**

This class has only one property, an identification number, which is produced as the result of evaluating a simple expression and used as an input of a more complex evaluation.

A.3 The Answer class

The answer class another simple class which is used when working with FOL in Replicode. This class is used to notify if, and what, formula evaluates to true, and is declared as so:

!class (answer `msg`**:st)**

A.4 Program weaving

Program weaving is a technique designed to enable evaluation of a disjunct literals and nested functions. In theory, nesting of programs can be achieved by placing programs within other programs, i.e., via dynamic code production, this technique will however not be covered in this chapter.

In short, when weaving programs, one creates a program for the innermost term in a nested literal with the condition for the literal being evaluated to true in its input, and outputs a result class with a unique identification number. Then the identification number that was produced is used as an input of a new program that is created for the surrounding literal. This can then be applied recursively to weave together complex formulas.

If we look at the literal $letter(p)$ which states that p is a letter, given that p is true. This would be translated first to a program that outputs an instance of the result class with the identification number *uid*, (result *uid*). Then there would be created a program that executes only if an instance of the result class with the identification number *uid* is noticed. The output of the latter program depends on the context. If we pose it as a query on the current knowledge base, we would want to output the answer class to separate it from other outputs, or we would want to weave it into a more complex formula.

A better example of program weaving is illustrated in Section 4.1.7.2 which covers disjunction.

Appendix B

Executables

The Replicode version which was used in this thesis can be fetched, along with all executables in this appendix at the following repository: http://cadia.ru.is/svn/repos/ReplicodeOther/Replicome/. The Replicode files are placed in the MScFiles folder.

B.1 Getting started

B.1.1 Setting up the development environment

These are the steps that need to be followed in order to get Replicode working[1].

Create a folder called ReplicomeOther on your C drive (this will make sure all paths are correct). Check out the Replicome folder from http://cadia.ru.is/svn/repos/ReplicodeOther/Replicome/ into the ReplicomeOther folder (That will give you the folder structure C:/ReplicodeOther/Replicome/).

In order to compile Replicode you will need:

1. A computer with at least 2 fast Cores (>2.4 GHz) and 3GB of RAM, running Win7 or Vista.Failure to meet these minimal requirements will end up in trouble. In particular, no virtualization software will do any good.
2. Visual Studio C++ 2008 Professional + SP1 + Feature Pack, or Visual Studio Express 2008 VS 2008 Express edition is available here. N.B.: Replicode and dependencies are 32 bits executables.
3. Compile the CoreLibrary.

[1] These directions were originally written by Eric Nivel and partly modified by Ólafur Hlynsson.

4. Compile Replicode. Notice that Replicode needs the CoreLibrary dll. You'll have to create a symlink in the working directory (e.g. Replicode/Debug or Replicode/Release) pointing to the CoreLibrary dll.

For editing Replicode source code, you may want to use notepad++: http://notepad-plus-plus.org/ along with a syntax highlighting profile: http://olafur.hlynsson.is/replicode.def.xml.

To run the executive, just run the Replicode/Test sub-project. Make sure you have the values in properties/debugging set to:

Command: $(TargetPath)
Command arguments: ..\Test\settings.xml
Working directory $(OutDir)

B.1.2 settings.xml

Code Listing B.1: The settings.xml file.

```
<TestConfiguration>
  <Load
    usr_operator_path = "./usr_operators.dll"
    usr_class_path = ↙
   "C:/Replicode/Test/V1.2/olafur/user.classes.replicode"
    source_file_name = ↙
   "C:/Replicode/Test/V1.2/olafur/dangerous.replicode"
  />
  <Init
    base_period = "50000"
    reduction_core_count = "3"
    time_core_count = "1"
  />
  <System
    mdl_inertia_sr_thr = "0.9"
    mdl_inertia_cnt_thr = "10"
    tpx_dsr_thr = "0.1"
    min_sim_time_horizon = "25000"
    max_sim_time_horizon = "100000"
    sim_time_horizon = "0.3"
    tpx_time_horizon = "500000"
    perf_sampling_period = "2500000"
    float_tolerance = "0.1"
    time_tolerance = "10000"
  />
  <Debug debug = "yes">
    <Resilience
      ntf_mk_resilience = "2000"
      goal_pred_success_resilience = "1000"
    />
  </Debug>
  <Run
    run_time = "1000"
    probe_level = "2"
    decompile_image = "yes"
    ignore_ontology = "yes"
    write_image = "no"
    image_path = "../Test/test.replicode.image"
    test_image = "no"
  />
</TestConfiguration>

<!--Usage
Init
  base_period: in us.
  reduction_core_count: number of threads processing ↙
   reduction jobs.
  time_core_count: number of thread processing update jobs.
System
  mdl_inertia_sr_thr: in [0,1].
  mdl_inertia_cnt_thr: in instance count.
  tpx_dsr_thr: in [0,1].
  min_sim_time_horizon: in us.
  max_sim_time_horizon: in us.
  sim_time_horizon: in [0,1]: percentage of (before-now) ↙
   allocated to simulation.
  tpx_time_horizon: in us.
  perf_sampling_prd: in us.
  float_tolerance: in [0,1].
  time_tolerance: in us.
Debug
  ntf_mk_resilience: in upr (i.e. relative to the ntf group).
  goal_pred_success_resilience: in upr (i.e. relative to the ↙
   ntf group).
Run
  run_time: in ms.
  probe_level: any probe set to a level > =  this level will ↙
   not be executed. 0 means no probe will be executed.
  decompile_image: yes or no.
  write_image: yes or no.
  test_image: yes or no (yes: reads back and decompiles).
-->
```

B.1.3 user.classes.replicode

The user.classes.replicode file contains custom classes and definitions that are application/domain specific. The executable s in this chapter assumes these definitions to be in place in order to execute. Note that the !load path needs to be changed.

