Tools and Techniques for Multi-Valued Networks using Rewriting Logic

PhD Thesis

Alhumaidan, Abdullah Saleh A

Supervisor: Dr Jason Steggles

School of Computing

Newcastle University

This dissertation is submitted for the degree of

Doctor of Philosophy

February 2019

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Alhumaidan, Abdullah Saleh A

February 2019

Acknowledgements

First and foremost, my parents Saleh and Norah have given me nothing but love and support throughout my PhD studies, and for that I am very grateful.

I would like to thank my supervisor Jason Steggles, who has gone above and beyond to make sure work was running smoothly by always making time to have regular meetings throughout the course of my research.

I would also like to thank the Ministry of Higher Education in Saudi Arabia for sponsoring my studies over the past few years.

Outside of work, I thank my friends in Newcastle for making my life in the north east a very social and enjoyable experience.

Abstract

Multi-valued networks (MVNs) are an important, widely used qualitative modelling technique where time and states are discrete. MVNs extend the well-known Boolean networks by providing a more powerful qualitative modelling approach for biological systems by allowing an entity's state to be within a range of discrete set of values instead of just 0 and 1. They provide a logical framework for qualitatively modelling and analysing control systems and have been successfully applied to biological systems and circuit design. While a range of support tools for developing and analysing MVNs exist, more work is needed to develop tools to support the practical applications of those techniques.

One of the frameworks that have been successfully applied to biological systems is *Rewriting Logic (RL)*, an algebraic specification framework that is capable of modelling and analysing the behaviour of dynamic, concurrent systems. The flexibility of RL techniques such as implementation of strategies has allowed it to be successfully used to model a wide range of different formalisms and systems, such as process algebras, Petri nets, and biological systems. RL specification, programming and computation is supported by a range of powerful analysis tools which was one of the motivations for choosing to use RL. We choose Maude as a tool in our work here which is a high-performance reflective language supporting both equational and RL specification. Maude is going to be used through this thesis to model and analyse a range of MVNs using RL.

In this thesis we aim to investigate the application of RL to modelling and analysing both synchronous and asynchronous MVNs, thus enabling the application of support tools available for RL. We start by constructing an RL model for MVNs using a translation approach that translates an MVNs set of equations into rewrite rules. We formally show that our translation approach is correct by proving its soundness and completeness. We illustrate the techniques and the developed RL framework for MVNs by presenting a range of case studies which provides a good illustration of the practical application of the developed RL framework. We then introduce an artificial, scalable MVN model in order to allow a range of model sizes to be considered and we investigate the performance of our RL framework. We analyse a larger regulatory network from the literature using our RL framework to give some insights into how it coped with a larger case study.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Context

Multi-valued networks (MVNs) [2, 3, 72] are an important, widely used qualitative modelling technique for biological systems where time and states are discrete. MVNs extend the well-known Boolean networks [72, 78, 80] allowing an entity's state to be within a range of discrete set of values instead of just 0 and 1 [52]. They provide a logical framework for qualitatively modelling and analysing control systems and have been successfully applied to biological systems [3, 4, 29, 5] and circuit design [2, 30]. While a range of support tools for developing and analysing MVNs exist, more work is needed to develop tools to support the practical applications of those techniques in the biological setting.

Rewriting Logic (RL) [34,36] is an algebraic specification framework that is capable of modelling and analysing the behaviour of dynamic, concurrent systems. It has been successfully used to model a wide range of different formalisms and systems, such as process algebras [10, 11], Petri nets [12, 13], and biological systems [14, 15]. RL specification and programming is supported by Maude [28], a high-performance reflective language for a wide range of applications.

In this thesis we investigate using Maude to model and analyse a range of MVNs using RL. Maude supports logical reflection in a systematic way thus making it remarkably extensible and powerful. It allows many advanced metaprogramming and metalanguage applications such as rewriting strategies[17, 41]. It also has a range of analysis tools such as a built-in model checker and the LTL model checker. Some of the most interesting applications of Maude are metalanguage applications where it is used to create executable environments for different languages and models of computation [28].

Techniques were developed during this work to construct RL models of MVNs that can be analysed using Maude. We define a semantic approach that translates the behaviour of an MVN to an RL model and we provide an argument for its correctness by showing its soundness and completeness. We develop tool support for this which takes in a description of an MVN and generates an RL model and a set of rewriting rules that model the behaviour of the MVN. We test the performance of our RL framework by developing a scalable test model that can be used to construct a range of test models of different sizes. We test the performance of our RL framework by performing a range of searches as well as LTL checks for the generated models for both synchronous and asynchronous semantics before we analyse a larger regulatory network from the literature using our RL framework and give some insights into how did it cope with a larger case study.

1.2 Aim and Contributions

This thesis sets out to develop RL techniques to support the use of MVNs. The high-level aim of this thesis can be stated as follows:

This thesis aims to investigate the application of RL to modelling and analysing MVNs, and to develop a set of tools and techniques to support this.

We set out to strengthen the tool support available for MVNs by developing translation techniques that translate the behaviour of an MVN to an RL model and so enabling the application of the support tools available for RL. A support tool was implemented in Java to implement those translation techniques by producing RL models when given a description of an MVN. We test the performance of our RL framework by developing a scalable test model that can be used to construct a range of test models of different sizes before we analyse a larger regulatory network from the literature using our RL framework and give some insights into how did it cope with a larger case study.

This thesis sets out to develop and evaluate the use of RL in modelling and analysing MVNs. In particular, this thesis will aim to contribute to the literature in the following key aspects:

1. **Define a semantic translation from an asynchronous MVN into an RL model and formally prove the correctness of the translation approach.**

   We start by constructing an RL model for MVNs based on their asynchronous update semantics, and we derive a set of rewrite rules that are used to capture an MVN's behaviour. We represent within our model the next state function associated with each entity and the associated asynchronous update rules using a translation approach that takes in a structured MVN file and produce an RL Model with required definitions and set of rewrite rules. Techniques and the developed RL framework are illustrated during that process alongside the resulting analysis possible by presenting a detailed case study. We use a published MVN model from the literature that is used for modelling and analysing the genetic regulatory network [74] for the synthesis of Tryptophan in *E. coli* [21, 22] as a case study for our RL model for asynchronous MVNs. The case study helps motivate our RL framework by illustrating the analysis techniques and tools when using RL and the tool Maude. We formally investigate the model using a range of powerful analysis tools provided by Maude, which was one of the motivations for choosing to use RL. We formally show that our translation process from an asynchronous MVN to an RL model is correct, that is done by showing and proving the soundness and completeness of our translation process. To do this we show that: 1) (*soundness*) each global state transition possible in our RL model represents a corresponding asynchronous update in the original MVN; and 2) (*completeness*) every asynchronous update possible in an MVN is specified in our RL model.

2. **Define a semantic translation from a synchronous MVN into an RL model and formally prove the correctness of the translation approach.**

   In a synchronous MVN, all entities update their state simultaneously and this leads to deterministic, infinite traces. While asynchronous MVNs are seen as being more realistic, they can result in too much behaviour and their dynamics can be more difficult to analyse. Synchronous MVNs are seen as being less realistic than asynchronous MVNs [12] because of their assumption of simultaneous updates. However, their dynamics can be easier to analyse and this has made them very popular. We provide a formal translation of synchronous MVNs to RL. The challenge here lies in the ability to coordinate update steps to ensure that entities are using the current states of other entities as inputs rather than next states. In order to handle this we define a rewriting strategy[64] to apply a two phase state update which is defined as follows: First, we define a metalevel operation that applies rewrite rules using current entities' states using synchronous rewrite rules. We then build on this by defining a metalevel operation that resets the current state of each entity after applying the first phase, and that gives us all we need to analyse our model using Maude. We illustrate the techniques and the RL framework developed for synchronous MVNs by presenting a case study using an existing MVN model for the genetic regulatory network controlling the lysis–lysogeny switch in the bacteriophage λ [25, 4]. The case study helps to motivate our RL framework by illustrating the analysis techniques and tools when using RL and Maude. We formally investigate the model using simple model checking based on the search command and LTL model checking.

3. **Develop a scalable benchmark test model to carry out a formal investigation into the scalability of the developed RL framework and it's performance.**

   The case studies presented for our RL framework for asynchronous and synchronous MVNs provide a good illustration of the practical application of the RL techniques we have developed. However, as they provide little indication of how the developed RL approach would scale when applied to larger MVN models and what impact the well–known state space explosion problem would have, we address this by defining an artificial, scalable MVN model in order to allow a range of model sizes to be considered and we investigate how our RL framework performs as the MVN size (i.e. number of entities) increases. We use a test model generation approach where we create a series of MVN test models in incremental steps of 5 and for every new test model we add, we connect an entity of the new model to a connecting entity in the previous test model to generate a more complicated behaviour within the test model. We use this scalable test model (both synchronous and asynchronous versions) to test the performance of our RL framework using Maude's search command with regards to states visited during different search commands and also with regards to time. We start our analysis for our scalable model with a basic model of 5 entities. Then we extend it repeatedly by adding a new block of entities and we perform some tests using the search command and the LTL model checker to come up with some performance results for our RL framework to give an idea on how well would our our RL framework cope with the scalable test model. We present our results for both asynchronous and synchronous versions of our model using a testing approach where we make use of Maude's search command as well as LTL.

4. **Develop an integrated toolset to support the developed RL framework and perform automatic translation from an MVN to an RL model and vice versa.**

   We develop an integrated toolset that is capable of studying and analysing MVNs behaviour. The toolset was implemented in Java to implement translation techniques that translate the behaviour of an MVN to an RL model thus enabling the application of the support tools available for RL. The toolset produces synchronous and asynchronous RL models containing a set of rewrite rules when given a description of an MVN's behaviour .We use the tool support available for RL (in this case Maude) to study and analyse the dynamical behaviour of a wide range of MVN case studies using the RL models generated by the toolset.

5. **Undertake a biologically relevant case study from the literature to investigate the practical applications of the developed RL framework and new RL techniques and tools using a parametric model.**

   We present a case study from the literature to test the developed tools and techniques where we analyse the gene regulatory network of the segment polarity gene family at the basis of *Drosophila* embryonic development. The network has been investigated through mathematical modelling to determine the network's capacity for generating and maintaining a specific gene expression pattern. During the initial stages of development of the fruit fly [96], three families of genes are successfully activated [95]: the gap genes; the pair-rule genes; and the segment polarity genes. A first mathematical model was proposed in an attempt to understand and study this network and its properties. Improvements to the model were introduced in [86], including an alternative mathematical description and analysis of its properties have been presented since [93]. We analyse this regulatory network using the tool support and the RL framework we develop and give some insights into how well they coped with a larger case study.

1.3 Organisation of Thesis

This thesis is organised as follows:

Chapter 2 Background

Introduces MVNs and explores their application and the available tool support, and introduces RL, an algebraic specification framework for modelling and analysing the behaviour of dynamic, concurrent systems. We introduce Maude, a high-performance reflective language supporting both equational and rewriting logic specification and programming and give a few examples to illustrate the Maude syntax by introducing rewriting, search and Linear temporal logic commands. We give an example of using rewriting strategies to control the rewriting performed by Maude. We introduce the tool support that was developed during the course of this thesis by giving a brief summary of the available functions within the toolset.

Chapter 3 An RL Model for Asynchronous MVNs

Builds on existing RL tool support to construct an RL model for asynchronous MVNs deriving a set of rewrite rules capturing a system's behaviour. We use a translation approach to translate an MVN behaviour to a set of rewrite rules and prove the correctness of the developed RL framework by formally showing its soundness and completeness. We illustrate the developed RL framework by presenting a case study from the literature and formally investigate the model using the range of powerful analysis tools provided by Maude.

Chapter 4 An RL Model for Synchronous MVNs

Follows on Chapter 3 by Constructing an RL model for synchronous MVNs deriving a set of synchronous rewrite rules capturing a system's behaviour. We use a translation approach to translate the synchronous behaviour of an MVN to a set of rewrite rules and prove the correctness of our translation approach by formally showing its soundness and completeness. We illustrate the developed RL framework for synchronous MVNs by presenting a case study from the literature an formally investigate the model using Maude.

Chapter 5 Performance Evaluation

We test the tools and techniques developed in Chapters 3 and 4 by introducing a scalable test model. We then introduce our testing approach that provides three search tests and two tests that are based on the LTL model checker. We start our analysis using a model of five entities and analyse it using the set of predefined tests. We then scale our model in increments of 5 by adding a new block of 5 entities and perform the same set of tests. We introduce both asynchronous and synchronous versions of our test model for these tests and provide a summary of results for each model size at the end of our tests.

Chapter 6 Case Study

Tests the developed tools and techniques in a larger case study from the literature. We introduce the model and our testing approach before we start testing our RL frame work using a set of defined tests for both asynchronous and synchronous versions of our model. We give a summary of our results and discuss the performance of our RL framework when applied to larger case studies.

Chapter 7 Concluding Remarks
Concludes this thesis by giving a brief summary of the thesis. We discuss the aim of this thesis and how well it was met by discussing key insights. We introduce some conclusions and discuss how well would our RL framework compare to existing tools such as the use of Petri nets. We discuss a number of interesting areas of future research that can be done to take this work forward.

1.4 Publications

The work presented in Chapters 3, 4 and 5 has been published as a journal paper [42] in $Fundamenta\ Informaticae$. This paper was jointly written with my supervisor who provided the initial idea for the translation into RL and also provided support and advice during the development of this work.

Chapter 2

Background

2.1 Introduction

We begin by introducing *Multi-valued networks* [2, 3] which are mathematical models of control systems where time and states are discrete. MVNs extend the well-known Boolean networks [3, 75] by providing a more powerful qualitative modelling approach for biological systems; they extend Boolean networks by allowing an entity's state to be within a range of discrete set of values instead of just 0 or 1.

We introduce Rewriting Logic which is an algebraic specification framework that is capable of modelling and analysing the behaviour of dynamic, concurrent systems. RL has been successfully used to model a wide range of different formalisms and systems, such as process algebras [10, 11], Petri nets [12, 13], and biological systems [14, 15]. After introducing RL we then introduce Maude, which is a high-performance reflective language supporting both equational and rewriting logic [41, 70] specification and programming for a wide range of applications.

We introduce the syntax of Maude and give some examples using models that were introduced alongside the introduction to MVNs.We provide some examples of using Maude's search and rewrite commands to explore certain properties of a model using our running examples. We extend our analysis by using Maude's built in Linear Temporal Logic (LTL) model checker and give some examples of some uses of LTL.

This chapter is organised as follows, we introduce Multi-valued networks in Section 2.3. In Section 2.4, we give an introduction to rewriting logic and give an example of using RL using a simple dynamic system. We then introduce Maude in Section 2.5 and discuss Maude's specifications using the same dynamic system introduced in Section 2.4, we give an example of performing rewrites in Maude before we introduce the search command and Linear Temporal Logic(LTL) which we use to model check our running example before introducing rewrite strategies in Section 2.5.5. In Section 2.6 we discuss existing tools and techniques in the literature before concluding the chapter with concluding remarks in Section 2.9.

2.2 Genetic Regulatory Networks

DNA (deoxyribonucleic acid) is an important molecule found in every living cell which contains information needed for constructing key molecules, called proteins, that control the construction and operation of a cell. DNA is primarily composed for genes, small units of DNA that directly represent a protein. The proteins produced by genes can regulate the use of other genes and this leads to complex regulation networks known as Gene Regulatory Networks (GRNs) [108, 109]. For an introduction to GRNs see [108]. We now give a list of the terminologies relevant to GRN's [108] which are used in the sequel:

• **DNA**: The molecule used in all living cells to code information about the construction and operation of the cell.

• **Gene**: A unit found in DNA that codes for a protein.

• **Protein**: An important molecule used by cells for constructing biological entities and controlling biological processes.

• **Transcription**: A key step in the process of using a gene to produce a protein.

• **Gene expression**: When a gene is being used to produce a protein.

• **Enzyme**: An important type of protein that promotes a chemical reaction.

• **Activation**: Facilitating the expression of a gene.

- **Inhibition**: Preventing the expression of a gene.
- **Repressor**: A molecule that inhibits the expression of a gene.

Figure 2.1 gives an example of a GRN which is found in an organism called bacteria phage lambda that controls a key biological behaviour known as the lysis-lysogeny switch (where lysis is the destruction of a cell and lysogeny is the process of embedding itself within another cell) [31]. It consists of two genes CI and Cro which behave as follows: the entity Cro inhibits the expression of CI (stops it producing its protein) and at high-levels of expression it also inhibits itself; gene CI inhibits the expression of Cro and promotes its own expression.

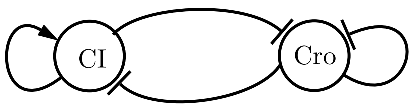


Fig. 2.1 The GRN for the lysis-lysogeny switch in bacteriophage lambda(based on [25]).

2.3 Multi-Valued Networks

Multi-Valued networks (MVNs) [2, 25, 3] extend the well-known Boolean networks [2, 3] (which are mathematical models of control systems) by providing a more powerful qualitative modelling approach for biological systems; they extend Boolean networks by allowing an entity's state to be within a range of discrete set of values instead of just 0 and 1.

Multi-valued networks provide a logical framework for qualitatively modelling and analysing control systems. They have been successfully applied to biological systems [3, 4, 29, 5] and circuit design [2, 30]. In this section we introduce the basic definitions for MVNs and provide an illustrative example.

An MVN global state is the composition of that MVN's entities' values at any given time. Entities update their values either synchronously [24, 44, 83] or asynchronously [33, 81] which can be more biologically realistic [79]. The number of network states grows exponentially with the number of entities, this is called the state-space explosion problem.

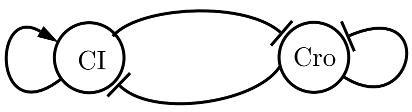
As the number of global states in an MVN is finite, that means that sooner or later a trace will visit a previously visited MVN state which results in a repeated set of states in the trace. The cycle of repeated states in an MVN is called an attractor [81]. There are two types of attractors: a point attractor, where the attractor has only one state (such as: 021 being constantly produced as a result of updating network entities), and a cycle attractor, where the attractor has more than one state, such as: $012 \rightarrow 111 \rightarrow 012$. The basin of an attractor [82] is the set of states that lead to that attractor. Source entities have no states leading to them and their only occurrence is at the start of a trace. The time an MVN takes to reach an attractor is referred to as transient time.

An MVN comprises a set of control entities each of which has a discrete state taken from a given set of states. The state of each entity is regulated by a subset of entities in the MVN and we refer to this subset as the neighbourhood of an entity (which may or may not be in its own neighbourhood). Each entity has a neighbourhood of entities which regulate its state (the entity itself may or may not play a part in its own regulation). An entity updates its state by applying a logical next–state function to the current states of the entities in its neighbourhood. A formal definition of an MVN can be given as follows.