Code Listing B.2: The user.classes.replicode file.

```
!load ↙
   C:\ReplicodeOther\Replicome\Replicode\Test\MScFiles\std.replicode

; utilities.

!def (std_grp _upr _sln_thr _act_thr _vis_thr _ntf_grps) (grp ↙
   _upr _sln_thr _act_thr _vis_thr 1 0 1 0 0 1 0 0 1 1 1 1 0 ↙
   0 0 0 0 0 1 0 1 1 0 1 0 0 _ntf_grps 1); c-salient and ↙
   c-active.

; domain-dependent classes.
!class (vec3 x:nb y:nb z:nb)
!class (speak word:st)
; Replicome specific
!class (vec2 x:nb y:nb)
!class (str (_obj s:st))
!class (rfol v:bl)
!class (interaction type:st role:st propval:nb)
!class (answer msg:st)


; device functions.
```

```
!dfn (alarm :); arg0: type of danger, arg1: position of danger

; Constants
!def dgr_thr 100
!def div_factor 0.9
!def exit (vec2 0 3)

; initial groups.

root:(std_grp 0 0 0 0 [nil]) [[SYNC_FRONT now 0 forever nil nil ↙
    COV_OFF 0]]
stdout:(std_grp 0 0 0 0 |[]) [[SYNC_FRONT now 0 forever root nil ↙
    COV_OFF 0]]
stdin:(std_grp 0 0 0 0 [stdout]) [[SYNC_FRONT now 0 forever root nil ↙
    COV_OFF 0]]

primary:(std_grp 2 0 0 0 |[]) [[SYNC_FRONT now 0 forever root nil ↙
    COV_OFF 0]]; ensure upr>0.
secondary:(std_grp 2 0 0 0 |[]) [[SYNC_FRONT now 0 forever root nil ↙
    COV_OFF 0]]; ensure upr>0.
grp_pair:(mk.grp_pair primary secondary 1) [[SYNC_FRONT now 0 ↙
    forever root nil]]
drives:(std_grp 0 0 0 0 [stdout]) [[SYNC_FRONT now 0 forever root nil ↙
    COV_OFF 0]]; holds pgm that inject drives periodically; ↙
    drive priority=pgm.act; be sure that the pgm tsc is larger ↙
    than the primary's upr.

; application ontology.
position:(ont 1) [[SYNC_FRONT now 0 forever root nil]]; usage: (↙
    mk.val x position (vec3 0 0 0) 1)
danger:(ont 1) [[SYNC_FRONT now 0 forever root nil]]

; values.
self:(ent 1) [[SYNC_FRONT now 0 forever root nil]]

; Replicome specific
reagent:(ent 1) [[SYNC_FRONT now 0 forever root nil]]
reactant:(ent 1) [[SYNC_FRONT now 0 forever root nil]]
subject:(ent 1) [[SYNC_FRONT now 0 forever root nil]]
keys:(ent 1) [[SYNC_FRONT now 0 forever root nil]]
```

```
run:(ont 1) [[SYNC_FRONT now 0 forever root nil]]
is:(ont 1) [[SYNC_FRONT now 0 forever root nil]]

p:(ent 1) [[SYNC_FRONT now 0 forever root nil]]
q:(ent 1) [[SYNC_FRONT now 0 forever root nil]]
r:(ent 1) [[SYNC_FRONT now 0 forever root nil]]
s:(ent 1) [[SYNC_FRONT now 0 forever root nil]]
```

B.1.4 std.replicode

The std.replicode file which is loaded from user.classes.replicode contains standard definitions for Replicode. The declarations in this file contains the connection between Replicode structures to macros that are written in C++.

This file contains the structures needed for the examples in this chapter to execute.