Definition 1. An MVN MV is a four-tuple $MV = (G, D, N, F)$ where:
i) $G = \{g_1, \ldots, g_n\}$ is a non-empty, finite set of entities;
ii) $D = (D(g_1), \ldots, D(g_n))$ is a tuple of state sets, where each $D(g_i) = \{0, \ldots, m_i\}$, for some $m_i \geq 1$, is the state space for entity g_i;
iii) $N = (N(g_1), \ldots, N(g_n))$ is a tuple of neighbourhoods, such that $N(g_i) \subseteq G$ is the neighbourhood of g_i; and
iv) $F = (f_{g_1}, \ldots, f_{g_n})$ is a tuple of next-state multi-valued functions, such that if $N(g_i) = \{g_{i_1}, \ldots, g_{i_n}\}$ then the function $f_{g_i} : D(g_{i_1}) \times \cdots \times D(g_{i_n}) \rightarrow D(g_i)$ defines the next state of g_i. □

To illustrate the above definition consider the example *PL2* (see figure 2.2). *PL2* is a simple example of an MVN that models the core regulatory mechanism for the lysis-lysogeny switch [11, 12] in the bacteriophage [13]. for example transcripts and proteins The model consists of two entities CI and Cro. In this MVN, the entity *Cro* inhibits the expression of *CI* (i.e. acts to lower its state) and at higher levels of expression, also inhibits itself. Meanwhile, entity *CI* inhibits the expression of *Cro* and promotes its own expression (i.e. acts to increase its state).

Figure 2.2 models the regulatory network underlying the *lysis–lysogeny switch* in the bacteriophage λ [3, 25, 31]. This MVN consists of two entities: *CI* and *Cro*, defined with

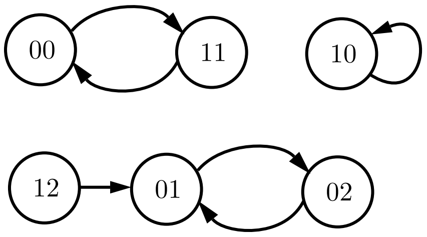


(a) Network structure

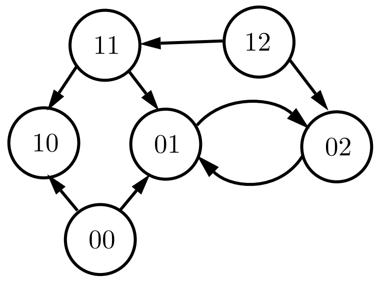
CI	*Cro*	*CI*
0	0	1
0	1	0
0	2	0
1	0	1
1	1	0
1	2	0

CI	*Cro*	*Cro*
0	0	1
0	1	2
0	2	1
1	0	0
1	1	0
1	2	1

(b) State transition tables



(c) Synchronous state graph



(d) Asynchronous state graph

Fig. 2.2 An MVN model *PL2* of the regulatory network for the lysis-lysogeny switch in bacteriophage λ (based on [25]).

neighbourhoods $N(CI) = \{CI, Cro\}$ and $N(Cro) = \{CI, Cro\}$. These entities have the state spaces $D(CI) = \{0, 1\}$ and $D(Cro) = \{0, 1, 2\}$, and their next-state functions are defined by the state transition tables given in Figure 2.2.(b).

In the sequel, let $MV = (G, D, N, F)$ be an arbitrary MVN. A *global state* of an MVN *MV* with n entities is represented by a tuple of states $(s_1, \ldots, s_n)$, where $s_i \in D(g_i)$ represents the state of entity g_i. The set of all global states, denoted S_{MV}, is then defined by $S_{MV} = D(g_1) \times \cdots \times D(g_n)$. As a notational convenience we normally write $s_1 \ldots s_n$ to represent a global state $(s_1, \ldots, s_n) \in S_{MV}$. When the current state of an MVN is clear from the context we let g_i denote both the name of an entity and its corresponding current state.

The global state of an MVN can be updated *synchronously* [6, 7], where the state of all entities is updated simultaneously in a single step, or *asynchronously* [32, 8], where entities update their state independently. Both update semantics are important and used widely in the literature; the synchronous semantics leads to simpler dynamic behaviour which aids analysis (thus historically it was favoured) while the asynchronous semantics is seen as providing a more realistic model since entities are allowed to react independently. Note that different variations of the asynchronous semantics have been considered in the literature (see for example [35]) but that we focus on the one most commonly used for MVNs.

Definition 2. (Synchronous Update) Given two states $S_1, S_2 \in S_{MV}$, we let $S_1 \xrightarrow{Syn} S_2$ represent a *synchronous update step* such that S_2 is the state that results from S_1 by simultaneously updating the state of each entity g_i, for $i = 1, \ldots, n$, using its next-state function f_{g_i} and the appropriate states from S_1 as indicated by the neighbourhood $N(g_i)$. □

Given a global state S_0 we can generate a *(synchronous) trace* by repeatedly applying the synchronous update step $S_0 \xrightarrow{Syn} S_1 \xrightarrow{Syn} S_2 \xrightarrow{Syn} \cdots$. Note in the sequel we will often represent such traces simply as a comma separated list of global states $S_0, S_1, S_2, \ldots$. Such traces are deterministic and so each possible initial state generates only a single trace. Since the set of global states is finite this means the set of synchronous traces is always finite. Each synchronous trace is itself infinite and will eventually enter an *attractor cycle* [33, 7]). For example, under the synchronous semantics *PL2* has three attractors: a point attractor $10, 10, \ldots$; and two cyclic attractors $00, 11, 00, \ldots$ and $01, 02, 01, \ldots$.

Definition 3. (Asynchronous Update) For any entity $g_i \in G$ in MV and any state $S \in S_{MV}$ we let $[S]^{g_i}$ denote the global state that results by updating the state of g_i in S using f_{g_i}. Define the global state function $next^{MV} : S_{MV} \to \mathscr{P}(S_{MV})$ on any state $S \in S_{MV}$ by

$$next^{MV}(S) = \{[S]^{g_i} \mid g_i \in G \; and \; [S]^{g_i} \neq S\}$$

Given $S_1 \in S_{MV}$ and $S_2 \in next^{MV}(S_1)$, we let $S_1 \xrightarrow{Asy} S_2$ denote an *asynchronous update step.*

□

Note that given the above definition, only asynchronous update steps that result in a change in the current global state are considered (see [8]). In the asynchronous case, traces are non-deterministic and can be finite or infinite. A single initial state can have an infinite number of possible asynchronous traces starting from it and thus in the asynchronous case there can be an infinite number of traces.

To illustrate the above, consider the global state 12 for *PL2* (see Figure 2.2) in which *CI* has state 1 and *Cro* has state 2. Then a single synchronous update step will result in the state 01. $12 \xrightarrow{Syn} 01$ is a single synchronous update step on this state resulting in the new state 01. The set of all synchronous traces for *PL2* is:

$00, 11, 00, \ldots$	$10, 10, \ldots$
$01, 02, 01, \ldots$	$11, 00, 11, \ldots$
$02, 01, 02, \ldots$	$12, 01, 02, 01, \ldots$

Applying the asynchronous update semantics, we have $next^{PL2}(12) = \{02, \; 11\}$ and so $12 \xrightarrow{Asy} 02$ and $12 \xrightarrow{Asy} 11$ are valid asynchronous update steps. The MVN *PL2* has the following (finite in this case) set of asynchronous traces:

$00, 01, 02, 01, \ldots$	10	$12, 02, 01, 02, \ldots$
$00, 10$	$11, 01, 02, 01, \ldots$	$12, 11, 01, 02, 01, \ldots$
$01, 02, 01, \ldots$	$11, 10$	$12, 11, 10$
$02, 01, 02 \ldots$		

When analysing the behaviour of an MVN it is important to consider its *attractors* [34, 35] since they can represent important biological phenomena.

The state transition behaviour of an MVN can be represented by a *state graph* [34] in which the nodes of the graph are the global states and the edges are precisely the update steps allowed. The synchronous and asynchronous state graphs for *PL2* are presented in Figure 2.2.

Any state $S \in S_{MV}$ which cannot be asynchronously updated, i.e. $next^{MV}(S) = \{\}$, is a *point attractor*. The notion of a simple cyclic attractor is not valid in the asynchronous case and instead we consider *terminal connected components* in an MVN's asynchronous state graph to be *attractors* [34, 35]. For example, the MVN *PL2* has one point attractor 10 and one cyclic attractor $01, 02, 01, \ldots$.

In the example presented we defined the next–state function for each entity using a state transition table. An alternative approach is to specify the next–state function equationally [2, 30] by using Boolean terms called *literals* to represent when an entity g_i is in one of its given states. The literals have the form g_iS, for any $S \subseteq D(g_i)$, and evaluate to true when $g_i \in S$ and to false otherwise. The literals can be combined using conjunction resulting in *product terms* which represent possible states for a collection of entities. These can then be used to construct equations to represent the next–state function for an entity.

As an example, consider the entity *CI* in the MVN *PL2*. From the state transition table in Figure 2.2.(b) we can see that *CI* will have next state 1 when we are in state $CI = 0$, $Cro = 0$ or when we are in state $CI = 1$, $Cro = 0$. Therefore, the product term $CI\{0,1\}Cro\{0\}$ specifies when *CI*'s next state will be 1. Using this approach we are able to completely specify equationally the next–state functions for the entities, as presented below:

$$\begin{array}{ll} CI\{0\} = CI\{0,1\}Cro\{1,2\} & Cro\{0\} = CI\{1\}Cro\{0,1\} \\ CI\{1\} = CI\{0,1\}Cro\{0\} & Cro\{1\} = CI\{0\}Cro\{0,2\} + CI\{1\}Cro\{2\} \\ & Cro\{2\} = CI\{0\}Cro\{1\} \end{array}$$

An important observation is that the Boolean terms derived above can normally be simplified using *multi-valued logic minimization techniques* [30] which has recently received increased interest due to a connection that has been made between it and the state assignment problem [64]. Briefly stated the idea of this connection is that the states of a finite state machine can be represented as the set of possible values for a single multiple-valued variable. We make use of this when modelling an MVN in the next section.

2.4 Rewriting Logic(RL)

Rewriting logic (RL) [9] is an algebraic specification framework which is capable of modelling and analysing the behaviour of dynamic, concurrent systems. RL has been successfully used to model a wide range of different formalisms and systems, such as process algebras [10, 11], Petri nets [12, 13], and biological systems [14, 15]. For a detailed introduction to RL we recommend [9, 17].

In RL the static states of a system are described using a standard equational specification. The dynamic behaviour of the system is then modelled using rewrite rules which are able to capture the non–deterministic state transitions that occur in such systems. RL also provides rewrite strategies which are able to control the application of rewrite rules and so allow an RL model to capture subtle aspects of the behaviour of a dynamic system.

As an example of using RL, consider modelling a simple dynamic system in which system states are multi–sets consisting of the symbols A , B, and C. The system's dynamic behaviour occurs by transforming symbols and can be summarised as follows:

• Symbol A can dynamically change to B.
• If symbol B is present then symbol C can change to A.
• Two occurrences of symbol B can be replaced by symbol C.

To model the above RL specification we introduce the sorts *Symbol* and $State$ to represent symbols and multi–sets of symbols. We define the sort *Symbol* to be a sub-sort of $State$, and so every symbol can be viewed as a singleton multi–set. Three constants are declared to represent the symbols: $A : State$, $B : State$, and $C : State$ in the system and an implicit multi-set union operator _ _ : $State\ State \to State$ (where _ is used to denote an infix argument location [17]) is declared. We define this operator by adding the following equations to be associative:

$A\ (B\ C) = (A\ B)\ C$

and commutative:

$A\ B = B\ A$

We then define a set of rewrite rules to capture the dynamic behaviour in the system:

$A \Rightarrow B$
$B\ C \Rightarrow B\ A$
$B\ B \Rightarrow C$

These rewrite rules are used to rewrite and navigate through system states. For example: let $A\ C$ be a multi–set representing the initial state of the system. Then the following rewrite trace represents one possible evolution of the system:

$A\ C \Rightarrow B\ C \Rightarrow B\ A \Rightarrow B\ B \Rightarrow C$

Starting with $A\ C$ the A gets rewritten to a B using the first rewrite rule giving us $B\ C$. Then, the $B\ C$ is rewritten to $B\ A$ using the second rule, $B\ A$ gets rewritten to $B\ B$ using the first rule again. Then finally the $B\ B$ gets rewritten to C where this trace terminates as there are no more possible rewrites.

An important motivation for using RL is the powerful support tools available (e.g. [17–19]). Examples of such tools include: *Maude* [17]; *Elan* [18]; and *Tom* [19]. We have chosen to use *Maude* [17] in this piece of work due to its range of analysis tools, such as a Linear Temporal Logic model checker [27], and meta–programming capabilities.

2.5 Maude

Maude[17, 28] is a high-performance reflective language supporting both equational and rewriting logic [41] specification and programming for a wide range of applications. Maude also supports logical reflection in a systematic way thus making Maude remarkably extensible and powerful. It allows many advanced metaprogramming and metalanguage applications. Some of the most interesting applications of Maude are metalanguage applications where it is used to create executable environments for different languages and models of computation [28].

Maude can be used to model check certain properties using the `search` command which allows us to explore a network using an initial state. The Maude LTL (Linear Temporal Logic) Logical Model Checker (LMC) [28] can be used to verify LTL properties in concurrent systems expressed as rewrite rules, using a pattern of initial states. Many applications in areas such as rewriting logic models of cellular biology [101] can be easily specified and model checked using the Maude model checker, while some basic properties in considerably smaller systems can be checked using the Maude `search` command.

2.5.1 Maude Specification

We illustrate using Maude by considering the simple RL example introduced in Section 2.4. Below is the formal description of this RL model in Maude:

```
mod EX1 is
sorts Symbol State .
subsort Symbol < State .

ops A B C : -> Symbol .
op __ : State State -> State [assoc comm].

rl [rule1] : A => B .
rl [rule2] : B C => B A .
rl [rule3] : B B => C .
endm
```

We define our model using the `mod` command and use `EX1` to represent the model name. We introduce the sorts `Symbol` and `State` and define sort `Symbol` as a sub-sort of `State`. Three constants are declared to represent the symbols `A`, `B`, and `C` in the system and an implicit multi-set union operator is declared:

```
 __ : State  State -> State
```

We define the union operator to be associative and commutative using appropriate flags within Maude:

```
 [assoc  comm]
```

Note that those represent the equations for those properties that were used in Section 2.4. We then define a set of rewrite rules to model the system's dynamic behaviour. These rules are labelled to allow their application to be traced and the definition of rewrite strategies.

2.5.2 Rewriting in Maude

Using our running model representation in Section 2.5.1, we can now use Maude to execute the rewrite system as follows:

```
Maude> rew[1] B C .
rewrite [1] in EXABC : B C .
rewrites: 1 in 0ms cpu (0ms real) (52631 rewrites/second)
result State: A B
```

The command `rew[1]` rewrites the system state `B C` one time and returns the resulting state `A B` (this is the order returned by Maude). We can search to termination in Maude by omitting the number of rewrites from the `rew` command. Using the system state `A C` from Section 2.4:

```
Maude> rew B C .
rewrite in EXABC : B C .
rewrites: 3 in 0ms cpu (0ms real) (130434 rewrites/second)
result Symbol: C
```

Maude rewrites the system state until no further rewrite rules can be applied and returns the resulting state `C` after performing 3 rewrites.

2.5.3 Searching in Maude

Maude provides a built–in model checking command `search S =>+ P`, which allows us to check if a pattern term `P` can be reached by rewriting an initial ground term `S`. The search command is defined in Maude as follows:

```
search {[ bound {,depth} ]} {in module :} subject searchtype
pattern {such that condition} .
```

We can use the `search` command to perform a search starting from state `A C` as follows:

```
Maude> search A C =>! s:State .
search in EXABC : A C =>! s:State .

Solution 1 (state 4)
s:State --> C
```

Notice the use of the symbol `!` in the search command (this specifies the type of search to be search to termination). Possible values for search type are:

`=>1` one step proof.
`=>+` one or more steps proof.
`=>*` zero or more steps proof.
`=>!` only canonical final states are allowed as solutions.

The optional bound argument provides an upper bound in the number of solutions to be found; if it is omitted, infinity is assumed. The optional depth argument indicates the maximum depth of the search. If it is omitted, infinity is assumed. It is also possible to give a depth bound without giving a bound on the number of solutions returned. The search type =>1 is an abbreviation of the search type =>+ with the depth bound set to 1. As usual, if the in clause is omitted, the current module is assumed [16].

The Search command can be used to check for network properties starting from a network state. We can use this search command to investigate whether we can reach a state containing `C C` from an initial state `A A B A A`:

```
search A A B A A =>+ C C s:State .
```

This search returns true, and we can view a corresponding witness rewrite trace. We can check that all four `A`'s are needed by executing the following search which returns false.

```
search A A B A =>+ C C s:State .
```

The Maude search command is helpful in exploring system states, for example: it can be used to check whether a certain state is reachable from a certain starting state as follows:

```
Maude> search A C =>+ C .
search in EXABC : A C =>+ C .
Solution 1 (state 4)
states: 5 rewrites: 4 in 0ms cpu (0ms real)
(34782 rewrites/second)
empty substitution
No more solutions.
states: 5 rewrites: 4 in 0ms cpu (0ms real)
(26490 rewrites/second)
```

This search is checking whether the state `C` is reachable starting from the state `A C` using the set of rewrite rules defined in the Maude file. Maude identified the state as reachable in 4 rewrites, the path that Maude took to come up with this result can be checked using the show path command:

```
Maude> show path 4 .
state 0, State: A C
===[ rl A => B [label rule1] . ]===>
state 1, State: B C
===[ rl B C => A B [label rule2] . ]===>
state 2, State: A B
===[ rl A => B [label rule1] . ]===>
state 3, State: B B
===[ rl B B => C [label rule3] . ]===>
state 4, Symbol: C
```

Maude shows the search path along with what rewrite rule was performed at every iteration until the resulting final state `C`.