Code Listing B.3: The std.replicode file.

```
; std.replicode

; utilities.
!class (_obj :~ psln_thr:nb)
!class (_grp (_obj {upr:nb sln_thr:nb act_thr:nb vis_thr:nb ↙
   c_sln:nb c_sln_thr:nb c_act:nb c_act_thr:nb dcy_per:nb ↙
   dcy_tgt:nb dcy_prd:nb dcy_auto:nb sln_chg_thr:nb ↙
   sln_chg_prd:nb act_chg_thr:nb act_chg_prd:nb avg_sln:nb ↙
   high_sln:nb low_sln:nb avg_act:nb high_act:nb low_act:nb ↙
   high_sln_thr:nb low_sln_thr:nb sln_ntf_prd:nb ↙
   high_act_thr:nb low_act_thr:nb act_ntf_prd:nb ntf_new:nb ↙
   low_res_thr:nb ntf_grps:[::grp] :~}))
!class (_fact (_obj {obj: after:us before:us cfd:nb :~}))
!class (val (_obj val:))
!class (val_hld val:)

; mapping low-level objects to r-code.
!class (ent (_obj nil))
!class (ont (_obj nil))
!class (dev (_obj nil))
!class (nod (_obj id:nid))
!class (view[] sync:bl ijt:us sln:nb res:nb grp:grp org:)
```

```
!class (grp_view[] sync:bl ijt:us sln:nb res:nb grp:grp org: ↙
    cov:bl vis:nb)
!class (pgm_view[] sync:bl ijt:us sln:nb res:nb grp:grp org: act:nb)
!class (ptn skel:xpr guards:[::xpr])
!class (|ptn skel:xpr guards:[::xpr])
!class (pgm (_obj {tpl:[::ptn] inputs:[] guards:[::xpr] prods:}))
!class (|pgm (_obj {tpl:[::ptn] inputs:[] guards:[::xpr] prods:}))
!class (grp (_grp nil))
!class (icmd function:fid args:[])
!class (cmd (_obj {function:fid args:[]}))
!class (cmd_arg (_obj {function:fid arg:ont}))
!class (ipgm (_obj {code: args:[] run:bl tsc:us nfr:bl}))
!class (icpp_pgm (_obj {code:st args:[] run:bl tsc:us nfr:bl}))

; mapping low-level markers to r-code.
!class (mk.rdx (_obj {code: inputs:[] prods:[]}))
!class (mk.low_sln (_obj obj:))
!class (mk.high_sln (_obj obj:))
!class (mk.low_act (_obj obj:))
!class (mk.high_act (_obj obj:))
!class (mk.low_res (_obj obj:))
!class (mk.sln_chg (_obj {obj: chg:nb}))
!class (mk.act_chg (_obj {obj: chg:nb}))
!class (mk.new (_obj obj:))

; mapping high-level objects to r-code.
!class (mk.val (_obj {obj: attr:ont val:}))
!class (mk.act (_obj {actr:ent cmd:cmd}))
!class (fact (_fact nil))
!class (|fact (_fact nil))
!class (cst (_obj {tpl_args:[] objs:[] fwd_guards:[] ↙
    bwd_guards:[] out_grps:[::grp]}))
!class (mdl (_obj {tpl_args:[] objs:[] fwd_guards:[] ↙
    bwd_guards:[] out_grps:[::grp] str:nb cnt:nb sr:nb dsr:nb}))
!class (icst (_obj {cst:cst tpl_args:[] args:[] wr_e:bl}))
!class (imdl (_obj {mdl:mdl tpl_args:[] args:[] wr_e:bl}))
!class (pred (_obj {obj:}))
!class (goal (_obj {obj: actr:ent}))
!class (success (_obj obj: evd:))
!class (mk.grp_pair (_obj {primary:grp secondary:grp}))
```

```
; performance counters (latencies).
!class (perf (_obj {rj_ltcy:nb d_rj_ltcy:nb tj_ltcy:nb ↙
   d_tj_ltcy:nb})); latencies and derivatives in us encoded as ↙
   floats.

; mapping operator opcodes to r-atoms.
!op (_now):us
!op (rnd :nb):nb
!op (equ : :):
!op (neq : :):
!op (gtr : :):
!op (lsr : :):
!op (gte : :):
!op (lse : :):
!op (add : :):
!op (sub : :):
!op (mul :nb :nb):nb
!op (div :nb :nb):nb
!op (dis : :):nb
!op (ln :nb):nb
!op (exp :nb):nb
!op (log :nb):nb
!op (e10 :nb):nb
!op (rmax : :):nb
!op (syn :):
!op (ins : :[] :bl :us :bl):
!op (red :[] :[] :[]):[]
!op (fvw : :):

; operator aliases.
!def now (_now)
!def = equ
!def <> neq
!def > gtr
!def < lsr
!def >= gte
!def <= lse
!def + add
!def - sub
```

```
!def * mul
!def / div
!def \ syn
!def max rmax

; mapping devices to r-atoms.
!def exe 0xA1000000; the executive.

; mapping device functions to r-atoms.
!dfn (_inj : :)
!dfn (_eje : : :nid)
!dfn (_mod : :nb)
!dfn (_set : :nb)
!dfn (_new_class :)
!dfn (_del_class :)
!dfn (_ldc :st)
!dfn (_swp :nb)
!dfn (_prb :nb :st :st :[])
!dfn (_stop)

; device functions aliases.
!def (inj args) (icmd _inj args)
!def (eje args) (icmd _eje args)
!def (mod args) (icmd _mod args)
!def (set args) (icmd _set args)
!def (new_class args) (icmd _new_class args)
!def (del_class args) (icmd _del_class args)
!def (ldc args) (icmd _ldc args)
!def (swp args) (icmd _swp args)
!def (prb args) (icmd _prb args)
!def (stop args) (icmd _stop args)

; various constants.
!counter __constant 0
!def OFF __constant
!def ON __constant

; parameters for tuning the behavior of reactive objects.
; member nfr.
!def SILENT false ; no notification upon production
```

```
!def NOTIFY true ; notification upon productions

; parameters for tuning the behavior of groups.
!def DCY_SLN 0
!def DCY_SLN_THR 1
!def COV_ON true
!def COV_OFF false

!def SYNC_FRONT true
!def SYNC_STATE false

!def RUN_ALWAYS true
!def RUN_ONCE false

; system internal constants.
!def MAX_TIME 18446744073709551615us; 2^64-1.
```

B.2 Chapter 3: Replicode

B.2.1 Hello world!

Code Listing B.4: The Hello World example.

```
hello_world:(pgm |[] |[] |[]
[]
   (inj []
      reply:(str "Hello World!" 1)
      [SYNC_FRONT now 1 1 root nil]
   )
1) |[]

ihello_world:(ipgm hello_world |[] RUN_ALWAYS 0us NOTIFY 1) []
   [SYNC_FRONT now 1 forever root nil 1]
```

B.2.2 Greeting program

Code Listing B.5: The Greeting example.

```
greeting_program:(pgm |[]
[]
    (ptn (str "Hello Replicode" ::) |[])
|[]
[]
    (inj []
        reply:(str "Hi there" 1)
        [SYNC_FRONT now 1 1 root nil]
    )
1) |[]

igreeting_program:(ipgm greeting_program |[] RUN_ALWAYS 0us ↙
    NOTIFY 1) []
    [SYNC_FRONT now 1 forever root nil 1]

;-------------------------------------------
; Test object

greeting:(str "Hello Replicode" 1) []
    [SYNC_FRONT now 1 forever root nil]
```

B.3 Chapter 4: An axiomatic approach to Replicode

B.3.1 Equality

Code Listing B.6: The Equality example.

```
run_if_p_is_q:(pgm |[]
[]
    (ptn f1:(fact (mk.val p is (rfol true) ::) ::) |[])
    (ptn f2:(fact (mk.val q is (rfol true) ::) ::) |[])
[]
    (= f1 f2)
[]
    (inj []
        a:(answer "p equals q?")
        [SYNC_FRONT now 1 1 root nil]
    )
1) |[]
```

```
irun_if_p_is_q:(ipgm run_if_p_is_q |[] RUN_ALWAYS 0us NOTIFY 1) ↙
   []
   [SYNC_FRONT now 1 forever root nil 1]



;-------------------------------------------
; Test object

p_is_true:(mk.val p is (rfol true) 1) []
   [SYNC_FRONT now 1 forever root nil]

fact_p_is_true:(fact p_is_true 0us 1000000us 1 1) []
   [SYNC_FRONT now 1 forever root nil]

q_is_true:(mk.val q is (rfol true) 1) []
   [SYNC_FRONT now 1 forever root nil]

fact_q_is_true:(fact q_is_true 0us 1000000us 1 1) []
   [SYNC_FRONT now 1 forever root nil]
```