2.5.4 Linear Temporal Logic (LTL) Model Checking

Model Checking [45, 63, 69] or property checking is a method used to automatically check whether a certain model of a system meets a given specification. Techniques of model checking are used to automatically verify correctness properties of finite-state systems. A simple model-checking problem is verifying whether a given structure is satisfying a given propositional logic formula. Checking models of hardware and software designs is also done using a class of model checking methods where the specification is given by a temporal logic formula [71]. Model checking is often applied to hardware designs, because for software, there is no decidability (there is no algorithmic procedure that can correctly decide whether some arbitrary mathematical propositions are true or false).[1]

Under appropriate conditions, mathematical models can be checked to check whether they satisfy some important properties, or in some cases, obtain a useful counterexample showing that such property is violated. LTL (Linear Temporal Logic) [68] or linear-time temporal logic is a modal temporal logic referring to time. Using LTL, future of certain paths

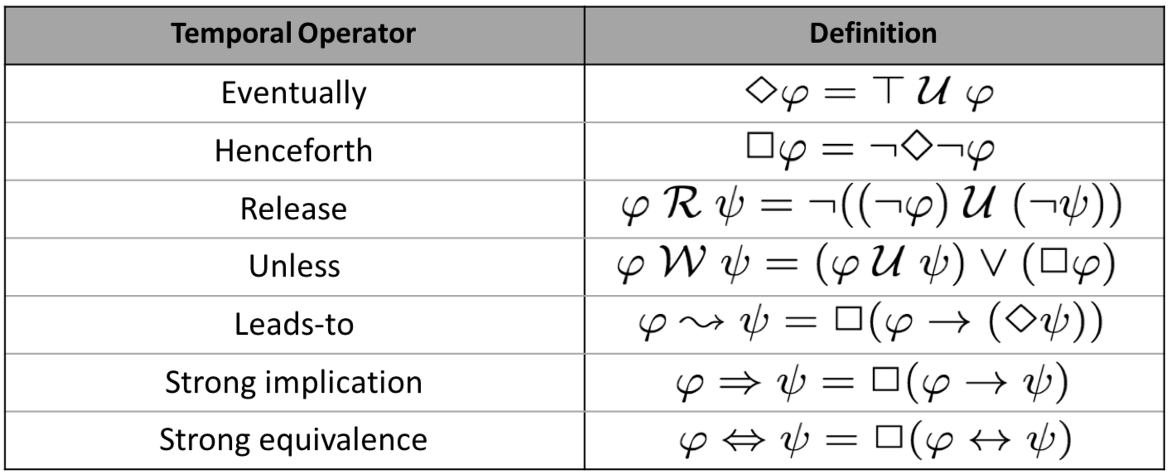
[1] Entscheidungsproblem [73], German for decision problem is a challenge proposed by David Hilbert in 1928 where it has been proven that it is not effectively decidable.

can be encoded using formulas, e.g., a condition will eventually be false, a condition will be true until another fact becomes false, etc. The Maude module model-checker.maude has a key operator *modelCheck* that takes an initial state and an LTL formula and returns either the Boolean true if the formula is satisfied, or a counterexample when it is not satisfied. Current work involves using LTL formulas to check for properties MVN's (both synchronous and asynchronous) and what sort of properties can only be proved using LTL model checking and not the Maude's search command. While any LTL property of a system can be model checked when the system is specified in Maude as a system module, Search allows for a simpler, yet very useful model checking capabilities, such as model checking of invariants, which can be accomplished just by using the Maude search command. Model checking using Maude's search command was the first approach taken to model-check the models and case studies mentioned in this document. Invariants are the most common and useful safety properties, where the property checked is stating that something bad should never happen. An invariant is a predicate defining a set of states that contains all states reachable from a starting state *s0*, if an invariant holds then it is guaranteed that something "bad" (a certain property) can never happen.

The Maude manual provides the following introduction to LTL formulas that given a set *AP* of atomic propositions, we define the formulas of the propositional linear temporal logic $LTL(AP)$ inductively as follows:

• **True**: $T\ \varepsilon\ LTL(AP)$.
• **Atomic propositions**: If $p\ \varepsilon\ AP$, then $p\ \varepsilon\ LTL(AP)$.
• **Next operator**: If $\wp\ \varepsilon\ LTL(AP)$, then $\bigcirc\ p\ \varepsilon\ LTL(AP)$.
• **Until operator**: If $p,\psi\ \varepsilon\ LTL(AP)$, then $p\ \upsilon\ \psi\ \varepsilon\ LTL(AP)$.
• **Boolean connectives**: If $p,\psi\ \varepsilon\ LTL(AP)$, then the formulas $\neg\ p$, and $p\ \bigvee\ \psi$ are in $LTL(AP)$.

Other LTL connectives can be defined in terms of the above minimal set of connectives as shown in Figure 2.3:



Temporal Operator	Definition
Eventually	$\Diamond\varphi = \top\,\mathcal{U}\,\varphi$
Henceforth	$\Box\varphi = \neg\Diamond\neg\varphi$
Release	$\varphi\,\mathcal{R}\,\psi = \neg((\neg\varphi)\,\mathcal{U}\,(\neg\psi))$
Unless	$\varphi\,\mathcal{W}\,\psi = (\varphi\,\mathcal{U}\,\psi) \vee (\Box\varphi)$
Leads-to	$\varphi \rightsquigarrow \psi = \Box(\varphi \rightarrow (\Diamond\psi))$
Strong implication	$\varphi \Rightarrow \psi = \Box(\varphi \rightarrow \psi)$
Strong equivalence	$\varphi \Leftrightarrow \psi = \Box(\varphi \leftrightarrow \psi)$

Fig. 2.3 Temporal Operators in LTL.

To use the model checker we define a range of atomic propositions to represent the key properties of interest. For example, for our running example introduced in 2.4 we define a proposition `hasA : -> Prop` , where `hasA(s)` is true only if an `A` is present. This can be done equationally as follows (where `[owise]` is a Maude short-cut which allows all remaining possibilities to be covered):

```
eq (A s:State) |= hasA = true .
eq (s:State) |= hasA = false [owise] .
```

We can then model check a range of interesting dynamic properties expressed using the temporal operators of LTL as illustrated by the example below for `EX1`(where `init1 = A B B`):

```
Maude> red modelCheck(init1, <> none(A)) .
reduce in EXABC-CHECK : modelCheck(init1, <> none(A)) .
rewrites: 12 in 0ms cpu (0ms real) (35398 rewrites/second)
result [ModelCheckResult]: counterexample({A B B,'rule3}
{A C,'rule1} {B C,'rule2} {A B,'rule1} {B B,'rule3},
{C,deadlock})
```

This example shows that starting from state `A B B`, eventually no `A's` will be present and the model checking command results in true.

2.5.5 Rewriting Strategies

With our running example introduced earlier in Section 2.5.1. Suppose we start from the system state `BBC` in Figure 2.4 we can either use `rule3` which will result in state `CC`, or `rule2` which will result in a different state `BBA` this shows the non-deterministic behaviour of our running example. Suppose we want to prioritise the application of rule 1 over the other two rules, we can do that with the help of rewriting strategies.

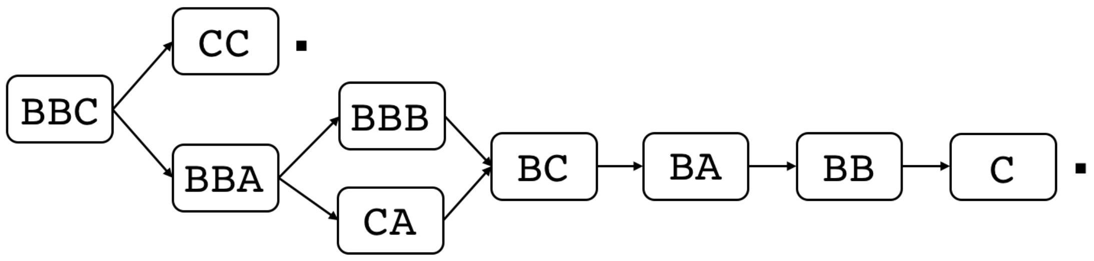


Fig. 2.4 The Non-Deterministic Behaviour of the Dynamic System.

The meta–programming capabilities offered by Maude are invaluable as they allow the construction of rewriting strategies which can control how rewrite rules are applied. As an example, suppose we want to prioritise the application of `rule3` over the other two rules. Then we can construct a corresponding rewrite strategy `threeFirst` in Maude to do this as shown below (Figure 2.5 shows the effect of prioritising `rule3` over the other two rules.):

```
ceq threeFirst(T) = if Step? :: Result4Tuple then getTerm(Step?)
else (if Step2? :: ResultPair then getTerm(Step4?) else T  fi)  fi
if Step? := metaXapply(upModule('EXABC,false),T,'rule1,none,0,unbounded
,0) /\  Step2? := metaRewrite(upModule('EXABC,false),T,1) .
```

Using the `red` command to apply this strategy on the term `B B C C` prioritizes `rule3` now instead of applying `rule2` as it would without this strategy in place:

```
Maude> red threeFirst( '__['B.Symbol,'B.Symbol,'C.Symbol,'C.Symbol] ) .
reduce in CheckABC : threeFirst('__['B.Symbol,'B.Symbol,'C.Symbol,
'C.Symbol]) .
rewrites: 10 in 0ms cpu (0ms real) (40650 rewrites/second)
result GroundTerm: '__['C.Symbol,'C.Symbol,'C.Symbol]
```

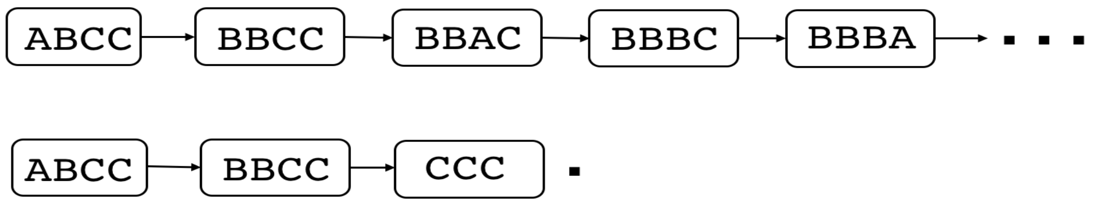


Fig. 2.5 The effect of prioritising rule3.

Prioritising `rule3` resulted in a much shorter trace reaching termination after two rewrite steps. Implementing this strategy has allowed us to control how rewrite rules are applied within our model resulting in a different trace. This strategy is implemented using a conditional equation and makes use of two metalevel operations: `metaXapply` which allows a specific rule to be applied (in this case `rule3`) to a term `T` ; and `metaRewrite` which allows a term `T` to be rewritten a given number of times using all rules in a module. Note the use of type checking to see whether the meta-level operations have been successfully applied, e.g. `Step? :: Result4Tuple` is used to check if `rule3` has been successfully applied. For full details of the notation used here and further strategy examples see the Maude manual [28].

2.6 Existing Work on Supporting MVNs

2.6.1 Petri Nets

A *Petri net* [46], also known as a place/transition (PT) net, is one of several mathematical modelling languages for the description of distributed systems whcih can be used to model MVNs. The theory of Petri nets (PNs) combines a formal mathematical semantics with a graphical notation for modelling and reasoning about complex concurrent systems [46]. A Petri net is a directed bipartite graph and consists of four basic components: places, which are denoted by circles; transitions denoted by black rectangles; arcs denoted by arrows; and tokens denoted by black dots.

PNs have been applied to many problem domains including system verification [47, 48], hardware design [49], medicine [50] and biological systems [51, 53]. Specifically, PNs offer a number of advantages for modelling biological systems, such as their ability to capture both the static structure and dynamical behaviour of a system in a concise way, and the wealth of formal techniques and tools for simulation and analysis that they have [54].

The state of a Petri net is given by its marking, which is the distribution of tokens on places within it, therefore the state space of a Petri net is the set of all possible markings. The dynamics of a Petri net are modelled by transitions which can fire to move tokens between places in a Petri net. Transitions are only enabled when each of their input places contain at least one token. An enabled transition can fire by consuming one token from each of its input places, then depositing one token on each of its output places. For example: in Figure 2.6 both transitions t_1 and t_2 are enabled. Firing transition t_2 would result in a token being taken from place p_2 and a new token being deposited in place p_4.

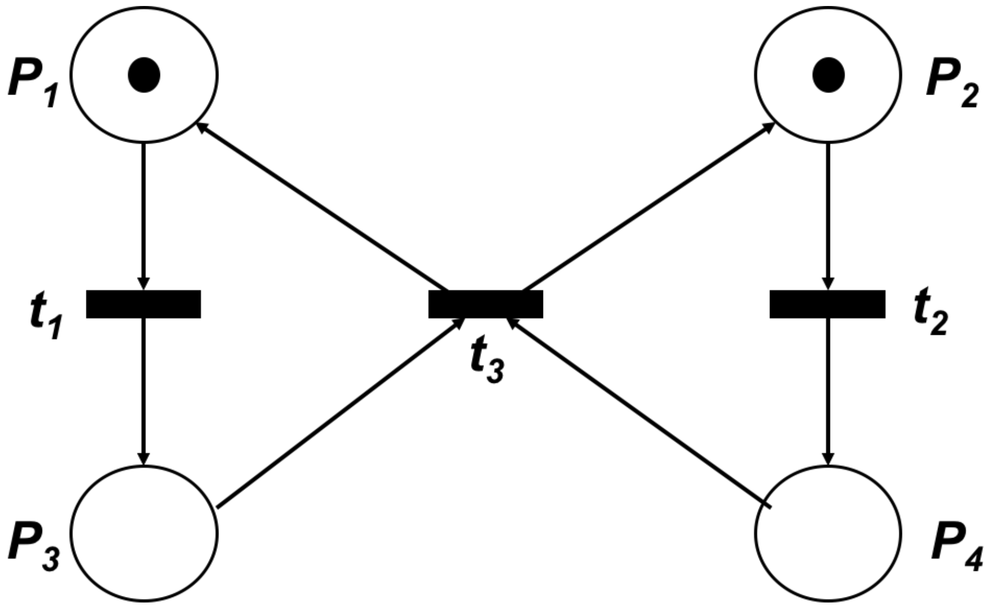


Fig. 2.6 A Simple Example of a Petri net [61].

An important advantage of Petri nets is the wide range of techniques and tools available for their simulation and analysis. For example: a Petri net can be analysed by constructing its reachability graph [56] which captures the possible firing sequences that can occur from a given initial marking. Petri nets can be automatically cheeked for boundedness and the presence of deadlocks [55]. Petri nets have been used to represent regulatory networks [77, 76] , more specifically high level Petri nets which has a unique transition and as many places as genes in the regulatory graph [58] where each place corresponds to a gene in the regulatory network and carries one token. Petri nets provide a natural alternative framework for modelling MVNs, one notable tool framework is GINsim [59] which we introduce in 2.6.2.

2.6.2 GINsim

GINsim [59, 84] (Gene Interaction Network simulation) is a Java software for the modelling and simulation of genetic regulatory networks. GINsim also provides an impressive array of techniques for modelling and analysing asynchronous MVN models. It consists of a simulator of qualitative models of genetic regulatory networks based on a discrete, logical formalism. GINsim allows the user to specify a model of a genetic regulatory network in term of asynchronous, multivalued logical functions, and to simulate and/or analyse its qualitative dynamical behaviour [43]. GINsim is based on public graph libraries, JGraph and JGraphT, and is designed in a modular way to ease extensions and collaborative developments [59].

Models in GINsim are stored in a specific XML file format, GINML, but can also be exported into various formats, including PNG (image), SVG, Graphviz and BioLayout (graphs), as well as Cytoscape (xgmml) formats. Thanks to the many plugins available for Cytoscape (e.g. BiNoM, which handles CellDesigner, BioPAX and SBML formats [38]), regulatory and state transition graphs defined with GINsim can now be converted into virtually any relevant format, thereby enabling the combination of various software tools with complementary functionalities.

GINsim encompasses four main modules: a user interface, a parser, a core simulator and a graph analysis toolbox. Implementation of new modules in GINsim can be done using GINsim's plug-in based architecture, in particular to complete the graph analysis toolbox [59].

As in example of using GINsim to model a regulatory network. We model the process of controlling the lysis-lysogeny decision in the phage lambda that is based on the model PL2 introduced earlier in Section 2.3.

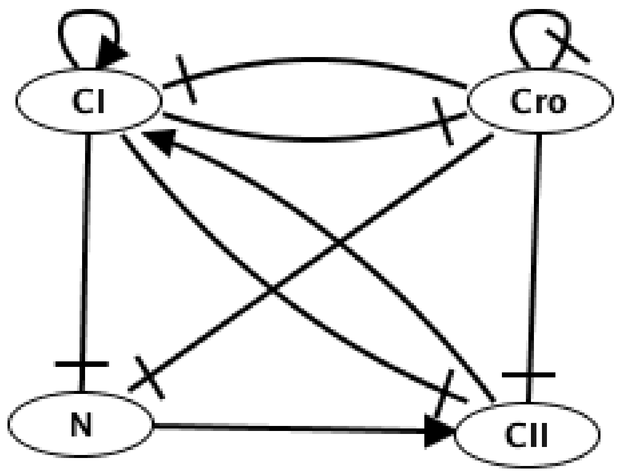


Fig. 2.7 A Simple Example of Modelling an MVN in GINsim [60].

Figure 2.7 gives an example of modelling a proposed four entities model encompassing the roles of the regulatory genes *CII* and *N* in addition to *CI* and *Cro*. This model is introduced in further detail and analysed in Section 4.4.1.

2.6.3 Other Support Tools

BooleSim [103] is a web-based easy-to-use Boolean network simulator. It is a standalone tool that only requires an HTML5-capable web browser to run. BooleSim is coded in JavaScript, which makes it platform-independent, and it's available under the software license: GNU Affero GPL v3. The tool can be either used on-line or off-line by installing the off-line version of the tool. BooleSim uses several other projects such as: biographer UI [104], which is a biographer simulator, libSBGN.js [105], a graphical notation that is aims to standardize the graphical notation used in maps of biological processes, D3 (Data-Driven Documents) [106] which is a JavaScript library that can be used to produce dynamic, interactive data visualizations in web browsers, and Rickshaw, which is also a JavaScript toolkit that can be used to produce interactive real-time graphs. The tool provides a user-friendly interface that allows the user to load Boolean networks in various formats; the user can then display and interactively simulate those networks.

The tool can be used by either starting a new Boolean network from scratch, or by importing an existing network. The tool website provides two demos: Pluripotency demo and Cell cycle demo. The tool provides an easy-to-use Boolean network simulator, but it doesn't produce either traces or state graphs. This tool also does not cover either clustering or basin of attraction.

BNS [13] is a tool for computing attractors in Boolean networks with synchronous update. Synchronous Boolean networks [24, 44, 83] are commonly used in modelling genetic regularity networks. BNS takes a .cnet file format that contains a Boolean network description. The tool will then print out the network's attractors. BNS binaries are available for Linux and Windows Cygwin with version 1.0, and have been tested on both platforms according to the tool website. The tool website provides a set of test input files with different number of attractors, the format used in those test input files is explained in the user manual available on the tool's website. The tool can find attractors in synchronous Boolean networks, but it is limited to that functionality, it does not produce any traces, state graphs or a wiring diagram. This tool also does not cover neither clustering nor basin of attraction.

2.7 Related work on RL

Pathway Logic (PL) [36, 37], an existing RL framework for symbolically modelling and analysing signal transduction and metabolic pathways. It represents each biomolecule involved in a biological pathway using a term containing three components: the name of the biomolecule; a modifier which indicates the state of the biomolecule (such as whether it is phoshorylatd or not); and the location of the biomolecule. Using the standard RL approach, the state of a cellular pathway is represented as a multi-set of biomolecule terms. Rewrite rules are used to capture the local changes that occur to biomolecules during the signal transduction steps and these rules then form the so called Rule Knowledge Base which specifies how a pathway can be traversed. The resulting pathway model can then be executed and analysed in Maude using the tool's model checking capabilities. A range of work has been done to develop interesting techniques and tools for PL (see [38] for an overview) and the end result is a powerful framework for engineering biological pathways.

PL is used as an approach to model biological processes as executable formal specifications in Maude where the resulting models can be queried using formal methods tools. The queries can aim to find some pathway given an initial state, find all reachable states meeting a given property and model-check by finding a pathway that satisfies a temporal formula. PL have been applied to pathways regulating a wide range of biological processes such as the epidermal growth factor receptor (EGFR) pathway [62]. Figure 2.8 shows a fragment of the mammalian EGFR system that illustrates the activation of a downstream signalling pathway and a potential mechanism for cross-communication between the EGFR and a G protein-coupled receptor (AT1) which is adapted from [62].

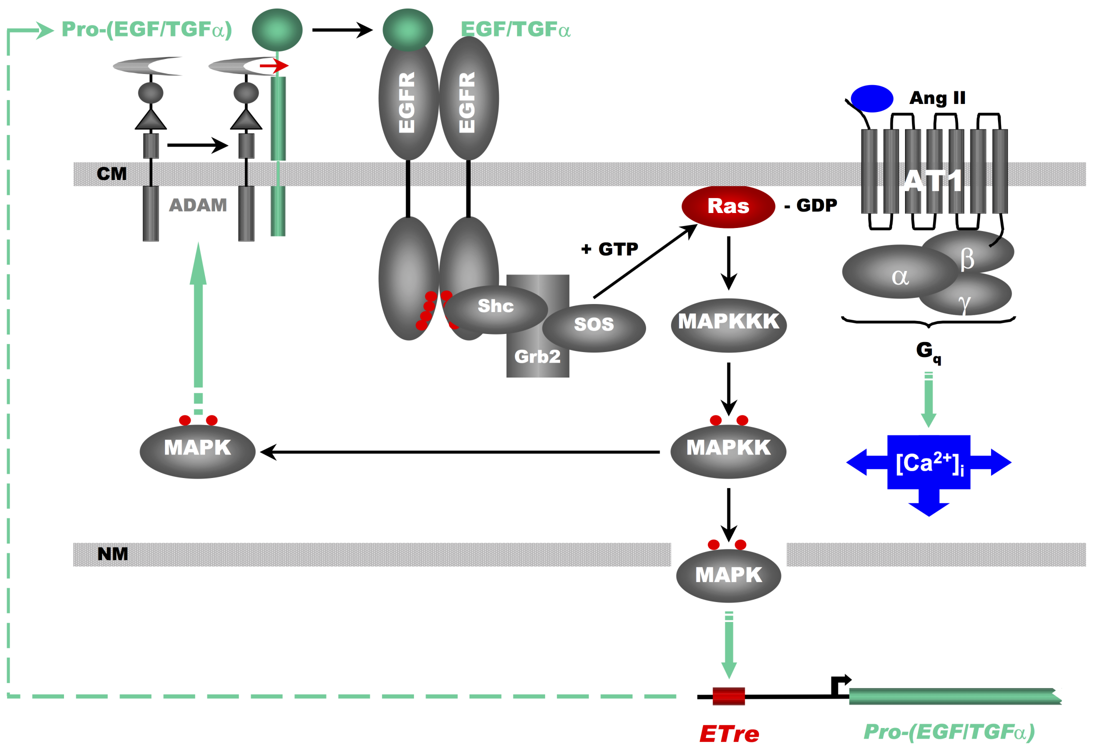


Fig. 2.8 A Fragment of the Mammalian EGFR System. Taken from [64].

The algebraic structure of PL can be expressed using Maude [28] to model the EGFR pathway logic where we can define sorts to represent amino acids and proteins to model the behaviour of the model.

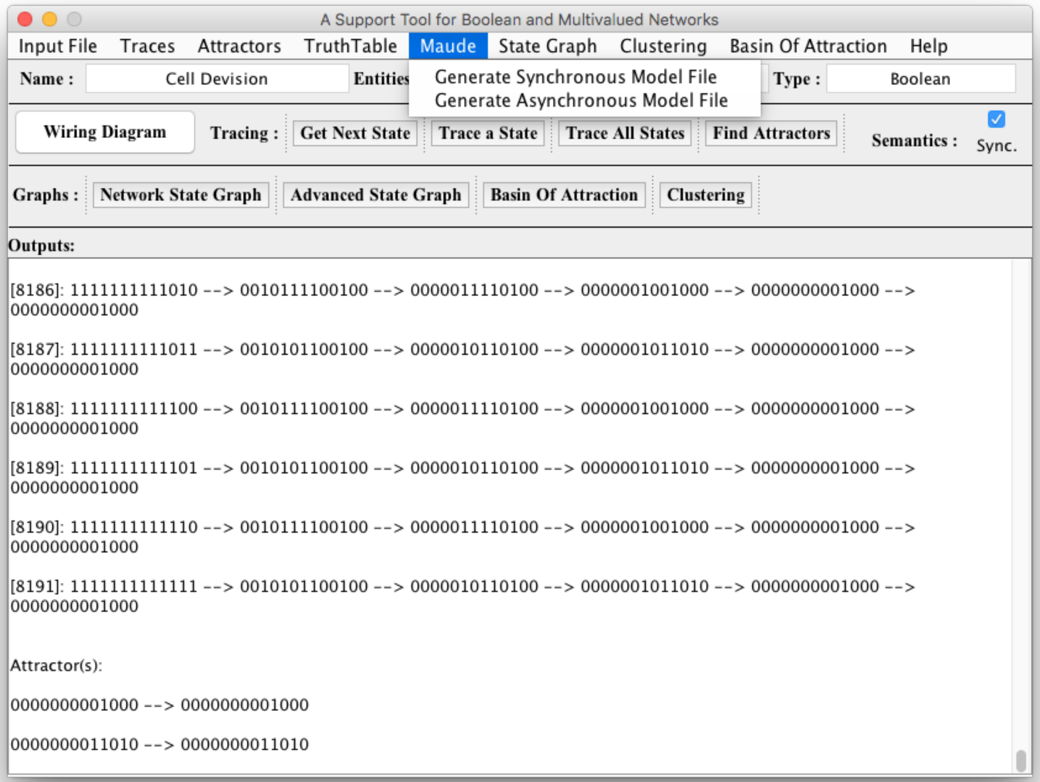
2.8 Tool Support

A tool support was developed over the course of this thesis using Java to aid in analysing MVNs and constructing RL models. The tool builds on what the author worked on during his Masters' dissertation where the focus was on building a basic support tool for Boolean networks. As a start, the tool was extended to support MVNs by reading an XML description of a network using its entities' truth tables. The tool would then allow the user to perform traces, identify attractors and produce state graphs of the MVN semantics. The tool was then extended to support RL by generating RL model files to enable the use of the powerful and wide range of RL applications such as Maude. The tool was used over the course of this thesis to perform a range of tasks to help model and analyse a range of MVN models using Maude and RL. Figure 2.9 shows a screen-shot of the tool in action.

Graphs were generated using the Java Universal Network/Graph Framework (JUNG) [57] which is a software library written in Java that provides an extendible language for analysing and visualising any type of data that can be represented as a network of entities or a graph.

JUNG was chosen because of its powerful yet easy to use visualization framework, which provides a range of layout algorithms for graphs to suit different types of networks. Filtering in JUNG has helped in making the generated graphs using the developed tool support a lot more meaningful and has allowed us to focus on specific areas or portions of the generated state graphs. This makes it easy to construct tools for the interactive exploration of network data. Users can use one of the layout algorithms provided, or use the framework to create their own custom layouts. In addition, filtering mechanisms are provided which allow users to focus their attention, or their algorithms, on specific portions of the graph.

The aim here is to produce a support toolset for MVNs using RL that can read in an MVN structure and produce traces and attractors that can help identify a certain behaviour.



```
[8186]: 1111111111010 --> 0010111100100 --> 0000011110100 --> 0000001001000 --> 0000000001000 -->
0000000001000

[8187]: 1111111111011 --> 0010101100100 --> 0000010110100 --> 0000001011010 --> 0000000001000 -->
0000000001000

[8188]: 1111111111100 --> 0010111100100 --> 0000011110100 --> 0000001001000 --> 0000000001000 -->
0000000001000

[8189]: 1111111111101 --> 0010101100100 --> 0000010110100 --> 0000001011010 --> 0000000001000 -->
0000000001000

[8190]: 1111111111110 --> 0010111100100 --> 0000011110100 --> 0000001001000 --> 0000000001000 -->
0000000001000

[8191]: 1111111111111 --> 0010101100100 --> 0000010110100 --> 0000001011010 --> 0000000001000 -->
0000000001000

Attractor(s):

0000000001000 --> 0000000001000

0000000011010 --> 0000000011010
```

Fig. 2.9 A Support Tool for Boolean and Multi-Valued Networks.

The tool set will also be able to produce different types of graphs such as a basin of attraction or a clustered version of the state graph. RL models representing the behaviour of an MVN can also be generated to enable the use of the powerful tool support available for RL.

2.9 Conclusions

In this chapter, we introduced MVNs which are mathematical models of control systems where time and states are discrete. MVNs extend the well-known Boolean networks[1,2,3] by providing a more powerful qualitative modelling approach for biological systems; they extend Boolean networks by allowing an entity's state to be within a range of discrete set of values instead of just 0 or 1. We gave a formal definition to what an MVN is and introduced an MVN model as an example. We gave formal definitions to what both synchronous and asynchronous updating mechanisms mean.

After that, we introduced Rewriting Logic which has been successfully used to model a wide range of different formalisms and systems, such as process algebras [10, 11], Petri nets [12, 13], and biological systems [14, 15]. After introducing RL we then introduced Maude, which is a high-performance reflective language supporting both equational and rewriting logic [41, 70, 28] specification and programming for a wide range of applications.

We introduced the syntax of Maude and give some examples using models that were introduced in Section 2.3. We provided some examples of using Maude's search and rewrite commands to explore certain properties of a model using our running examples. We extended our analysis by using Maude's built in Linear Temporal Logic (LTL) model checker and gave some examples of some uses of LTL.

We introduced rewriting strategies which can control how rewrite rules are applied within a model using Maude's meta-programming capabilities allowing the construction of rewriting strategies. A discussion of related work and existing support tools was provided in Section 2.6.

In the next chapter we develop an RL model for asynchronous MVNs based on their asynchronous update semantics. We present a translation approach that translates an MVN into an RL model and we provide a formal correctness argument for our approach. We present a detailed case study using techniques and the developed RL framework alongside the resulting analysis made possible using those tools and techniques.

Chapter 3

An RL Model for Asynchronous MVNs

3.1 Introduction

While a range of tools and techniques for developing and analysing MVNs exists in the literature (see Section 2.6), more work is needed to develop tool support to help with the practical applications of those techniques. In this chapter we set out to strengthen the tool support available for MVNs by linking them to RL and so enabling the application of the wide range of support tools available for RL.

We start by constructing an RL model for MVNs based on their asynchronous update semantics, we derive a set of rewrite rules that are used to capture an MVN's behaviour. We use Maude to introduce a module file representing the model, we represent within our model the next state function of network entities and the associated asynchronous update rules. We use a translation approach that takes in a structured MVN file and produce an RL Model with required definitions and set of rewrite rules.

We formally show that our translation process from an asynchronous MVN to an RL model is correct, that is done by showing and proving the soundness and completeness of our translation process. To do this we show that: 1) (*soundness*) each global state transition possible in our RL model represents a corresponding asynchronous update in the original MVN; and 2) (*completeness*) every asynchronous update possible in an MVN is specified in our RL model.

Techniques and the developed RL framework are illustrated alongside the resulting analysis possible by presenting a published case study from the literature that is based on modelling and analysing the genetic regulatory network for the synthesis of Tryptophan in *E. coli*[21, 22]. This case study helps to motivate our RL framework by illustrating the analysis techniques and tools available when using RL and the tool *Maude*. We formally investigate the model using a range of powerful analysis tools provided by Maude, which was one of the motivations for choosing to use RL.

This chapter is organised as follows: in Section 3.2, we construct an RL model for an asynchronous MVN, and we derive a set of rewrite rules using a translation approach that enables us to produce a formal representation of an MVN using RL. In Section 3.3 we prove the correctness of our translation approach by showing its soundness and completeness. In Section 3.4, we illustrate our RL framework by presenting a detailed case study based on modelling and analysing the genetic regulatory network for the synthesis of Tryptophan in E. coli. Finally in Section 3.5, we conclude the work presented in this chapter with concluding remarks.

3.2 Constructing an RL Model for an Asynchronous MVN

Let $MV = (G,D,N,F)$ be an MVN and assume that we are using the asynchronous update semantics. Then we construct a corresponding RL model $RL_A(MV)$ as follows.

For each discrete state space $D(g) = \{0,\ldots,m\}$ associated with some entity $g \in G$ define a corresponding sort D_m in $RL_A(MV)$ with constants $i : D_m$, for $i = 0,\ldots,m$. We define the sort *Entity* in $RL_A(MV)$ for entities and represent each entity $g \in G$ in our MVN by a function $g : D_m \to \mathit{Entity}$, where $D(g) = \{0,\ldots,m\}$. We represent global states in an MVN as non-empty multi–sets of entities. To do this we introduce the sort *GState*, defining *Entity* to be a sub sort of *GState*, and then use the normal approach of adding an implicit union operator (see the example in Section 2.3). Note that given the above definitions not all terms of sort *GState* will correspond to well–defined global states of MV (they may omit an entity or contain multiple terms for an entity). For simplicity, we avoid restricting the generation of global state terms and instead introduce appropriate restrictions as part of our correctness argument (see Section 3.3 below).

As an example, consider the following signature generated for our running example *PL2* introduced earlier in section 2.3 (see Figure 2.2) presented using the Maude syntax [17]:

```
mod PL2 is
sorts Entity GState .
sorts D1 D2 .
subsort Entity < GState .

ops 0 1 : -> D1 .
ops 0 1 2 : -> D2 .
op CI : D1 -> Entity .
op Cro : D2 -> Entity .
op __ : GState GState -> GState [assoc comm].
endm
```

We define our model using the `mod` command and use `PL2` to represent the model name, sorts `Entity` and `GState` are defined and sort `Entity` is defined as a sub-sort of `GState` (meaning that a combination of entities can be used to form a `GState`). Two constants are declared to represent entities `CI` and `Cr0` in the model and an implicit union operator `__ : GState GState` $\rightarrow$ `GState` is declared which we define to be associative and commutative using the flags ([*assoc comm*]) in Maude. Entity `CI` has a state space that is represented using sort $D1 = \{0,1\}$, while `Cr0` has a state space of $D2 = \{0,1,2\}$. Given the above definitions a global state in which *CI* has state 1 and *Cro* has state 2 would be represented by the term `CI(1) Cro(2)`.

The final stage in modelling an MVN is to represent within our RL model the next–state function associated with each entity and the associated asynchronous update rule. Note that in doing this we will implicitly capture the neighbourhood information associated with the MVN. In order to do this we begin by deriving a equational representation for the next–state functions as described in Section 2.3. We then simplify the resulting equations using *multi–*

valued logic minimization techniques [30] (note that this process can be automated using tools such as *MVSIS* [36]). Each of the simplified equations will have the general form

$$g\{s\} = P_1 + \cdots + P_k$$

for some $k > 0$, $g \in G$ and $s \in D(g)$. The equation specifies that g will have next state s precisely when one or more of the product terms $P_1, \ldots, P_k$ is true. We generate the rewrite rules for asynchronously updating g to state s by deriving a set of rewrite rules for each product term P_i, $i = 1, \ldots, k$, as follows.

First we need to ensure that a term representing the current state of g occurs within P_i. To do this we check P_i and if no term of the form gS' is present, for some $S' \subseteq D(g)$, then we use conjunction to add the term $g(D(g) - \{s\})$ to P_i (in a slight abuse of notation we let P_i still denote this new term). Now suppose

$$P_i = g_{E_1}S_1 \ \ldots \ g_{E_m}S_m \ gS_{m+1}$$

for some $m \geq 0$, and state sets $S_j \subseteq D(g_{E_j})$, $1 \leq j \leq m$ and $S_{m+1} \subseteq D(g)$. Then we add the following rewrite rules to $RL_A(MV)$:

$$g_{E_1}(u_1) \ \ldots \ g_{E_m}(u_m) \ g(u_{m+1}) \Longrightarrow g_{E_1}(u_1) \ \ldots \ g_{E_m}(u_m) \ \ g(s)$$

for each $u_j \in S_j$, $1 \leq j \leq m+1$, such that $u_{m+1} \neq s$. Note the final restriction $u_{m+1} \neq s$ is needed here to ensure that only state transitions which change the current state of entity g are allowed (this is a fundamental part of the asynchronous update semantics).

To illustrate the above process consider applying it to our running example *PL2*. First we derive the equations for *PL2* (see Section 2.3). We then apply multi–valued logic minimization to simplify these to the following set of equations:

1) $CI\{0\} = Cro\{1,2\}$
2) $CI\{1\} = Cro\{0\}$

3) $Cro\{0\} = CI\{1\}Cro\{0,1\}$
4) $Cro\{1\} = CI\{0\}Cro\{0\} + Cro\{2\}$
5) $Cro\{2\} = CI\{0\}Cro\{1\}$

The final stage is to use the simplified equations to derive a set of rewrite rules to model the asynchronous update of *PL2*. Figure 3.1 describes the translation process starting from simplified equations representing entities behaviour and ending up with the matching Maude rewrite rule.

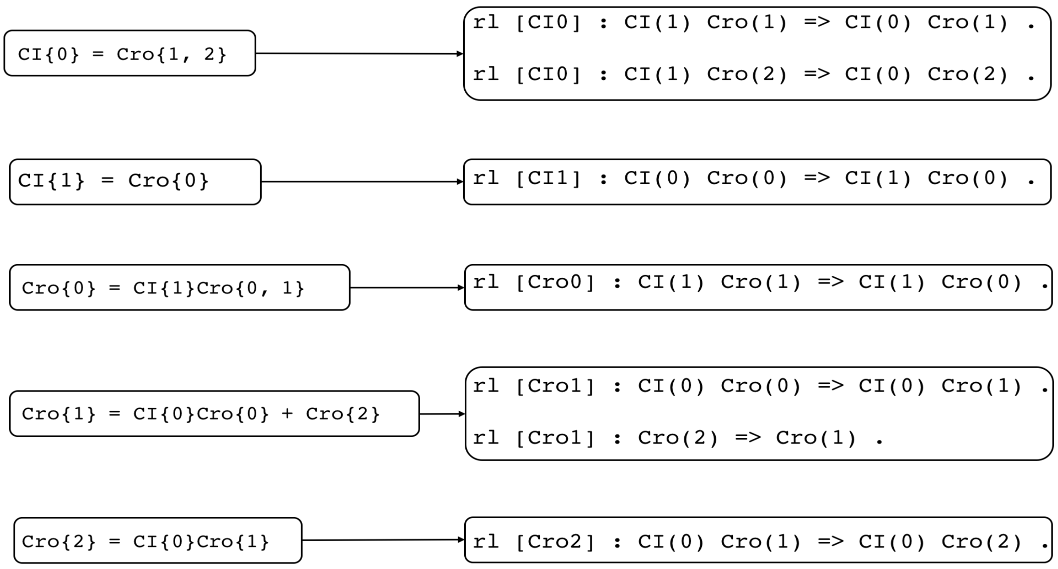

CI{0} = Cro{1, 2}
rl [CI0] : CI(1) Cro(1) => CI(0) Cro(1) .
rl [CI0] : CI(1) Cro(2) => CI(0) Cro(2) .
CI{1} = Cro{0}
rl [CI1] : CI(0) Cro(0) => CI(1) Cro(0) .
Cro{0} = CI{1}Cro{0, 1}
rl [Cro0] : CI(1) Cro(1) => CI(1) Cro(0) .
Cro{1} = CI{0}Cro{0} + Cro{2}
rl [Cro1] : CI(0) Cro(0) => CI(0) Cro(1) .
rl [Cro1] : Cro(2) => Cro(1) .
Cro{2} = CI{0}Cro{1}
rl [Cro2] : CI(0) Cro(1) => CI(0) Cro(2) .