B.3.2 Conjunction

Code Listing B.7: Example of conjunction.

```
run_if_p_q_r_s:(pgm |[]
[]
   (ptn (fact (mk.val p is (rfol true) ::) ::) |[])
   (ptn (fact (mk.val q is (rfol true) ::) ::) |[])
   (ptn (fact (mk.val r is (rfol true) ::) ::) |[])
   (ptn (fact (mk.val s is (rfol true) ::) ::) |[])
|[]
[]
   (inj []
      a:(answer "Found p, q, r and s!" 1)
      [SYNC_FRONT now 1 1 root nil]
   )
1) |[]

irun_if_p_q_r_s:(ipgm run_if_p_q_r_s |[] RUN_ALWAYS 0us NOTIFY ↙
   1) []
```

```
    [SYNC_FRONT now 1 forever root nil 1]



;------------------------------------------
; Test object

p_is_true:(mk.val p is (rfol true 0) 1) []
    [SYNC_FRONT now 1 forever root nil]

fact_p_is_true:(fact p_is_true 0us 1000000us 1 1) []
    [SYNC_FRONT now 1 forever root nil]

q_is_true:(mk.val q is (rfol true 0) 1) []
    [SYNC_FRONT now 1 forever root nil]

fact_q_is_true:(fact q_is_true 0us 1000000us 1 1) []
    [SYNC_FRONT now 1 forever root nil]

r_is_true:(mk.val r is (rfol true 0) 1) []
    [SYNC_FRONT now 1 forever root nil]

fact_r_is_true:(fact r_is_true 0us 1000000us 1 1) []
    [SYNC_FRONT now 1 forever root nil]

s_is_true:(mk.val s is (rfol true 0) 1) []
    [SYNC_FRONT now 1 forever root nil]

fact_s_is_true:(fact s_is_true 0us 1000000us 1 1) []
    [SYNC_FRONT now 1 forever root nil]
```

B.3.3 Disjunction

Code Listing B.8: Example of disjunction.

```
; p
run_if_p:(pgm |[]
[]
    (ptn (fact (mk.val p is (rfol true) ::) ::) |[])
|[]
[]
```

```
 (inj []
      p:(result 0); id for this program
      [SYNC_FRONT now 1 1 root nil]
   )
1) |[]

irun_if_p:(ipgm run_if_p |[] RUN_ALWAYS 0us NOTIFY 1) []
   [SYNC_FRONT now 1 forever root nil 1]

; q
run_if_q:(pgm |[]
[]
   (ptn (fact (mk.val q is (rfol true) ::) ::) |[])
|[]
[]
 (inj []
      p:(result 0); id for this program
      [SYNC_FRONT now 1 1 root nil]
   )
1) |[]

irun_if_q:(ipgm run_if_q |[] RUN_ALWAYS 0us NOTIFY 1) []
   [SYNC_FRONT now 1 forever root nil 1]

; p OR q
run_if_p_or_q:(pgm |[] []
   (ptn (result id:) []; guard on the result class,
      (= id 0); id must equal 0
   )
|[] []
   (inj []
      a:(answer "Found p or q!" 1)
      [SYNC_FRONT now 1 1 root nil]
   )
1) |[]

irun_if_p_or_q:(ipgm run_if_p_or_q |[] RUN_ALWAYS 0us NOTIFY 1) ↙
   []
   [SYNC_FRONT now 1 forever root nil 1]
```

```
;-------------------------------------------
; Test object

p_is_true:(mk.val p is (rfol true 0) 1) []
   [SYNC_FRONT now 1 forever root nil]

fact_p_is_true:(fact p_is_true 0us 1000000us 1 1) []
   [SYNC_FRONT now 1 forever root nil]

;q_is_true:(mk.val q is (rfol true 0) 1) []
;   [SYNC_FRONT now 1 forever root nil]

;fact_q_is_true:(fact q_is_true 0us 1000000us 1 1) []
;   [SYNC_FRONT now 1 forever root nil]
```

B.4 Chapter 5: Replicome v.1.0

B.4.1 Dangerous interactions

Code Listing B.9: Dangerous interactions

```
dangerous_interaction:(pgm |[]
[]
   (ptn (fact (mk.val o1: danger (interaction t1: "reagent" pv1:) ::) ↙
   ::) |[])
   (ptn (fact (mk.val o2: danger (interaction t2: "reactant" pv2:) ::) ↙
   ::) |[])
   (ptn (fact (mk.val o3: position p1:(vec2 : :) ::) ::) |[])
   (ptn (fact (mk.val o4: position p2:(vec2 : :) ::) ::) |[])
[]
   (<> o1 o2)
   (= o1 o4)
   (= o2 o3)
   (= t1 t2)
   (> (/ (* pv1 pv2) (max div_factor (dis p1 p2))) dgr_thr)
[]
   (cmd alarm [t1 p1] 1)
1) |[]
```

```
idangerous_interaction:(ipgm dangerous_interaction |[] ↙
   RUN_ALWAYS 0us NOTIFY 1) []
   [SYNC_FRONT now 1 forever root nil 1]

;------------------------------------------
; Test object
reagent:(ent 1) [[SYNC_FRONT now 0 forever root nil]]
reactant:(ent 1) [[SYNC_FRONT now 0 forever root nil]]

; Reactant
reactant_pos:(mk.val reactant position (vec2 2 0) 1) []
   [SYNC_FRONT now 1 forever root nil]

fact_reactant_pos:(fact reactant_pos 0us 1000000us 1 1) []
   [SYNC_FRONT now 1 forever root nil]

reactant_type:(mk.val reactant danger (interaction "blaze" ↙
   "reactant" 10) 1) []
   [SYNC_FRONT now 1 forever root nil]

fact_reactant_type:(fact reactant_type 0us 1000000us 1 1) []
   [SYNC_FRONT now 1 forever root nil]


reagent_pos:(mk.val reagent position (vec2 2 0) 1) []
   [SYNC_FRONT now 1 forever root nil]

fact_reagent_pos:(fact reagent_pos 0us 1000000us 1 1) []
   [SYNC_FRONT now 1 forever root nil]

reagent_type:(mk.val reagent danger (interaction "blaze" "reagent" ↙
   10) 1) []
   [SYNC_FRONT now 1 forever root nil]

fact_reagent_type:(fact reagent_type 0us 1000000us 1 1) []
   [SYNC_FRONT now 1 forever root nil]
```