Fig. 3.1 The Translation Process, Equations to Rewrite Rules.

To illustrate the approach defined above consider deriving the rewrite rules for *Cro* entering state 0. From the equations above we see that *Cro* has next state 0 whenever *CI* is in state 1 and *Cro* is in state 0 or 1. However, a state change occurs here only if *Cro* is currently in state 1 so we need only one rewrite rule to model this state transition:

$$CI(1)\ Cro(1) \Longrightarrow CI(1)\ Cro(0)$$

In equation number 1, *CI* has the value of 0 when the value of *Cro* is either 1 or 2. With *Cro* having two possible values here there is a need for two rewrite rules to cover both values. Also there is a need to add the value of $CI\{1\}$ to represent the change in behaviour for *CI* going from state 1 to state 0. Equation number 2 is straightforward and requires only a single rewrite rule to model it that covers the case of $Cro\{0\}$ and *CI* going from state 0 to state 1. In equation number 3, *Cro* has two values here but only one of them represents a change in its value, so $Cro\{0\}$ is not needed in the rewrite rule since there is no point of rewriting $Cro\{0\}$ as $Cro\{0\}$ again. We have an equation of two parts that are combined by disjunction in equation number 4 which means that we need two rewrite rules to represent the two parts of the equation. Finally, equation number 5 is straightforward and the rewrite rule is a simple rewrite of $Cro\{1\}$ to $Cro\{2\}$.

The full set of rules derived to model the asynchronous next–state functions for *PL2* is given below:

```
rl [CI0] : CI(1) Cro(1) => CI(0) Cro(1) .
rl [CI0] : CI(1) Cro(2) => CI(0) Cro(2) .
rl [CI1] : CI(0) Cro(0) => CI(1) Cro(0) .

rl [Cro0] : CI(1) Cro(1) => CI(1) Cro(0) .
rl [Cro1] : CI(0) Cro(0) => CI(0) Cro(1) .
rl [Cro1] : Cro(2) => Cro(1) .
rl [Cro2] : CI(0) Cro(1) => CI(0) Cro(2) .
```

The tool support was extended at this point to automatically apply the above translation process given an XML description of an MVN. Equations and Rewrite rules for the module are automatically generated using tool support and are stored in text files in the same format Maude uses for rewrite rules. The tool also produces basic Maude files with needed variables and rewrite rules to represent a network.

3.3 RL Model Correctness

In this section we formally show that our translation from an asynchronous MVN to an RL model given in the previous section is correct. To do this we show that: 1) (*soundness*) each global state transition possible in our RL model represents a corresponding asynchronous update in the original MVN; and 2) (*completeness*) every asynchronous update possible in an MVN is specified in our RL model.

We begin by precisely defining which terms in our RL model represent well–defined global states. Note that not all RL terms of sort $GState$ represent represent valid global states. For example, in our running example introduced earlier in section 3.2, $CI(1)$ does not represent a valid global state as it does not represent the state of Cro. We also can not have an entity present at two different states, such as: $CI(1)\ CI(0)$, every entity has to appear exactly once within a global state for that state to be a valid state within the MVN. Let MV be an MVN with entities $G = \{g_1, \ldots, g_n\}$, for some $n > 0$. Then we define the set $validGS(MV)$ of RL terms of sort *GState* representing well–defined global states in MV by

$$validGS(MV) = \{g_1(s_1)\ \ldots\ g_n(s_n) \mid s_1 \in D(g_1), \ldots, s_n \in D(g_n)\}$$

The following result shows that rewrite steps in our RL model preserve valid global state terms.

Theorem 4. For any $GS_1 \in validGS(MV)$, if $GS_1 \Longrightarrow GS_2$ in one rewrite step in $RL_A(MV)$ then $GS_2 \in validGS(MV)$.

Proof. Let $GS_1 = g_1(s_1)\ \ldots\ g_n(s_n) \in validGS(MV)$, for some $s_i \in D(g_i)$, $1 \leq i \leq n$. Suppose that $GS_1 \Longrightarrow GS_2$ in one rewrite step in $RL_A(MV)$. Then by the definition of the rewrite rules in $RL_A(MV)$ we know that no term representing the state of an entity is added or removed from a global state term when a rewrite rule is applied. Noting that the multi–set union operator is associative and commutative, it therefore follows that GS_2 must have the form $GS_2 = g_1(s'_1)\ \ldots\ g_n(s'_n)$, where for some $j \in \{1, \ldots, n\}$ we have $s'_i = s_i$, for $i = 1, \ldots, n$, $i \neq j$, and $s'_j \in D(g_j)$ such that $s'_j \neq s_j$. Then by definition we have GS_2 is a valid global state term (i.e. $GS_2 \in validGS(MV)$). □

In order to formally map global states of an MVN MV to corresponding global state terms in $RL_A(MV)$ we define a global state term mapping $\phi : D(g_1) \times \cdots \times D(g_n) \to validGS(MV)$ by

$$\phi(s_1 \dots s_n) = g_1(s_1)\ \dots\ g_n(s_n),$$

for any states $s_i \in D(g_i)$, $1 \leq i \leq n$. Note that since ϕ can be shown to be a bijective mapping it has an inverse $\phi^{-1} : validGS(MV) \to (D(g_1) \times \cdots \times D(g_n))$.

We can now show that $RL_A(MV)$ correctly captures the semantics of the MVN. To do this we show two things: first, the soundness of our translation which means that for every rewrite step in $RL_A(MV)$, a rewrite rule must have been applied to transition from state GS_1 to state GS_2 that was derived from an equation in MV. Secondly, the completeness of our translation which means that for every asynchronous update step in MV that updates a global state S_1 to state S_2, there must exist a rewrite rule derived from an equation in MV that can be used to rewrite S_1 to S_2.

$$\begin{tikzcd}
S_1 \arrow[r, "Asy"] & S_2 \\
GS_1 \arrow[u, "\phi^{-1}"] \arrow[r, "rewrites"'] & GS_2 \arrow[u, "\phi^{-1}"']
\end{tikzcd}$$

(a) Soundness.

$$\begin{tikzcd}
S_1 \arrow[r, "Asy"] \arrow[d, "\phi"'] & S_2 \arrow[d, "\phi"] \\
GS_1 \arrow[r, "rewrites"'] & GS_2
\end{tikzcd}$$

(b) Completeness.

Fig. 3.2 Correctness Requirements for our Translation Approach.

Theorem 5. (Soundness) Let $GS_1, GS_2 \in validGS(MV)$ be any valid global state terms such that $GS_1 \Longrightarrow GS_2$ in a single rewrite step in $RL_A(MV)$. Then $\phi^{-1}(GS_1) \xrightarrow{Asy} \phi^{-1}(GS_2)$. That is shown in figure 3.2 (a), (where $S_1, S_2 \in D(g_1) \times \cdots \times D(g_k)$).

Proof. Suppose that $GS_1 \Longrightarrow GS_2$ in a single rewrite step in $RL_A(MV)$. Then a rewrite rule with the following form must have been applied:

$$g_{E_1}(s_1)\ \dots\ g_{E_m}(s_m)\ g(s) \Longrightarrow g_{E_1}(s_1)\ \dots\ g_{E_m}(s_m)\ \ g(s')$$

for some $m \in \mathbf{N}$, distinct $g_{E_1},\dots,g_{E_m},g \in G$, $s_i \in D(g_{E_i})$, $i = 1,\dots,m$, and $s,s' \in D(g)$ such that $s \neq s'$. By the definition of $RL_A(MV)$ this rule was derived from an equation of the form $g\{s'\} = P_1 + \cdots + P_k$, for some $k > 0$ and product terms P_i, $i = 1,\dots,k$. In particular, the rewrite rule was derived from one of the product terms, say P_i, for some $i \in \{1,\dots,k\}$. It is therefore clear that product term P_i must be true in global state $\phi^{-1}(GS_1)$ and so the state of entity g can be asynchronously updated to state s' (as specified by the equation). In other words, we have that $\phi^{-1}(GS_1) \xrightarrow{Asy} \phi^{-1}(GS_2)$ as required. □

Theorem 6. (Completeness) Let $S_1,S_2 \in D(g_1)\times\cdots\times D(g_n)$ be global states in MV such that $S_1 \xrightarrow{Asy} S_2$. Then $\phi(S_1) \Longrightarrow \phi(S_2)$ in a single rewrite step in $RL_A(MV)$. That is shown in figure 3.2 (b), (where $S_1,S_2 \in D(g_1)\times\cdots\times D(g_k)$).

Proof. Suppose that the asynchronous update $S_1 \xrightarrow{Asy} S_2$ can occur in MV. Then this means the global state S_2 is produced by updating the state of one entity in S_1 while keeping the states of all other entities the same. Suppose that the entity which has been updated is g, for some $g \in G$ and that its state has changed from s to s', for some states $s,s' \in D(g)$ such that $s \neq s'$. Then there must exist an equation of the form $g\{s'\} = P_1 + \cdots + P_k$, for some $k > 0$ and product terms P_i, $i = 1,\dots,k$. Furthermore, at least one of the product terms P_i, for some $i \in \{1,\dots,k\}$, must evaluate to true in the global state S_1. It follows by the definition of $RL_A(MV)$ that there must exist a rewrite rule derived from the above equation and in particular, derived from the product term P_i which has the form:

$$g_{E_1}(s_1)\ \dots\ g_{E_m}(s_m)\ g(s) \Longrightarrow g_{E_1}(s_1)\ \dots\ g_{E_m}(s_m)\ \ g(s')$$

for some $m \in \mathbf{N}$, distinct $g_{E_1},\dots,g_{E_m} \in G$, $s_i \in D(g_{E_i})$, $i = 1,\dots,m$. Clearly, given the assumptions above we must be able to apply this rewrite rule to $\phi(S_1)$ and the resulting state term will correspond to $\phi(S_2)$. In other words, we have that $\phi(S_1) \Longrightarrow \phi(S_2)$ in a single rewrite step in $RL_A(MV)$ as required. □

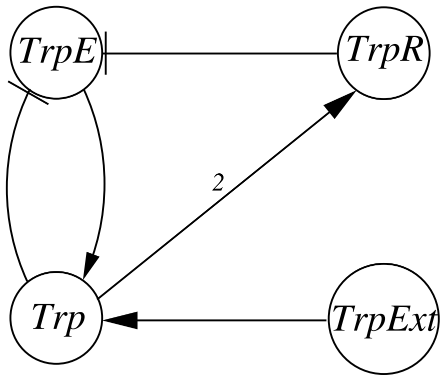
We have now shown that our translation is formally correct. Therefore, anything we derive about our RL model must also hold in the original MVN. Now we can use the powerful support tool available for RL to better analyse and understand an MVN by illustrating the analysis techniques and flexibility available when using RL.

3.4 Case Study: The Regulation of Biosynthesis of Tryptophan in E. coli

We illustrate the RL framework developed above by presenting a case study based on modelling and analysing the genetic regulatory network for the synthesis of tryptophan in *E. coli* [21, 22]. This case study helps to motivate our RL framework by illustrating the analysis techniques and flexibility available when using RL and the tool *Maude* [17].The case study chosen here is a published MVN model of the regulatory system used to control the biosynthesis of tryptophan in E. coli.

3.4.1 Tryptophan Model

Tryptophan is an amino acid that is essential for the development of E. coli. The biosynthesis of the tryptophan amino acid in *E. coli* is carefully regulated because of how essential it is to the growth of the bacteria and how costly it is to produce. The synthesis of tryptophan is resource intensive and for this reason it is carefully controlled to ensure it is only synthesized when no external source of tryptophan is available [21, 22]. An MVN for the underlying genetic regulatory network for the biosynthesis of tryptophan in *E. coli* (based on [23]) is presented in Figure 3.3. The regulatory network has 4 entities:
$TrpE$ – indicates the presence of the activated enzyme required for synthesising tryptophan, has neighbourhood $N(TrpE) = \{TrpR, Trp\}$ and state space $D(TrpE) = \{0, 1\}$;



TrpE	*TrpExt*	*Trp*	[*Trp*]
0	0	0,1	0
0	0	2	1
0	1	0,1,2	1
0	2	0	1
0	2	1,2	2
1	0,1	0,1,2	1
1	2	0	1
1	2	1,2	2

Trp	[*TrpR*]
0,1	0
2	1

Trp	*TrpR*	[*TrpE*]
0	0	1
0	1	0
1,2	0,1	0

Fig. 3.3 An MVN model *MTRP* of the regulatory mechanism for the biosynthesis of Tryptophan in *E. coli* (based on [23]). The state transition table for *TrpExt* has been omitted as this is a simple input entity that does not change its initial state. Note that the state transition tables use a shorthand notation where an entity is allowed to be in any of the states listed for it in a particular row.

$TrpR$ – indicates if the repressor gene for tryptophan production is active, has neighbourhood $N(TrpR) = \{Trp\}$ and state space $D(TrpR) = \{0,1\}$;

$TrpExt$ – an input entity indicating the level of external tryptophan, has $D(TrpExt) = \{0,1,2\}$;

Trp – indicates the level of tryptophan within the bacteria, has neighbourhood $N(Trp) = \{TrpExt, TrpE\}$ and state space $D(Trp) = \{0,1,2\}$.

The above entity order is used when presenting global states for *MTRP*. We can see from the model that the presence of Tryptophan in the external medium *TrpExt* directly affects the level of tryptophan within the bacteria *Trp*. The regulatory network works as follows: the activated enzyme *TrpE* is required to synthesise Tryptophan but this enzyme is deactivated by the presence of Tryptophan within *E. coli* and at higher-levels of Tryptophan the production of *TrpE* is inhibited by the activation of the repressor *TrpR*. The state graph (shown in figure 3.4) consists of 36 global states and has the following three attractors: two point attractors

0011 and 0122 which occur in the presence of external tryptophan; and a cyclic attractor $0000 - - > 1000 - - > 1001 - - > 0001 - - > 0000$ representing tryptophan synthesis.

3.4.2 Constructing the RL Model

Following the approach defined in Section 3.2 we can construct an RL model $RL_A(MTRP)$ for *MTRP*. We begin by deriving equations for the next state functions of *MTRP* from the state transition tables in Figure 3.3. Equations are used to come up with the needed rewrite rules for the Maude file representing the model using the translation process. The regulatory network that controls the biosynthesis of tryptophan by E. coli has been chosen to illustrate basic model checking using Maude.We start by defining the 4 entities (*TrpE*, *TrpR*, *TrpExt* and *Trp*) and assign a sort defining the values the states that they can be in. Entities *TrpE* and *TrpR* both have a state space of size 2 containing the values 0 and 1, while entities *TrpExt* and *Trp* have a state space of size 3 for values 0,1 and 2. The Maude file now represents the network structure as follows:

```
mod EXTRP is
protecting NAT .
sort Entity .
sort GState .
sorts D1 D2 .
subsort Entity < GState .
ops 0 1 : -> D1 .
ops 0 1 2 : -> D2 .
ops TrpE TrpR : D1 -> Entity .
ops TrpExt Trp : D2 -> Entity .
op __ : Entity Entity -> GState [assoc comm].
vars E1 E2 : Entity .
vars s1 s11 : D1 .
endm
```

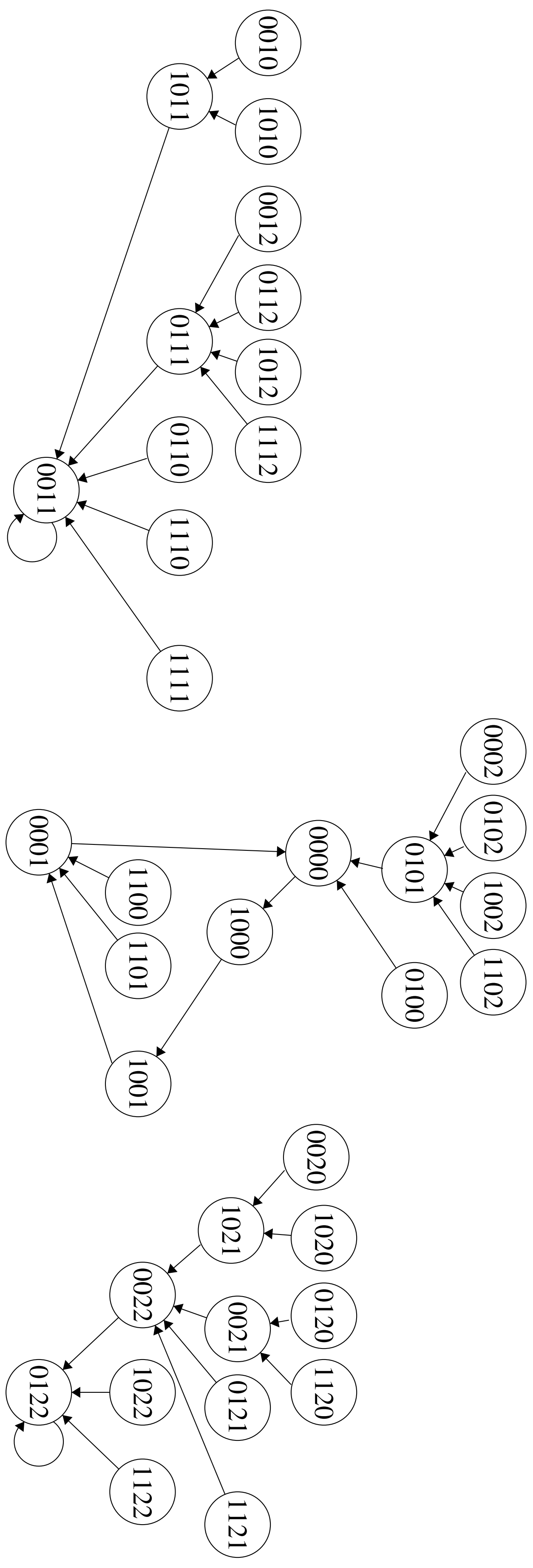


Fig. 3.4 The State Graph of the MTRP Model.