B.4.2 Forgetfulness situation

Code Listing B.10: Dangerous interactions

```
remember_keys:(pgm |[]
[]
   (ptn (fact (mk.val subject position p1:(vec2 x: y:) ::) ::) |[])
   (ptn (fact (mk.val keys position p2:(vec2 x: y:) ::) ::) |[])
[]
   (= p1 exit)
   (<> p1 p2)
[]
   (cmd alarm ["Forgetting keys" p2] 1)
1) |[]

iforgetfulness:(ipgm remember_keys |[] RUN_ALWAYS 0us NOTIFY 1) ↙
   []
   [SYNC_FRONT now 1 forever root nil 1]



;-------------------------------------------
; Test object

; subject
subject_pos:(mk.val subject position (vec2 0 3) 1)[]
   [SYNC_FRONT now 1 forever root nil]

fact_subject_pos:(fact subject_pos 0us 1000000us 1 1) []
   [SYNC_FRONT now 1 forever root nil]

; keys
keys_pos:(mk.val keys position (vec2 1 2) 1) []
   [SYNC_FRONT now 1 forever root nil]

fact_keys_pos:(fact keys_pos 0us 1000000us 1 1) []
   [SYNC_FRONT now 1 forever root nil]
```

APPENDIX C

Extending the Replicode executive

In the implementation of the Replicome systems we use some non standard Replicode operators. The Replicode solution which contains the executable examples has been extended as described in this appendix.

C.1 Expanding the "dis" operator

At the time of writing, Replicode does not support the `dis` operator for `vec2` classes. On order for the `dis` operator to understand the `vec2` class we need to do two things.

First we need to register the opcode `vec2` such that Replicode knows of its existence. This is done in the constructor of the Operators class in the `usr_operators` project.

Code Listing C.1: The new Init function in operators.cpp

```
void    Operators::Init(OpcodeRetriever r){

    const   char    *vec3="vec3";
    const   char    *vec2="vec2";

    Vec3Opcode=r(vec3);
    Vec2Opcode=r(vec2);
}
```

The second thing we need to do is to give the algorithm that calculates the distance between two objects in two-dimensional space.

Code Listing C.2: The new dis function in operators.cpp

```
bool    dis(const      r_exec::Context &context, uint16 &index){

        r_exec::Context lhs=*context.getChild(1);
        r_exec::Context rhs=*context.getChild(2);

        if(lhs[0].asOpcode()==Vec3Opcode          &&        ↙
    rhs[0].asOpcode()==Vec3Opcode){

                float32 d1 = (*lhs.getChild(1))[0].asFloat() ↙
    - (*rhs.getChild(1))[0].asFloat();
                float32 d2 = (*lhs.getChild(2))[0].asFloat() ↙
    - (*rhs.getChild(2))[0].asFloat();
                float32 d3 = (*lhs.getChild(3))[0].asFloat() ↙
    - (*rhs.getChild(3))[0].asFloat();

                float32 norm2 = d1*d1 + d2*d2 + d3*d3;
                ↙
    index=context.setAtomicResult(Atom::Float(sqrt(norm2)));
                return true;

        }else if(lhs[0].asOpcode()==Vec2Opcode && ↙
    rhs[0].asOpcode()==Vec2Opcode){

                float32 d1 = (*lhs.getChild(1))[0].asFloat() ↙
    - (*rhs.getChild(1))[0].asFloat();
                float32 d2 = (*lhs.getChild(2))[0].asFloat() ↙
    - (*rhs.getChild(2))[0].asFloat();

                float32 norm2 = d1*d1 + d2*d2;
                ↙
    index=context.setAtomicResult(Atom::Float(sqrt(norm2)));
                return true;
        }

        index=context.setAtomicResult(Atom::Nil());
        return false;
}
```

This is all we need to be able to calculate distances between to `vec2` objects.

C.2 Creating the “max” operator

The `max` operator is completely alien to Replicode, so we need to do a bit more coding to add that function to Replicode. Because the function name max is reserved, we are actually going to create a `rmax` function which we give the alias `max` in the std.replicode file.

Code Listing C.3: Declaration of the max operator along with an alias. showlines

```
!op (rmax : :):nb; rmax declaration
!def max rmax; alias
```

Now we need to make modifications to three different files in the r_exec project. First we register the new operator in the constructor in init.cpp where all the operators are registered by adding this line:

Code Listing C.4: Registration of the rmax operator in the Init function in init.cpp

```
bool    Init(const      char    *user_operator_library_path, ↙
   uint64     (*time_base)()){


        ...
        Operator::Register(operator_opcode++,rmax);
        ...
}
```

Next we add the new function in the operator.h file:

Code Listing C.5: Registration of the rmax function in operator.h

```
namespace        r_exec{


        ...
        bool     rmax(const      Context &context,uint16 &index);
        ...
}
```

Finally we add the logic needed to give us the maximum of two numbers in the operator.cpp file.

Code Listing C.6: Logic of the rmax function in operator.cpp

```
namespace r_exec{

    ...
    bool rmax(const Context &context,uint16 &index){

        Context lhs=*context.getChild(1);
        Context rhs=*context.getChild(2);

        if(lhs[0].isFloat()){
            if(rhs[0].isFloat()){
                ↙
  index=context.setAtomicResult(Atom::Float( ↙
  max(lhs[0].asFloat(),rhs[0].asFloat())));
                return true;
            }
        }

        index=context.setAtomicResult(Atom::Nil());
        return false;
    }
}
```

First two conditionals are just to make sure that we have numbers, then we simply return the value that is returned from the C++ max function.

Appendix D

Replicode v1.1a grammar

The following grammar was built by using ANTLR, and is therefore on ANTLR format.

Code Listing D.1: Replicode grammar

```
grammar Replicode;
options {
  //output=AST; ASTLabelType=CommonTree;
 // backtrack=true;
}
// Author: Ólafur Hlynsson
// Rules are extracted from version 1.1a of the specification ↙
   for Replicode.


// begin bnf (used for automatic parsing)

//-----------------------------------------------------------------
//    Program structure

// NOTE: Abstract rules from which all rules are derived.
r_program
    :    (statements NEWLINE)+
    ;

// NOTE: Represents how the structure of a Replicode file can ↙
   be.
statements
    :    class_template_instantiations
```

```
     |    instantiated_macro_expression
     |    program
     |    directive
     ;


// NOTE: Derived from rules for negative and positive ↙
   preconditions, i.e. from list of directives.
directive
     :    undefine_directive
     |    macro_directive
     |    undefine_directive
     |    class_directive
     |    forward_class_directive
     |    operator_directive
     |    macro_expression_directive
     |    load_directive
     |    counter_directive
     |    device_function_directive
     |    negative_precondition_directive
     |    positive_precondition_directive
     ;


// NOTE: Defined on page 14.
program
     :    '(pgm' WS '[' set_of_patterns ']' WS '[' ↙
   set_of_patterns WS set_of_timing_constraints WS ↙
   set_of_guards ']'  WS set_of_productions WS set_of_views ↙
   WS other_data')'
     ;

// NOTE: Defined on page 56.
view
     :    '[' sync_value WS injection_time WS saliency WS ↙
   resilience WS origin']'
     ;


// NOTE: Defined on page 56.
```

```
resilience
    :    FLOAT
    ;

// NOTE: Defined on page 56.
saliency
    :    PERCENTAGE
    ;

// NOTE: Defined on page 12.
pattern
    :    '(ptn' WS skeleton set_of_guards ')'
    ;

// NOTE: Defined on page 12. Guard is an expression.
guard
    :    expression
    ;

// NOTE: Definition starts on page 38.
atom
    :    'nil' | '[]' | '|[]' | '|nb' | '|us' | '|did' ↙
   |'|fid' | '|nid' | '|bl' | '|st' | 'forever' | 'true' | ↙
   'false'
    |    a_name ':' expression
    |    a_name '#' expression ':'
    |    a_name ':'
    |    a_name
    |    ':'
    |    '::'
    |    FLOAT
    |    FLOAT '|' FLOAT
    |    time_stamp
    |    time_stamp '|' time_stamp
    |    string
    |    'this'
    |    'this' '.' a_member
    |    a_name '.' a_member
    |    opcode
    |    identifier
```

```
      |    index
      ;

// NOTE: Defined on page 39.
expression
      :    '(' opcode WS list_of_elements ')'
      ;

// NOTE: Defined on page 39.
opcode
      :    operator
      |    marker_class
      |    object_class
      ;

// NOTE: Defined on page 48.
marker_class
      :    ( MK_XET | MK_RDX | MK_ARDX | MK_LOW_SLN | ↙
    MK_HIGH_SLN | MK_LOW_ACT | MK_HIGH_ACT | MK_LOW_RES | ↙
    MK_SLN_CHG )
      ;


// NOTE: Defined on page 47.
operator
      :    ( EQU | NEQ | GTR | LSR| GTE | LSE | NOW | ADD | SUB ↙
    | MUL | DIV | DIS | LN | EXP | LOG | E10 | SYN | INS | RED ↙
    | RND | FVW)
      ;


// ####### Refactor block 1 ##############################
// Refactor to RNAME?
// Note: Defined on page 38 and on page 43. In the definition ↙
    on page 38 it is defined as alphanumerical identifier and ↙
    on page 43 it is defined as a macro name and a counter name.
a_name
      :    alphanumerical_identifier
      |    a_macro_name
      |    a_counter_name
    ;

// NOTE: Defined on page 39.
a_member
    :   alphanumerical_identifier
    ;

// NOTE: Defined on page 39. Not used anywhere.
label
    :    RNAME ':'
    ;

// Note: structure of named variables, labels and members are ↙
   on page 39.
alphanumerical_identifier
    :    LITERAL+ ( INT | LITERAL )*
    ;
// ####### End Refactor block 1 ##########################

// Note: Defined on page 38.
time_stamp
    :    FLOAT 'us'
    ;

// NOTE: Defined on page 39.
set
    :    '[' list_of_elements ']'
    |    list_of_elements
    ;

// NOTE: Defined on page 39.
separators
    :    WS
    |    NEWLINE
    |    line_continuation
    ;

// NOTE: Defined on page 39.
line_continuation
    :    '' NEWLINE
    ;

string
    :   '"' LITERAL  '"'
    ;


// NOTE: Defined on page 39. Not clear if the list can ↙
   contain mixed elements (líklega þó).
list_of_elements
    :    element (WS element)*
    ;

// NOTE: Defined on page 43.
element
    :    expression
    |    set
    |    atom
    |    a_macro_name
    |    a_counter_name
    |    instantiated_macro_expression
    ;

// NOTE: Defined on page 43.
instantiated_macro_expression
    :    '(' a_macro_name WS list_of_elements ')'
    ;


// NOTE: Defined on page 42.
macro_directive
    :    '!def' WS a_macro_name WS macro_replacement
    ;

// NOTE: Defined on page 43.
macro_expression_directive
    :    '!def' WS macro_expression WS macro_replacement
    ;

// NOTE: Defined on page 43.
```

```
macro_replacement
     :     expression
     |     set
     |     atom
     |     development
     ;

// NOTE: Defined on page 43.
class_directive
     :    '!class' WS '('class_name (WS ↙
    list_of_member_declarations)? ')'
     ;


// NOTE: Defined on page 44.
class_name
     :    LITERAL '[]'?
     ;

// NOTE: Defined on page 43.
development
     :    '{' list_of_elements '}'
     ;

// NOTE: Defined on page 43.
macro_expression
     :   '(' a_macro_name WS list_of_variables ')'
     ;

// NOTE: Defined on page 43.
a_macro_name
     :    '|'? LITERAL
     ;

// NOTE: Defined on page 43.
undefine_directive
     :    '!undef' WS a_name
     ;

// NOTE: Defined on page 42.
```

```
counter_directive
    :    '!counter' WS a_counter_name WS initial_value
    ;

// NOTE: Defined on page 42.
initial_value
    :    a_counter_name
    |    INT
    ;

// NOTE: Defined on page 42.
a_counter_name
    : '_' '_' LITERAL
    ;

// NOTE: Defined on page 43.
variable
    :    ':' variable_name
    ;

// NOTE: Defined on page 43.
positive_precondition_directive
    :    '!ifdef' WS a_name NEWLINE
         '!else' WS NEWLINE
         list_of_directives NEWLINE
         '!endif' NEWLINE
    ;

// NOTE: Defined on page 43.
negative_precondition_directive
    :    '!ifundef' WS a_name NEWLINE
         '!else' WS NEWLINE
         list_of_directives NEWLINE
         '!endif' NEWLINE
    ;

// NOTE: Defined on page 44.
type
    :    'nb'
    |    'us'
```

```
        |    'bl'
        |    'sid'
        |    'fid'
        |    'did'