We now have within this module all the necessary operators and variables to represent the structure of this network. With $GState$ being used to represent network entities states and entities defined as a sub sort of $GState$. The simplified set of equations are then used to derive a set of rewrite rules for $RL_A(MTRP)$ to model the asynchronous behaviour of $MTRP$ as described in Section 3.2. We begin by presenting the set of unsimplified equations:

$$\begin{aligned}
&\mathit{TrpE}\{0\} = \mathit{Trp}\{0\}\mathit{TrpR}\{1\} + \mathit{Trp}\{1,2\}\mathit{TrpR}\{0,1\}\\
&\mathit{TrpE}\{1\} = \mathit{Trp}\{0\}\mathit{TrpR}\{0\}\\
&\\
&\mathit{TrpR}\{0\} = \mathit{Trp}\{0,1\}\\
&\mathit{TrpR}\{1\} = \mathit{Trp}\{2\}\\
&\\
&\mathit{Trp}\{0\} = \mathit{TrpE}\{0\}\mathit{TrpExt}\{0\}\mathit{Trp}\{0,1\}\\
&\mathit{Trp}\{1\} = \mathit{TrpE}\{0\}\mathit{TrpExt}\{0\}\mathit{Trp}\{2\} + \mathit{TrpE}\{0\}\mathit{TrpExt}\{1\}\mathit{Trp}\{0,1,2\} +\\
&\mathit{TrpE}\{0,1\}\mathit{TrpExt}\{2\}\mathit{Trp}\{0\} + \mathit{TrpE}\{1\}\mathit{TrpExt}\{0,1\}\mathit{Trp}\{0,1,2\}\\
&\mathit{Trp}\{2\} = \mathit{TrpE}\{0,1\}\mathit{TrpExt}\{2\}\mathit{Trp}\{1,2\}
\end{aligned}$$

Simplifying these equations results in the following set of equations:

$$\begin{aligned}
\mathit{TrpE}\{0\} &= \mathit{TrpR}\{1\} + \mathit{Trp}\{1,2\}\mathit{TrpR}\{0\}\\
\mathit{TrpE}\{1\} &= \mathit{Trp}\{0\}\mathit{TrpR}\{0\}\\
\\
\mathit{TrpR}\{0\} &= \mathit{Trp}\{0,1\}\\
\mathit{TrpR}\{1\} &= \mathit{Trp}\{2\}\\
\\
\mathit{Trp}\{0\} &= \mathit{TrpE}\{0\}\mathit{TrpExt}\{0\}\mathit{Trp}\{0,1\}\\
\mathit{Trp}\{1\} &= \mathit{TrpE}\{0\}\mathit{TrpExt}\{0\}\mathit{Trp}\{2\} + \mathit{TrpExt}\{1\} +\\
&\ \mathit{TrpExt}\{2\}\mathit{Trp}\{0\} + \mathit{TrpE}\{1\}\mathit{TrpExt}\{0\}\\
\mathit{Trp}\{2\} &= \mathit{TrpExt}\{2\}\mathit{Trp}\{1,2\}
\end{aligned}$$

We use these equations to derive a set of rewrite rules that are used by Maude to navigate through network states. Taking $TrpR$ as an example we have the following equations for $TrpR$:

$$\begin{aligned} TrpR\{0\} &= Trp\{0,1\} \\ TrpR\{1\} &= Trp\{2\} \end{aligned}$$

$TrpR$ updates its value to 0 when Trp is either at state 0 or 1, and it updates its value to 1 when Trp is at state 2. Since a value can only be updated if there is a change in the value itself, we represent that by adding $TrpR(1)$ for $TrpR\{0\}$ and $TrpR(0)$ for $TrpR\{1\}$ as follows:

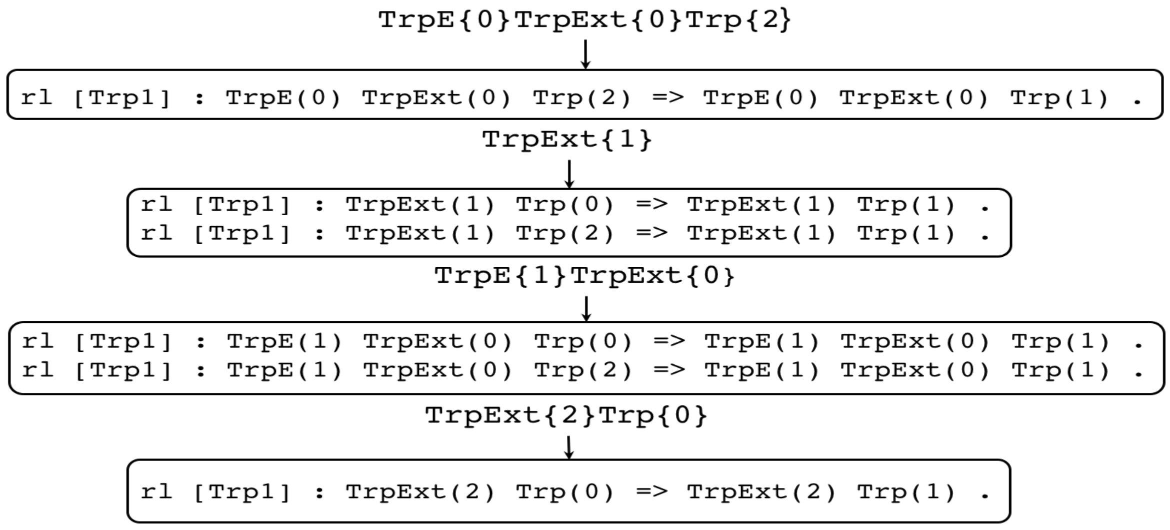
```
rl [TrpR0] : TrpR(1) Trp(0) => TrpR(0) Trp(0) .
rl [TrpR0] : TrpR(1) Trp(1) => TrpR(0) Trp(1) .
rl [TrpR1] : TrpR(0) Trp(2) => TrpR(1) Trp(2) .
```

Rewrite rules are derived from the network entities truth tables. They are used to rewrite network states (going from one network state to the other). As an example: Entity `TrpR` has the value of 0 when `Trp` is either 0 or 1, we use rewrite rules to describe that (also `TrpR` has to be at state 1 to change to state 0 because rewrite rules are only used when there are changes in the network entities values). Following the same procedure for the remaining network entities, we end up with the full set of rewrite rules that goes into the Maude file. The set of rewrite rules derived for $RL_A(MTRP)$ is as follows:

```
rl [TrpR0] : TrpR(1) Trp(0) => TrpR(0) Trp(0) .
rl [TrpR0] : TrpR(1) Trp(1) => TrpR(0) Trp(1) .
rl [TrpR1] : TrpR(0) Trp(2) => TrpR(1) Trp(2) .

rl [TrpE0] : TrpE(1) TrpR(1) => TrpE(0) TrpR(1) .
rl [TrpE0] : TrpE(1) Trp(1) TrpR(0) => TrpE(0) Trp(1) TrpR(0) .
rl [TrpE0] : TrpE(1) Trp(2) TrpR(0) => TrpE(0) Trp(2) TrpR(0) .
rl [TrpE1] : TrpE(0) Trp(0) TrpR(0) => TrpE(1) Trp(0) TrpR(0) .
rl [Trp0] : TrpE(0) TrpExt(0) Trp(1) => TrpE(0) TrpExt(0) Trp(0) .
rl [Trp1] : TrpE(0) TrpExt(0) Trp(2) => TrpE(0) TrpExt(0) Trp(1) .
rl [Trp1] : TrpExt(1) Trp(0) => TrpExt(1) Trp(1) .
rl [Trp1] : TrpExt(1) Trp(2) => TrpExt(1) Trp(1) .
rl [Trp1] : TrpE(1) TrpExt(0) Trp(0) => TrpE(1) TrpExt(0) Trp(1) .
rl [Trp1] : TrpE(1) TrpExt(0) Trp(2) => TrpE(1) TrpExt(0) Trp(1) .
rl [Trp1] : TrpExt(2) Trp(0) => TrpExt(2) Trp(1) .
rl [Trp2] : TrpExt(2) Trp(1) => TrpExt(2) Trp(2) .
```

Figure 3.5 shows the four parts of the $Trp\{1\}$ equation and the corresponding rewrite rules, this was singled out because it produced many rewrite rules. First part of the equation produces a rewrite rule that is straight forward. Second and third parts are interesting because Trp is not present in those equations. With Trp having a state space of 3 values (0,1 and 2) and it being not present in the equations for $Trp\{1\}$, we have to cover both possibilities of Trp being either `0` or `2` before it was updated to `1` and that is why have 2 equations for the second and third equation parts while the fourth part of the $Trp\{1\}$ produces a straightforward rewrite rule.



```
rl [Trp1] : TrpE(0) TrpExt(0) Trp(2) => TrpE(0) TrpExt(0) Trp(1) .

rl [Trp1] : TrpExt(1) Trp(0) => TrpExt(1) Trp(1) .
rl [Trp1] : TrpExt(1) Trp(2) => TrpExt(1) Trp(1) .

rl [Trp1] : TrpE(1) TrpExt(0) Trp(0) => TrpE(1) TrpExt(0) Trp(1) .
rl [Trp1] : TrpE(1) TrpExt(0) Trp(2) => TrpE(1) TrpExt(0) Trp(1) .

rl [Trp1] : TrpExt(2) Trp(0) => TrpExt(2) Trp(1) .
```

Fig. 3.5 Rewrite Rules Produced for `Trp1`.

Once a model has been developed for a biological system then the next stage is to analyse its behaviour. The idea is to validate the model by checking that it has known biological properties and to produce important new insights that can then be experimentally investigated by biologists. We illustrate the wide range of analysis possible using our RL framework and the support tool Maude by providing a selection of analysis examples for $RL_A(MTRP)$.

3.4.3 Analysis in Maude

The above model can now be formally investigated using the range of powerful analysis tools provided by Maude which was introduced in Section 2.5. We briefly illustrate the range of analysis techniques available below (for a more detailed introduction to Maude's analysis tools see [28]).

Rewrite rules can now be used by Maude to rewrite system states and perform searches. Using Maude's `rew` command we rewrite the system state `0000` one time (i.e. gets the next state for `0000`); the resulting output is as follows:

```
Maude> rew [1] TrpR(0) TrpE(0) TrpExt(0) Trp(0) .
rewrite [1] in exTrp : TrpR(0) TrpE(0) TrpExt(0) Trp(0) .
rewrites: 1 in 0ms cpu (0ms real) (1000000 rewrites/second)
result [State]: TrpE(1) TrpR(0) TrpExt(0) Trp(0)
```

Maude performs a tracing simulation of one rewrite and the resulting state is `1000` as expected. Maude also shows the rewriting CPU and real time alongside the number of rewrites performed per second. The number of rewrites can be specified in Maude as follows:

```
Maude> rew [3] TrpR(0) TrpE(1) TrpExt(1) Trp(2) .
rewrite [3] in exTrp : TrpR(0) TrpE(1) TrpExt(1) Trp(2) .
rewrites: 3 in 0ms cpu (0ms real) (3000000 rewrites/second)
result [State]: TrpE(0) TrpR(1) TrpExt(1) Trp(1)
```

Maude performs 3 rewrites on the given initial state and returns the state `0111` as a result. We can also omit the number of rewrites and Maude will rewrite to termination as follows:

```
Maude> rew TrpR(0) TrpE(1) TrpExt(1) Trp(2) .
rewrite in exTrp : TrpR(0) TrpE(1) TrpExt(1) Trp(2) .
rewrites: 4 in 0ms cpu (0ms real) (4000000 rewrites/second)
result [State]: TrpE(0) TrpR(0) TrpExt(1) Trp(1)
```

Maude rewrites the initial state to termination after performing 4 rewrites resulting in state `0011`.

Basic model checking can be done using the Maude search command. We can start by checking for Trp changing from state `0` to state `1` starting from the global state `0010` using the following search command:

```
search TrpR(0) TrpE(0) TrpExt(1) Trp(0) =>+
Trp(1) TrpR(t1:D1) TrpE(t2:D1)TrpExt(t3:D2) .
Solution 1 (state 2)
states: 3 rewrites: 2 in 0ms cpu (0ms real)
(40816 rewrites/second)
t2:D1 --> (0).D1
t1:D1 --> (0).D1
t3:D2 --> (1).D2
Solution 2 (state 3)
states: 4 rewrites: 3 in 0ms cpu (0ms real)
(34090 rewrites/second)
t2:D1 --> (1).D1
t1:D1 --> (0).D1
t3:D2 --> (1).D2
No more solutions.
states: 4 rewrites: 4 in 0ms cpu (0ms real)
(33333 rewrites/second)
```

This search is limited to reachable states starting from 0010 that has Trp at state 1. Maude's Search Command explores all reachable states that match the pattern given (`Trp(1)`

`TrpR(t1:D1) TrpE(t2:D1) TrpExt(t3:D2))` which means any reachable state where the value of Trp changes to 1. As seen above two states are reachable which are 0011 and 0111. Basic model checking can be done in Maude using the Search command. Next, we present some examples to illustrate the type of invariant analysis possible using Maude's built-in `search` command [17] which provides interesting ways to search the set of states reachable from a given initial state. For example, the following search allows the attractors in our model to be investigated by checking whether an initial state leads to a point attractor (note the use of `=>!` here to ensure the rewriting terminates).

```
search TrpE(1) TrpR(0) TrpExt(2) Trp(2) =>! GS:GState .
```

This command confirms that 1022 leads to the point attractor 0122 in *MTRP* and we can view the associated trace for this behaviour.

Using the `show path` command we could then view an example trace for this behaviour.

```
state 0, GState: TrpE(1) TrpR(0) TrpExt(2) Trp(2)
===[ rl TrpR(0) Trp(2) => TrpR(1) Trp(2) [label nxtTrpR1] . ]===>
state 1, GState: TrpE(1) TrpR(1) TrpExt(2) Trp(2)
===[ rl TrpE(1) TrpR(1) => TrpE(0) TrpR(1) [label nxtTrpE0] . ]===>
state 3, GState: TrpE(0) TrpR(1) TrpExt(2) Trp(2)
```

A similar search shows that the initial state 1102 does not lead to a point attractor.

```
search TrpE(1) TrpR(1) TrpExt(0) Trp(2) =>! GS:GState .
```

We can also check general invariant properties on the state space reachable from a given initial state such as checking whether entities reach a certain level of activation as the example below illustrates.

```
search TrpE(1) TrpR(1) TrpExt(1) Trp(0) =>+ Trp(2) GS:GState .
```

This search confirms that *Trp* can not reach state 2 from the initial state 1110. A similar search shows that *Trp* can reach state 2 from initial state 0120 (in fact, there are four different traces that result in this behaviour and we can view these). As a final example, consider the following search which confirms that *TrpE* and *TrpR* are not mutually exclusive from the initial state 0120.

```
search TrpE(0) TrpR(1) TrpExt(2) Trp(0) =>+ TrpE(1) TrpR(1) GS:GState .
```

Maude indicates there is a single counter example trace here, and we able to view it to gain insight into this behaviour. A similar search confirms that *TrpE* and *TrpR* are mutually exclusive from 0121. Furthermore, we can show that the basins of attraction for each attractor in *MTRP* are disjoint. Using this approach we can explore a wide range of interesting properties for $trpMV$.

3.4.4 LTL Model Checking

The above examples give an initial idea of the wide range of analysis checks possible of our RL model. Maude also provides a model checking tool for *Linear Temporal Logic (LTL)* [37] which allows dynamic properties that can not be checked using the search command to be analysed for finite state rewrite systems [27]. Since any MVN has a finite state space we can use this model checking tool to investigate a wide range of biologically relevant dynamic properties of an MVN.

To use the model checker we start by defining a range of atomic propositions to represent the key properties of interest. For example, for *MTRP* we might define an atomic proposition `atTrp:D2 -> Prop`, where `atTrp(s)` is true only if *Trp* is in state `s`. This can be done equationally as follows (where `[owise]` is a Maude shortcut which allows all remaining possibilities to be covered):

```
eq Trp(s2) GS |= atTrp(s2) = true .
eq GS |= atTrp(s2) = false [owise] .
```

We can then model check a range of interesting dynamic properties expressed using the temporal operators of LTL as illustrated by the examples below for *MTRP*.

The first property we consider is used to validate the model by showing that the Tryptophan enzyme is always eventually present in the bacteria. We represent this property in

Maude using the LTL formula `[] <> (atTrp(1) \/ atTrp(2))`, where `[]` stands for the always operator and <> for eventually [37]. We can check this LTL formula for the initial state 0000 using the following Maude reduce command:

```
red modelCheck(TrpE(0) TrpR(0) TrpExt(0) Trp(0),
[] <> (atTrp(1) \/ atTrp(2)))
```

This returns true showing that the property holds and indeed it holds for all initial states considered.

Another property we might consider is whether an entity eventually becomes stable in a given state from a given initial state. For example, suppose we want to check that from initial state 0020 in which input entity *TrpExt* is in state 2, *Trp* must eventually become fixed in state 2 (i.e. present at high–levels in the bacteria). We can confirm this property holds using the command:

```
red modelCheck(TrpE(0) TrpR(0) TrpExt(2) Trp(0), <> [] atTrp(2))
```

We can extend this property further to check whether a high–level of external tryptophan is enough to ensure a high–level of tryptophan in the bacteria. We represent this property using the formula `atExt(2) -> (<> [] atTrp(2))`, and model checking shows it holds for all initial states sampled.

As a final example, suppose we want to check whether an entity entering a specific state can trigger some important behaviour. For example, the following LTL formula `[] (atTrp(2) -> <> atR(1))`, captures the property that if *Trp* is ever fully expressed then eventually *TrpR* must become active. We can model check this LTL formula for some initial state, say , *TrpE*$\{1\}$ *TrpR*$\{0\}$ *TrpExt*$\{0\}$ *Trp*$\{2\}$, using the following Maude command:

```
red modelCheck(TrpE(1) TrpR(0) TrpExt(0) Trp(2),
[] (atTrp(2) -> <> atR(1)))
```

This returns false showing that the formula does not hold (as expected) and provides a counter example trace which gives important insight into the result. Repeating the above check we are able to confirm that the property does hold for initial state 1020.

Under appropriate conditions, mathematical models can be checked to check whether they satisfy some important properties, or in some cases, obtain a useful counterexample showing that such property is violated. While any Linear Temporal Logic (LTL) property of a system can be model checked when the system is specified in Maude as a system module, Search allows for a simpler, yet very useful model-checking capabilities, such as model-checking of invariants, which can be accomplished just by using the Maude Search command.

3.5 Conclusions

In this chapter, we worked on developing an RL model for asynchronous MVNs based on their asynchronous update semantics. We started by introducing a translation approach that translates an MVN into an RL model. The translation approach deals with straightforward translations taking a single equation part and producing a matching rewrite rule. More importantly, the translation approach deals with more complex translations when an entity is missing from the equation and can hold more than one value hence the need for two or more equations as was shown in Section 3.4.2. The translation approach also has to make sure to add missing terms in case they are not present.

We provided a formal correctness argument for this translation approach where we formally showed that our translation from an asynchronous MVN to an RL model is correct. This was done by proving the soundness (see Section 3.3) (each global state transition possible in our RL model represents a corresponding asynchronous update in the original MVN) and completeness (see Section 3.3) (every asynchronous update possible in an MVN is specified in our RL model) of the RL model (see Figure 3.2).

Techniques and the developed RL framework were illustrated in this chapter using a detailed case study that was based on modelling and analysing the genetic regulatory network for the synthesis of tryptophan in E. coli. The case study have helped motivate our RL framework by illustrating the analysis techniques and flexibility available when using RL and the tool *Maude* [17]. We formally investigated the model using a range of powerful analysis tools provided by Maude, which was one of the motivations for choosing to use RL. We demonstrated a wide range of analysis where we started by using basic rewrite commands before moving into the `search` command, we performed some basic search commands checking for certain properties. After that we introduced LTL model checking and showed the powerful and wide range of analysis that LTL allows us to perform.

We extend our translation approach in the next chapter to cover the synchronous semantics of MVNs and prove correctness using a detailed case study. The synchronous mechanisms are more challenging as we need to introduce and make use of rewriting strategies (more on that in Chapter 4).

In order to evaluate the performance and the scalability of our RL framework, further ahead in Chapter 5 we take a look at a scalable case study using a performance test model. The model is a multi-valued network of five entities. The network has four boolean entities and one multi–valued one, with A acting as the only multi-valued entity in the network with a state space of size three for values 0, 1 and 2. Using the developed techniques and tools we start by presenting an asynchronous version of the model. We perform some analysis using the Maude's search command and rewriting logic. Then we introduce the synchronous version of the model and perform a similar analysis. We perform some LTL model checking before scale the model in increments of five and then providing analysis summary tables.

In order to illustrate the practical application of tools and techniques developed for our RL framework, we carry out a larger case study in Chapter 6 by introducing models of sizes 13 and 22 using a scalable test model based on the gene regulatory network of the segment polarity gene family which is at the basis of *Drosophila* embryonic development [95, 94].

Chapter 4

An RL Model for Synchronous MVNs

4.1 Introduction

Synchronous MVNs [24, 44, 83] are an important modelling technique that has a wide range of applications. In this chapter we focus on using the synchronous update rule for MVNs as opposed to the asynchronous rule used in the previous chapter. Synchronous MVNs are seen as being less realistic than asynchronous MVNs [12] because of their assumption of simultaneous updates which can lead to deterministic, infinite traces. Their dynamics can be easier to analyse and this has made them very popular, while asynchronous MVNs can result in too much behaviour which can make their dynamics more difficult to analyse.

In this chapter, we extend our work on asynchronous MVNs (Chapter 3) to synchronous MVNs to continue to strengthen the tool support available for MVNs. In particular, we provide a formal translation of synchronous MVNs to RL. The challenge here lies in the ability to coordinate update steps to ensure that entities are using the current states of other entities as inputs rather than next states. In order to handle this we make use of rewriting strategies to implement a two phase update [41]. This includes first storing all entities' next states and then performing a coordinated state update. We implement this by making use of rewriting strategies implemented using Maude's metalevel capabilities [16, 41, 70].

We start by developing an RL model for synchronous MVNs. Similar to what we have done in Chapter 3, we derive a set of rewrite rules from the MVN's set of equations to represent system transitions using our translation approach. However the difference here is that we have two values for each entity in our module storing both current and next states. We develop our model using Maude and implement within our model the next state function associated with each entity and the required synchronous update rules. We make use of rewriting strategies which are required for synchronous updates to apply a two phase state update as follows: First, we define a metalevel operation that applies rewrite rules using current entities' states using synchronous rewrite rules. We then build on this by defining a metalevel operation that resets the current state of each entity after applying the first phase, and that gives us all we need to analyse our model using Maude.

We formally show that our translation process from a synchronous MVN to an RL model is correct by proving the soundness and completeness of our translation process (similar to the approach used in Section 3.3). In particular, this includes showing the *soundness* and *completeness* of our translation approach which we introduced in Section 3.3.

We illustrate the techniques and the RL framework developed by presenting a case study using an existing MVN model for the genetic regulatory network controlling the lysis–lysogeny switch in the bacteriophage λ [25, 4]. The case study helps to motivate our RL framework by illustrating the analysis techniques and tools when using RL and Maude. We formally investigate the model using simple model checking based on the search command and LTL model checking.

This chapter is organised as follows, In Section 4.2, We construct an RL model for a synchronous MVN, and we derive a set of rewrite rules using a translation approach that enables us to produce a formal representation of an MVN using RL. In Section 4.3 we prove the correctness of our translation approach by showing it soundness and completeness. In Section 4.4, we illustrate our RL framework by presenting a detailed case study based on an existing MVN model for the genetic regulatory network controlling the lysis–lysogeny switch in the bacteriophage λ. Finally in Section 4.5, we conclude the work presented in this chapter and give a brief introduction to Chapter 5 where we evaluate the performance of the techniques developed in this Chapter and Chapter 3.

4.2 Constructing an RL Model for a Synchronous MVN

Modelling the synchronous update semantics for an MVN in RL follows along similar lines to the asynchronous approach in Section 3.2. However, it is more complex since we have to ensure that all entities update their state simultaneously when moving from one global state to another. For this reason we use a *two phase update approach* [38] for computing global next states:

1) Compute and record the next state of each entity while preserving their current states.

2) Update the current states of all entities to reflect the recorded next states.

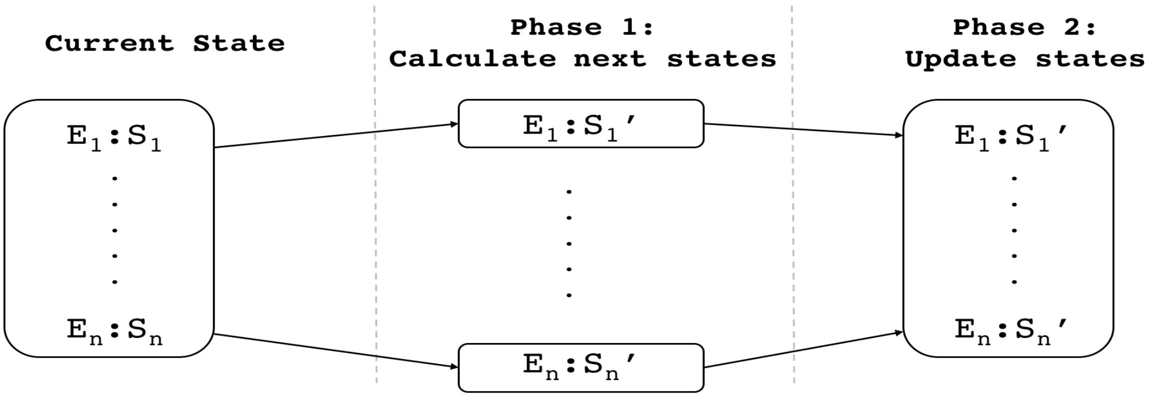


Fig. 4.1 The Two Phase Update for Synchronous MVNs.

4.2.1 Basic RL Model

To model the first phase of the update step we take the basic sort and function definitions given in Section 3.2 and adapt the term representation of entities so that they contain a second state component to represent the recorded next state. Given an MVN $MV = (G,D,N,F)$ we let $RL_S(MV)$ represent the RL model resulting from the construction outlined below for the synchronous update semantics of MV. For each entity $g \in G$ in the MVN we define the function $g : D_m \times D_m \to Entity$, where $D(g) = \{0,\ldots,m\}$. The idea is that $g(s_1,s_2)$ represents that g is currently in state s_1 and will have next state s_2. We will use the convention that before a synchronous update takes place each entity's current and next state are the same. For example, the global state 12 in the example MVN *PL2* (see Section 2.3) would be represented by the term `CI(1,1) Cro(2,2)`. Thus the current and next states of an entity will only differ when we are partly through the synchronous update step. Figure 4.2 illustrates our two phase update approach for state $CI(1,1)\ Cro(2,2)$ transitioning to state $CI(0,0)\ Cro(1,1)$:

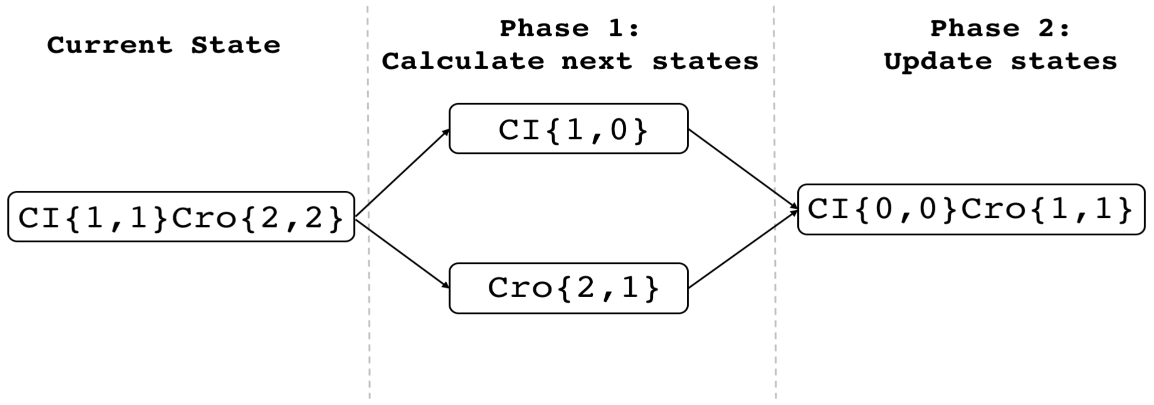


Fig. 4.2 The Two Phase Update for Synchronous semantics.

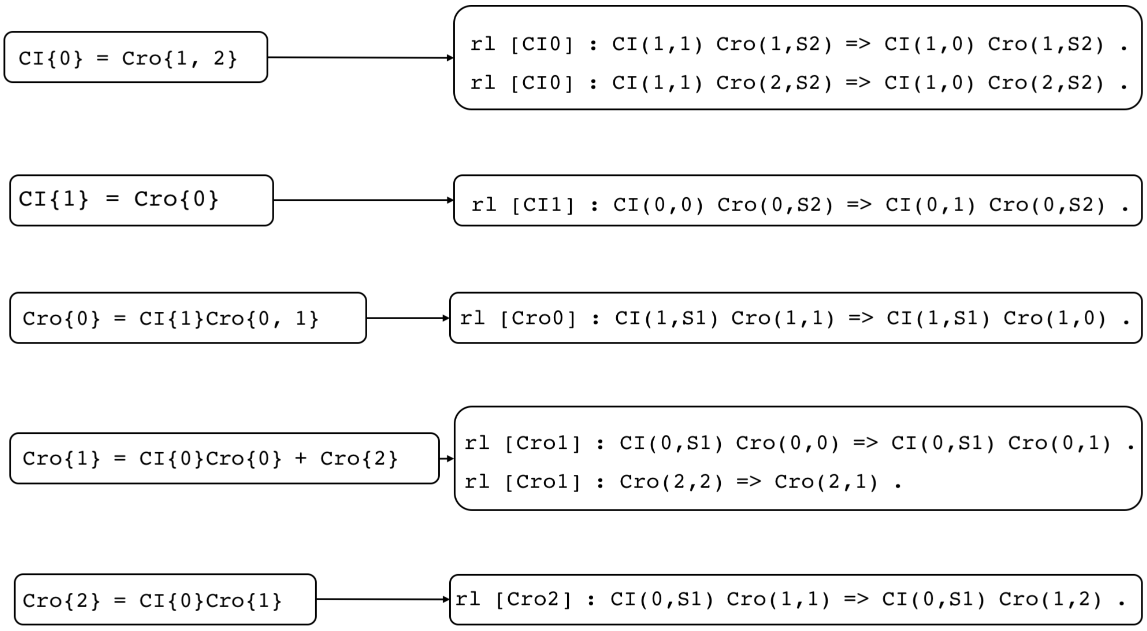
We formalize the next–state function associated with each entity using a similar approach to that detailed in Section 3.2 for the asynchronous model. The difference here is that we reformulate the resulting rewrite rules so that the current state is not changed at this stage (Phase 1) and instead the next state of an entity is simply recorded. To illustrate this consider the rewrite rule

```
rl [CI0] : CI(1) Cro(1) => CI(0) Cro(1) .
```

derived previously to represent an asynchronous update of *CI* from state 1 to 0 in *PL2* (see Section 2.3). This rule is replaced by the following one in the synchronous case:

```
rl [CI0] : CI(1,1) Cro(1,s2) => CI(1,0) Cro(1,s2) .
```

There are now two values for every entity, old and next value; if the two values match it means that the value of the entity has not been updated yet. While if the two values are different it means that the update step has already taken place. The use of variable `s2` is needed since the next state of *Cro* may or may not have been updated at this stage. Note that this rule can only be applied if the current and next state of *CI* are the same. This is important as it ensures the rule can only be applied at most once during a given update step and links to our assumption above about the representation of global states. Figure 4.3 shows our translation approach for the synchronous case applied to our running example *PL2*.



```
rl [CI0] : CI(1,1) Cro(1,S2) => CI(1,0) Cro(1,S2) .
rl [CI0] : CI(1,1) Cro(2,S2) => CI(1,0) Cro(2,S2) .
rl [CI1] : CI(0,0) Cro(0,S2) => CI(0,1) Cro(0,S2) .
rl [Cro0] : CI(1,S1) Cro(1,1) => CI(1,S1) Cro(1,0) .
rl [Cro1] : CI(0,S1) Cro(0,0) => CI(0,S1) Cro(0,1) .
rl [Cro1] : Cro(2,2) => Cro(2,1) .
rl [Cro2] : CI(0,S1) Cro(1,1) => CI(0,S1) Cro(1,2) .
```

Fig. 4.3 The Translation Process: Equations to Rewrite Rules (Applied to *PL2*).

The full set of rewrite rules used to model the synchronous case for *PL2* is presented below:

```
rl [CI0] : CI(1,1) Cro(1,s2) => CI(1,0) Cro(1,s2) .
rl [CI0] : CI(1,1) Cro(2,s2) => CI(1,0) Cro(2,s2) .
rl [CI1] : CI(0,0) Cro(0,s2) => CI(0,1) Cro(0,s2) .
rl [Cro0] : CI(1,s1) Cro(1,1) => CI(1,s1) Cro(1,0) .
rl [Cro1] : CI(0,s1) Cro(0,0) => CI(0,s1) Cro(0,1) .
rl [Cro1] : Cro(2,2) => Cro(2,1) .
rl [Cro2] : CI(0,s1) Cro(1,1) => CI(0,s1) Cro(1,2) .
```

The idea is that rewriting using the resulting set of rewrite rules will update the next states of entities and that when rewriting terminates the first phase of the two phase synchronous update will be complete. To model the second, synchronization phase of the update we need to replaces each current state by its recorded next state. In order to do this we define an update function `upDate:GState -> GState` recursively as follows:

1) For each entity `E`, we have an update equation of the form (where `s1` and `s2` are state variables corresponding to possible state at `E`):

```
eq upDate(E(s1,s2)) = E(s2,s2) .
```

2) We also have a recursive equation that calls `upDate` over network states and updates each entity's state (where `GS1` and `GS2` are variables of sort `GState`):

```
eq upDate(GS1 GS2) = upDate(GS1) upDate(GS2) .
```

For our running example *PL2*, these equations for *CI* and *Cro* would be written as follows (where `s1`, `s11` are state variables of sort $D1$, and `s2` and `s22` are state variables of sort $D2$):

```
eq upDate(CI(s1,s11)) = CI(s11,s11) .
eq upDate(Cro(s2,s22)) = Cro(s22,s22) .
eq upDate(GS1 GS2) = upDate(GS1) upDate(GS2) .
```

We now have all the functionality required to implement the two phase synchronous state update and what is now needed is a way to combine them correctly. We do this by making use of Maude's metalevel capabilities [17] to define an operator to capture the required rewriting strategy.

4.2.2 Using a Rewriting Strategy for Synchronous Updates

When working at the metalevel global states will be represented using Maude's meta–notation and are given the meta–type `Term`. For example, the metalevel term:

```
'__['CI['0.D1,'0.D1],'Cro['1.D2,'1.D2]]
```

will be used to represent the global state term `CI(0,0) Cro(1,1)` (Note that Maude provides the operator `upTerm` to lift a term to the metalevel.) The rewriting strategy we require for synchronous updates is defined in two parts (following the two phase update):

1) We define a metalevel operation `phaseOne : Term -> Term` which implements the first phase of the global state update by applying the synchronous rewrite rules (in this case assumed to be in module `EXPL2`) to a global state term `T` using `metaRewrite` (which takes as arguments the metarepresentation of a module, the metarepresentation of a term `T`, and a value `b` of the sort `Bound`, i.e., either a natural number or the constant `unbounded`):

```
ceq phaseOne(T) =
    if Step? :: ResultPair then
        getTerm(Step?)
    else
        T
    fi
    if Step? := metaRewrite(upModule('EXPL2, false), T, unbounded) .
```

2) We then build on this by defining a metalevel operation `next : Term -> Term` which applies `phaseOne` to a term and then resets the current state using the `upDate` function:

```
ceq next(T) =
    if Step? :: ResultPair then
        getTerm(Step?)
    else
        T1
    fi
    if T1 := phaseOne(T) /\
        Step? := metaReduce(upModule('EXPL2, false), 'upDate[T1]) .
```

Note that in the above we use the metalevel representation of `upDate` as indicated by the back–quote and that `metaReduce` is used to apply its defining equations.

We can now use the metalevel operator `next` to simulate synchronous update steps. For example, the synchronous step 12 $\xrightarrow{Syn}$ 01 in *PL2* can be reproduced by the following Maude command:

```
red next('__['CI['1.D1,'1.D1],'Cro['2.D2,'2.D2]]) .
```

which correctly returns the metalevel state term:

```
'__['CI['0.D1,'0.D1], 'Cro['1.D2,'1.D2]].
```

The first phase of the two phase update can be produced by the following Maude command:

```
red phaseOne('__['CI['1.D1,'1.D1],'Cro['2.D2,'2.D2]]) .
```

which correctly returns the metalevel state term:

```
'__['CI['1.D1,'0.D1], 'Cro['2.D2,'1.D2]].
```

We can now make use of the rewriting strategy `next` to produce traces (based on repeatedly applying a step using an equation `run` that is recursively defined on `next`):

```
op run : Nat Term -> Term .
eq run(0,T) = T .
eq run(s(N),T) = run(N,next(T)) .
```

The first equation returns the input term as a result, while the second equation is a recursive call to `next` to produce a trace using `N` as the number of steps to be taken. We can now make use of `run` to produce a trace as follows:

```
Maude> red run( 3 , '__['CI['1.D1,'1.D1],'Cro['2.D2,'2.D2]]) .
reduce in PL2-META : run(3, '__['CI['1.D1,'1.D1],'Cro['2.D2,'2.D2]]) .
rewrites: 53 in 0ms cpu (2ms real) (68564 rewrites/second)
result GroundTerm: '__['CI['0.D1,'0.D1],'Cro['1.D2,'1.D2]]
```

In order to use the full range of Maude's analysis tools to investigate the behaviour of the *PL2* model, we have to ensure the rewriting strategy `next` we developed is invoked when rewriting the model at the metalevel. This can be done by adding the following rewrite rule (where `T1` and `T2` are variables of type `Term`):

```
rl [step] : '__[T1,T2] => next('__[T1,T2]) .
```

This approach allows `next` to be introduced after each synchronous update step and is based on the fact that Maude always applies equations first before considering rewrite rules, this is critical when it comes to searching and using LTL at the metalevel.