        |    '[]'
        |    'st'
        |    class_name
        |    '[' list_of_member_declarations ']'
        |    '[' iterative_member_declaration ']'
        |    ''
        ;

// NOTE: Defined on page 44.
iterative_member_declaration
        :    '::' type
        ;

// NOTE: Defined on page 44.
forward_class_directive
        :   '!class' WS class_name
        ;

// NOTE: Defined on page 44.
member_declaration
        :    member_name ':' type
        |    member_name ':'
        |    ':' type
        |    '{' list_of_member_declarations '}'
        |    ':'
        |    'nil'
        ;

// NOTE: Defined on page 44 - 45.
class_template_instantiations
        :    '(' class_template WS list_of_member_declarations ')'
        |    member_name ':' '(' class_template WS ↙
    list_of_member_declarations ')'
        ;

// NOTE: Defined on page 45.
```

```
operator_directive
    :    '!op' WS '(' operator_name WS ↙
   list_of_member_declarations ')'
    ;

// NOTE: Defined on page 45.
anonymous_member_declaration
    :    ':' type
    ;

// NOTE: Defined on page 45.
device_function_directive
    :    '!dfn' WS '(' function_name WS ↙
   list_of_anonymous_member_declarations ')'
    ;

// NOTE: Defined on page 45.
load_directive
    :    '!load' WS file_locator
    ;

// NOTE: Defined on page 45.
file_locator
    :    LITERAL
    ;

// ####### Refactor block 2 ############################
// Elements here could be renamed to one commom token.
// NOTE: Defined on page 43.
variable_name
    :   LITERAL
    ;

// NOTE: Defined on page 44.
member_name
    :    LITERAL
    ;

// NOTE: Not defined.
function_name
```

```
     :    LITERAL
     ;

// NOTE: Not defined.
operator_name
     :    LITERAL
     ;
// ####### End Refactor block 2 ##########################



// NOTE: Same definition as for index on page 39, not certain ↙
   if this is right.
identifier
     :    INT
     ;

// NOTE: Same definition as for identifier on page 39, not ↙
   certain if this is right.
index
     :    INT
     ;


//  These are not defined properly.
// NOTE: Not defined anywhere, derived from the rule for ↙
   program. Subject for debate.
set_of_patterns
     :    '[' pattern (WS pattern)* ']'
     ;

// NOTE: Not defined anywhere, derived from the rule for ↙
   program. Subject for debate.
set_of_timing_constraints
     :   '[' timing_constraint (WS timing_constraint)* ']'
     ;

// NOTE: Not defined anywhere, derived from the rule for ↙
   program. Subject for debate.
set_of_guards
```

```
     :    '[' guard (WS guard)* ']'
     ;

// NOTE: Not defined anywhere, derived from the rule for ↙
   program. Subject for debate.
set_of_views
     :    '[' view (WS view)* ']'
     ;

// NOTE: Not defined anywhere, derived from the rule for ↙
   program.
set_of_productions
     :    '[' production (WS production)* ']'
     ;

// NOTE: Not defined anywhere, derived from other rules.
list_of_member_declarations
     :    member_declaration (WS member_declaration)*
     ;

// NOTE: Not defined anywhere, derived from other rules.
list_of_anonymous_member_declarations
     :    anonymous_member_declaration (WS ↙
   anonymous_member_declaration)*
     ;

// NOTE: Not defined anywhere, derived from other rules.
list_of_directives
         :       directive (WS directive)*
         ;

// NOTE: Not defined anywhere, derived from other rules.
list_of_variables
     :    variable (WS variable)*
     ;


// NOTE: Not defined.
class_template
     :    class_name
     ;

// NOTE: Not defined.
origin
     :    'nil'
     |    group_name
     ;

// NOTE: Not defined.
group_name
     :    LITERAL
     ;

// NOTE: Not defined.
sync_value
     :    SYNC_FRONT
     |    SYNC_BACK
     ;


// NOTE: Not defined.
injection_time
     :    'now'
     |    FLOAT
     ;

// NOTE: Defined on page 47. Referred to in the definition of ↙
   opcode.
object_class
     :    ( 'view' | 'ptn' | '|ptn' | 'pgm' | '|pgm' | 'ipgm' ↙
    | 'grp' | 'rgrp' | 'fun' | 'ifun' | 'fact' | '|fact' | ↙
    'hyp' | 'sim' | 'pred' | 'goal' | 'assm' | 'cmd' | 'dev' | ↙
    'nod' )
     ;


// NOTE: Not defined anywhere, not sure what exactly is ↙
   allowed here. What is known to be allowed is the inj, mod ↙
   and set commands.
production
```

```
     :    ''
     ;

// NOTE: Not defined anywhere, not sure what exactly is ↙
   allowed here.
other_data
     :    FLOAT
     ;

// NOTE: Not defined anywhere, not sure what exactly is ↙
   allowed here.
skeleton
     :   ''
     ;

// NOTE: Not defined anywhere, not sure what exactly is ↙
   allowed here. It has been hinted that timing constraints ↙
   are syntactically equivalent to guards, hence expression.
timing_constraint
     :    expression
     ;


// NOTE: Not defined specifically, naming is subject for ↙
   debate.
PERCENTAGE
     :    '1'
     |    '0'? '.' ('0'..'9')
     |    '0'
     ;

// NOTE: Defined on page 42.
LITERAL
     :    '|'? ('a'..'z'|'A'..'Z'|'_')+
     ;

FLOAT
     :    ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
     |    '.' ('0'..'9')+ EXPONENT?
     |    ('0'..'9')+ EXPONENT
```

```
    ;

EXPONENT
    :    ('e'|'E') ('+'|'-')? ('0'..'9')+
    ;

SYNC_FRONT
    :    'SYNC_FRONT'
    ;

SYNC_BACK
    :    'SYNC_BACK'
    ;

MK_XET
    :    'mk.xet'
    ;

MK_RDX
    :    'mk.rdx'
    ;

MK_ARDX
    :    'mk.|rdx'
    ;


MK_LOW_SLN
    :    'mk.low_sln'
    ;

MK_HIGH_SLN
    :    'mk.high_sln'
    ;

MK_LOW_ACT
    :    'mk.low_act'
    ;

MK_HIGH_ACT
```

```
    :    'mk.high_act'
    ;

MK_LOW_RES
    :    'mk.low_res'
    ;

MK_SLN_CHG
    :    'mk.sln_chg'
    ;

WS
    :    ' '
    ;

NEWLINE
    :    '\r'
    ;

RNAME
    :     ('a'..'z'|'A'..'Z') ↙
   (('a'..'z'|'A'..'Z')|'_'|'0'..'9')*
    ;

INT
    :    '0'..'9'+
    ;

EQU
    :    '='
    ;

NEQ
    :    '<>'
    ;

GTR
    :    '>'
    ;
```

```
LSR
      :     '<'
      ;

GTE
      :     '>='
      ;

LSE
      :     '<='
      ;



ADD
      :     '+'
      ;

SUB
      :     '-'
      ;

MUL
      :     '*'
      ;

DIV
      :     '/'
      ;

DIS
      :     'dis'
      ;

LN
      :     'ln'
      ;

EXP
      :     'exp'
```

```
    ;

LOG
    :   'log'
    ;

E10
    :   'e10'
    ;

SYN
    :   '\\'
    ;

INS
    :   'ins'
    ;

RED
    :   'red'
    ;

RND
    :   'rnd'
    ;

NNOW
    :   '_now'
    ;

FVW
    :   'fwv'
    ;

// NOTE: Defined as an operator.
NOW
    :   'now'
    ;
```

Bibliography

AAL-Europe (2012). Ambient assisted living. http://www.aal-europe.eu.

Augusto, J. C., Liu, J., Mccullagh, P., Wang, H., and bo Yang, J. (2005). Management of uncertainty and spatio-temporal aspects for monitoring and diagnosis in a smart home. Technical report.

Cacciagrano, D., Corradini, F., and Culmone, R. (2010). Resourcehome: An rfid-based architecture and a flexible model for ambient intelligence. In *Proceedings of the 2010 Fifth International Conference on Systems*, ICONS '10, pages 6–11, Washington, DC, USA. IEEE Computer Society.

Cacciagrano, D., Corradini, F., Merelli, E., Vito, L., and Romiti, G. (2009). Resourceome: A multilevel model and a semantic web tool for managing domain and operational knowledge. In *Proceedings of the 2009 Third International Conference on Advances in Semantic Processing*, SEMAPRO '09, pages 38–43, Washington, DC, USA. IEEE Computer Society.

CADIA (2012). Humanobs. http://www.humanobs.org/.

Cycorp (2012a). Cyc. http://www.cyc.com/.

Cycorp (2012b). Opencyc. http://www.opencyc.org/.

Eric Nivel, N. T. Y. B. (2012). Replicode language specification. Technical Report 2.2.

Foundation, O. (2012a). Opencog. http://opencog.org/.

Foundation, O. (2012b). Opencog prime. http://wiki.opencog.org/w/OpenCogPrime.

Hart, D. and Goertzel, B. (2008). Opencog: A software framework for integrative artificial general intelligence. In *Proceedings of the 2008 conference on Artificial General Intelligence 2008: Proceedings of the First AGI Conference*, pages 468–472, Amsterdam, The Netherlands, The Netherlands. IOS Press.

Horatiu Cirstea, Claude Kirchner, M. M. P.-E. M. (2004). Production systems and rete algorithm formalisation.

Jeff Hawkins, D. G. (2012). Hierarchical temporal memory. http://en.wikipedia.org/wiki/Hierarchical_temporal_memory.

Khakpour, N., Jalili, S., Talcott, C. L., Sirjani, M., and Mousavi, M. R. (2010). Pobsam: Policy-based managing of actors in self-adaptive systems. *Electr. Notes Theor. Comput. Sci.*, 263:129–143.

Laura Aureli, S. S. F. L. (2011). Automated environment for objects research and safety control.

Maude (2012). The real-time maude tool. http://heim.ifi.uio.no/peterol/RealTimeMaude/.

MedicineNet (2012). Definition of adl (activities of daily living). http://www.medterms.com/script/main/art.asp?articlekey=2152.

Menzies, P. (2009). Counterfactual theories of causation. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Fall 2009 edition.

Numenta (2012). Prologue. http://www.onintelligence.org/excerpt.php.

OMG, O. M. G. (2009). Production rule representation (prr). Version 1.0.

Pearl, J. (2009). *Causality: Models, Reasoning and Inference - 2nd ed.* Cambridge University Press, 32 Avenure of the Americas, New York, NY 10013-2473, USA.

Piaget, J. (2012). Piaget's theory of cognitive development. http://en.wikipedia.org/wiki/Piaget's_theory_of_cognitive_development.

Project, H. (2012a). Smart home for elderly people (hope). http://www.hope-project.eu/.

Project, R. (2012b). Guidance and awareness services for independent living. www.aal-rosetta.eu/.

Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition.

Snyder, W. and Schmolze, J. G. (1996). Rewrite semantics for production rule systems: Theory and applications. In *CADE*, pages 508–522.

Thorisson, K. (2009). From constructionist to constructivist a.i.

Wang, P. (1995). *Non-axiomatic reasoning system - Exploring the essence of intelligence*. PhD thesis, Indiana University.

Wang, P. (2004). Experience-grounded semantics: A theory for intelligent systems.

Wang, P. (2006). *Rigid Flexibility: The Logic of Intelligence (Applied Logic Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Wang, P. (2010). Non-axiomatic logic (nal) specification.

School of Computer Science
Reykjavík University
Menntavegi 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.reykjavikuniversity.is
ISSN 1670-8539

University of Camerino,
Piazza Cavour 19 / f 62032 Camerino (MC)
P.IVA: 00291660439
CF 81001910439
http://www.unicam.it/