4.3 RL Model Correctness

We now show that the RL model proposed above for an MVN using the synchronous update semantics is correct by following a similar approach to that used in Section 3.3 for the asynchronous case. We begin by precisely defining which terms in our RL model represent well–defined global states in the synchronous case. We define the set $validGS(MV)$ of RL terms of sort *GState* representing well–defined global states in an MVN *MV* by:

$$validGS(MV) = \{g_1(s_1,s_1)\ \ldots\ g_n(s_n,s_n) \mid s_1 \in D(g_1),\ldots,s_n \in D(g_n)\}$$

The following result shows that the metalevel operator `next` preserves valid global state terms. Note that to apply `next` we need to move to and from the metalevel representation of state terms.

Theorem 7. For any $GS_1 \in validGS(MV)$, if $GS_1 \Longrightarrow GS_2$ in one application of `next` in $RL_S(MV)$ then $GS_2 \in validGS(MV)$.

Proof. Let $GS_1 = g_1(s_1,s_1)\ \ldots\ g_n(s_n,s_n) \in validGS(MV)$, for some $s_i \in D(g_i)$, $1 \leq i \leq n$. Suppose that $GS_1 \Longrightarrow GS_2$ in one application of `next` in $RL_S(MV)$. By the definition of `phaseOne` and the synchronous rewrite rules in $RL_S(MV)$ we know that no term representing the state of an entity is added or removed from a global state term. It follows that after applying `phaseOne` to GS_1 we must have a state term of the form $g_1(s_1,s'_1)\ \ldots\ g_n(s_n,s'_n)$, for some $s'_i \in D(g_i)$, $1 \leq i \leq n$. Applying the reset function `upDate` to this state term will result in the state term $GS_2 = g_1(s'_1,s'_1)\ \ldots\ g_n(s'_n,s'_n)$ which is clearly a valid global state term as defined above. □

We define a global state term mapping $\phi : D(g_1) \times \cdots \times D(g_n) \to validGS(MV)$ by

$$\phi(s_1 \ldots s_n) = g_1(s_1,s_1)\ \ldots\ g_n(s_n,s_n),$$

for any states $s_i \in D(g_i)$, $1 \leq i \leq n$.

The following results show that $RL_S(MV)$ is a correct (i.e. *sound* and *complete*) model of MV under the synchronous update semantics.

$$\begin{tikzcd} S_1 \arrow[r, "Syn"] & S_2 \\ GS_1 \arrow[u, "\phi^{-1}"] \arrow[r, "next"'] & GS_2 \arrow[u, "\phi^{-1}"'] \end{tikzcd}$$

(a) Soundness.

$$\begin{tikzcd} S_1 \arrow[r, "Syn"] \arrow[d, "\phi"'] & S_2 \arrow[d, "\phi"] \\ GS_1 \arrow[r, "next"'] & GS_2 \end{tikzcd}$$

(b) Completeness.

Fig. 4.4 Correctness Requirements for our Translation Approach.

Theorem 8. (Soundness) Let $GS_1, GS_2 \in validGS(MV)$ be any valid global state terms such that $GS_1 \Longrightarrow GS_2$ in one application of `next` in $RL_S(MV)$. Then $\phi^{-1}(GS_1) \xrightarrow{Syn} \phi^{-1}(GS_2)$.

Proof. Let $GS_1 = g_1(s_1,s_1)\ \ldots\ g_n(s_n,s_n)$, $GS_2 = g_1(s'_1,s'_1)\ \ldots\ g_n(s'_n,s'_n) \in validGS(MV)$, for some $s_i, s'_i \in D(g_i)$, $1 \leq i \leq n$. Suppose that $GS_1 \Longrightarrow GS_2$ in one application of `next` in $RL_S(MV)$. Then by the definition of ϕ we need to show that $s_1 \ldots s_n \xrightarrow{Syn} s'_1 \ldots s'_n$.

For each $i = 1, \ldots, n$ there are two possible cases to consider:

Case 1: Suppose $s_i \neq s'_i$. Then a rewrite rule with the following form must have been applied:

$$g_{E_1}(s_{E_1},v_1)\ \dots\ g_{E_m}(s_{E_m},v_m)\ g_i(s_i,s_i) \Longrightarrow g_{E_1}(s_{E_1},v_1)\ \dots\ g_{E_m}(s_{E_m},v_m)\ \ g_i(s_i,s_i')$$

for some $m \in \mathbf{N}$, distinct entities $g_{E_1},\dots,g_{E_m} \in G$ and state variables $v_1,\dots,v_m$. By the definition of $RL_S(MV)$ this rule was derived from an equation of the form $g_i\{s_i'\} = P_1 + \cdots + P_k$, for some $k > 0$ and product terms P_j, $j = 1,\dots,k$. It is therefore clear that the right hand side of the equation must be true in the global state $s_1 \dots s_n$ and so by definition the state of entity g_i must be updated to s_i' after a synchronous update is applied.

Case 2: Suppose $s_i = s_i'$. Then the state of entity g_i is not changed meaning that none of the rewrite rules updating g_i were applicable in GS_1. Since by the definition the rewrite rules of $RL_S(MV)$ are derived from the next state equations of MV this implies that the only equation applicable for g_i was of the form $g_i\{s_i\} = P_1 + \cdots + P_k$. Therefore, it follows that the state of g_i remains unchanged after a synchronous update step is applied to $s_1 \dots s_n$. □

Theorem 9. (Completeness) Let $S_1, S_2 \in D(g_1) \times \cdots \times D(g_n)$ be global states in MV such that $S_1 \xrightarrow{Syn} S_2$. Then $\phi(S_1) \Longrightarrow \phi(S_2)$ in one application of `next` in $RL_S(MV)$.

Proof. Let $S_1 = s_1 \dots s_n$, $S_2 = s_1' \dots s_n' \in D(g_1) \times \cdots \times D(g_n)$ and suppose that the synchronous update step $s_1 \dots s_n \xrightarrow{Syn} s_1' \dots s_n'$ can occur. Then for each $i = 1,\dots,n$ there are two cases to consider:

Case 1: Suppose $s_i \neq s_i'$. Then the state of g_i changes from s_i to s_i' during the synchronous step and so the next–state equation applicable for g_i must have been of the form $g\{s_i'\} = P_1 + \cdots + P_k$, for some $k > 0$ and product terms P_j, $j = 1,\dots,k$. In particular, at least one of the product terms, say P_j, for some $j \in \{1,\dots,k\}$, must be true. Then by definition of $RL_S(MV)$ there must be a rewrite rule

$$g_{E_1}(s_{E_1},v_1)\ \dots\ g_{E_m}(s_{E_m},v_m)\ g_i(s_i,s_i) \Longrightarrow g_{E_1}(s_{E_1},v_1)\ \dots\ g_{E_m}(s_{E_m},v_m)\ \ g_i(s_i,s_i')$$

that was derived from the equation above based on P_j. Clearly, this rewrite rule will be applicable in the global state term $\phi(S_1)$ and so by definition of `next` the entity term $g_i(s_i, s_i)$ must be updated to $g_i(s_i', s_i')$ as required.

Case 2: Suppose $s_i = s_i'$. Then the state of entity g_i is not changed in the update step meaning that the only next–state equation applicable for g_i was of the form $g_i\{s_i\} = P_1 + \cdots + P_k$. Therefore, it follows by definition that no rewrite rule exists in $RL_S(MV)$ that can be applied to the global state term $\phi(S_1)$ and so the entity term $g_i(s_i, s_i)$ remains unchanged from $\phi(S_1)$ to $\phi(S_2)$ as required. $\square$

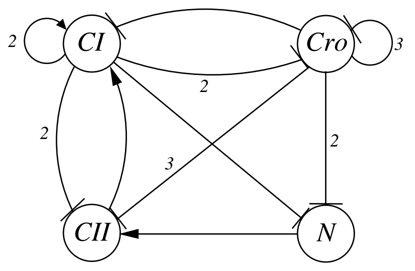
4.4 Case Study

In this section we build on the results of the previous section by developing an RL model for synchronous MVNs. The challenge here is to be able to coordinate update steps and we make use of Maude's metalevel capabilities to achieve this. We again show that the resulting model construction is formally correct and illustrate the developed RL framework for synchronous MVNs with a case study. We illustrate the RL framework developed above by presenting a case study In this section we illustrate the RL framework developed for synchronous MVNs by considering an existing MVN model for the genetic regulatory network controlling the lysis–lysogeny switch in the bacteriophage λ [25, 4]. This example extends the simple two entity model introduced in Section 2.3 to a more detailed four entity model [25].

The temperate bacteriophage λ is a virus infects the bacteria *Escherichia coli* [3, 31]. Interestingly, after infecting a host cell λ makes a decision based on environmental factors whether to enter the *lytic* cycle and *lysogenic* cycles [3]. In most cases, λ enters the *lytic cycle*, where it generates as many new viral particles as the host cell resources allow before producing an enzyme to lyse the cell wall, releasing the new phage into the environment. Alternatively, the λ DNA may integrate into the host DNA and enter the *lysogenic cycle*. Importantly, genes expressed in the λ DNA synthesize a repressor which blocks expression of other phage genes including those involved in its own excision. As such, the host cell establishes an immunity to external infection from other phages, and the phage λ is able to lie dormant, replicating with each subsequent cell division of the host.

4.4.1 The Model

An MVN *PL4* of the resulting regulatory network underlying the lysis–lysogeny switch is presented in Figure 4.5. Note that a truth table entry that has multiple values for an entity means that the output is the same for all such values. For example: when *CI* is 0, *Cro* can be either 0 or 1 for *N* to become 1.



CI	*Cro*	[*N*]
0	0,1	1
0	2,3	0
1,2	0,1,2,3	0

CI	*Cro*	*CII*	[*CI*]
0,1	1,2,3	0	0
0	0,1,2,3	1	1
0	0	0	1
1	0,1,2,3	1	2
2	0,1,2,3	0,1	2
1	0	0	2

CI	*Cro*	[*Cro*]
0,1	0	1
0,1	1	2
0,1	2	3
0,1,2	3	2
2	0,1	0
2	2	1

CI	*Cro*	*N*	[*CII*]
0,1,2	0,1,2,3	0	0
0,1	0,1,2	1	1
0	3	1	0
1	3	1	0
2	0,1,2,3	1	0

Fig. 4.5 An extended MVN model *PL4* of the control mechanism for the lysis-lysogeny switch in bacteriophage λ (based on [25]).

This MVN extends *PL2* [25] and contains four entities: *N*, with $D(N) = \{0,1\}$, which promotes *CII* expression; *CII*, with $D(CII) = \{0,1\}$, which activates *CI*; *CI*, with $D(CI) = \{0,\ldots,2\}$, a repressor which is expressed in the lysogenic cycle; and *Cro*, with $D(Cro) = \{0,\ldots,3\}$, a repressor present in the lytic cycle. This MVN has a global state space consisting of 48 states and has two attractor cycles: $0003, 0002, 0003, \ldots$ which corresponds to the lytic cycle; and $0020, 0020, \ldots$ which corresponds to the lysogenic cycle.

4.4.2 Constructing the RL Model

Applying our RL translation approach detailed above we begin by deriving the following simplified equations for *PL4* from the truth tables given in Figure 4.5:

$$N\{0\} = CI\{0\}Cro\{2,3\} \ + \ CI\{1,2\}$$
$$N\{1\} = CI\{0\}Cro\{0,1\}$$

$$CII\{0\} = N\{0\} + Cro\{3\} + CI\{2\}$$
$$CII\{1\} = N\{1\}CI\{0,1\}Cro\{0,1,2\}$$

$$CI\{0\} = CII\{0\}CI\{0,1\}Cro\{1,2,3\}$$
$$CI\{1\} = CII\{1\}CI\{0\} \ + \ CII\{0\}CI\{0\}Cro\{0\}$$
$$CI\{2\} = CII\{1\}CI\{1\} \ + \ CII\{0,1\}CI\{2\} \ + \ CII\{0\}CI\{1\}Cro\{0\}$$
$$Cro\{0\} = CI\{2\}Cro\{0,1\}$$
$$Cro\{1\} = CI\{0,1\}Cro\{0\} \ + \ CI\{2\}Cro\{2\}$$
$$Cro\{2\} = CI\{0,1\}Cro\{1\} \ + \ Cro\{3\}$$
$$Cro\{3\} = CI\{0,1\}Cro\{2\}$$

From these equations we are then able to derive the rewrite rules required in $RL_S(PL4)$ to model the synchronous behaviour of *PL4*. To illustrate further this approach we present below the set of rewrite rules derived for this network's entities:

To illustrate further this approach we present below the rewrite rules derived from the previous set of equations:

```
rl [N0] : N(1,1) CI(0,s2) Cro(2,s3) => N(1,0) CI(0,s2) Cro(2,s3) .
rl [N0] : N(1,1) CI(0,s2) Cro(3,s3) => N(1,0) CI(0,s2) Cro(3,s3) .
rl [N0] : N(1,1) CI(1,s2) => N(1,0) CI(1,s2) .
rl [N0] : N(1,1) CI(2,s2) => N(1,0) CI(2,s2) .
rl [N1] : N(0,0) CI(0,s2) Cro(0,s3) => N(0,1) CI(0,s2) Cro(0,s3) .
rl [N1] : N(0,0) CI(0,s2) Cro(1,s3) => N(0,1) CI(0,s2) Cro(1,s3) .
rl [CII0] : N(0,s0) CII(1,1) => N(0,s0) CII(1,0) .
rl [CII0] : CII(1,1) Cro(3,s3) => CII(1,0) Cro(3,s3) .
rl [CII0] : CII(1,1) CI(2,s2) => CII(1,0) CI(2,s2) .
rl [CII1] : N(1,s0) CII(0,0) CI(0,s2) Cro(0,s3) =>
N(1,s0) CII(0,1) CI(0,s2) Cro(0,s3) .
rl [CII1] : N(1,s0) CII(0,0) CI(0,s2) Cro(1,s3) =>
N(1,s0) CII(0,1) CI(0,s2) Cro(1,s3) .
rl [CII1] : N(1,s0) CII(0,0) CI(0,s2) Cro(2,s3) =>
N(1,s0) CII(0,1) CI(0,s2) Cro(2,s3) .
rl [CII1] : N(1,s0) CII(0,0) CI(1,s2) Cro(0,s3) =>
N(1,s0) CII(0,1) CI(1,s2) Cro(0,s3) .
rl [CII1] : N(1,s0) CII(0,0) CI(1,s2) Cro(1,s3) =>
N(1,s0) CII(0,1) CI(1,s2) Cro(1,s3) .
rl [CII1] : N(1,s0) CII(0,0) CI(1,s2) Cro(2,s3) =>
N(1,s0) CII(0,1) CI(1,s2) Cro(2,s3) .

rl [CI0] : CII(0,s1) CI(1,1) Cro(1,S3) => CII(0,s1) CI(1,0) Cro(1,S3) .
rl [CI0] : CII(0,s1) CI(1,1) Cro(2,S3) => CII(0,s1) CI(1,0) Cro(2,S3) .
rl [CI0] : CII(0,s1) CI(1,1) Cro(3,S3) => CII(0,s1) CI(1,0) Cro(3,S3) .
rl [CI1] : CII(1,s1) CI(0,0) => CII(1,s1) CI(0,1) .
rl [CI1] : CII(0,s1) CI(0,0) Cro(0,s3) => CII(0,s1) CI(0,1) Cro(0,s3) .
rl [CI2] : CII(1,s1) CI(1,1) => CII(1,s1) CI(1,2) .
rl [CI2] : CII(0,s1) CI(1,1) Cro(0,s3) => CII(0,s1) CI(1,2) Cro(0,s3) .

rl [Cro0] : CI(2,s2) Cro(1,1) => CI(2,s2) Cro(1,0) .
rl [Cro1] : CI(0,s2) Cro(0,0) => CI(0,s2) Cro(0,1) .
rl [Cro1] : CI(1,s2) Cro(0,0) => CI(1,s2) Cro(0,1) .
rl [Cro1] : CI(2,s2) Cro(2,2) => CI(2,s2) Cro(2,1) .
rl [Cro2] : CI(0,s2) Cro(1,1) => CI(0,s2) Cro(1,2) .
rl [Cro2] : CI(1,s2) Cro(1,1) => CI(1,s2) Cro(1,2) .
rl [Cro2] : Cro(3,3) => Cro(3,2) .
rl [Cro3] : CI(0,s2) Cro(2,2) => CI(0,s2) Cro(2,3) .
rl [Cro3] : CI(1,s2) Cro(2,2) => CI(1,s2) Cro(2,3) .
```

Note that for some equation parts there was more than a single rewrite rule being produced. For example: there was a single equation for $CII\{1\}$, but with the value of *CII* being missing, *CI* having two possible values and *Cro* having 3, that equation ended up producing 6 rewrite rules to cover all possible combinations of those entities' values.

4.4.3 Analysis in Maude

The above model can now be formally investigated using the range of powerful analysis tools provided by Maude. We start with a simple `rewrite` command for state 1023:

```
Maude> rew N(1,1) CII(0,0) CI(2,2) Cro(3,3) .
rewrite in SWITCH : N(1, 1) CII(0, 0) CI(2, 2) Cro(3, 3) .
rewrites: 2 in 0ms cpu (0ms real) (29850 rewrites/second)
result GState: N(1, 0) CII(0, 0) CI(2, 2) Cro(3, 2)
```

Maude performs 2 rewrites returning the state 0022 as a result. We can also instruct Maude to perform a certain number of rewrites. For example: let's say we want Maude to rewrite the state 1103 three times, we can do that using the following instruction:

```
Maude> rew[3] N(1,1) CII(1,1) CI(0,0) Cro(3,3) .
rewrite [3] in SWITCH : N(1, 1) CII(1, 1) CI(0, 0) Cro(3, 3) .
rewrites: 3 in 0ms cpu (0ms real) (1500000 rewrites/second)
result GState: N(1, 0) CII(1, 0) CI(0, 1) Cro(3, 3)
```

Maude performs the three rewrites and returns the resulting state 0013.

We now introduce some searches at the metalevel similar to those illustrated in Section 3.4. For example, the following search checks whether *CI* and *CII* can be simultaneously in state 1 starting from initial state 0000:

```
search '__['N['0.D1,'0.D1],'CII['0.D1,'0.D1],'CI['0.D2,'0.D2],
'Cro['0.D3,'0.D3]] =>+ '__['CI['1.D2,'1.D2],'CII['1.D1,'1.D1],
T1:Term,T2:Term] .
```

Maude returns that this search has no solutions showing that the property doesn't hold. A similar search using initial state 1001 does hold and this indicates there is one solution that results in state 0113. Using the same starting state we can check whether *Cro* ever reaches state 3 as follows:

```
search '__['N['0.D1,'0.D1],'CII['0.D1,'0.D1],'CI['0.D2,'0.D2],
'Cro['0.D3,'0.D3]] =>+ '__[T1:Term,T2:Term,'Cro['3.D3,'3.D3],T3:Term] .
```

Maude returns two solutions in states 3 and 5 of the trace:

```
Solution 1 (state 3)
states: 4  rewrites: 70 in 8ms cpu (9ms real) (7961 rewrites/second)
T1 --> 'CI['1.D2,'1.D2]
T2 --> 'CII['0.D1,'0.D1]
T3 --> 'N['0.D1,'0.D1]

Solution 2 (state 5)
states: 6  rewrites: 113 in 11ms cpu (11ms real) (10091 rewrites/second)
T1 --> 'CI['0.D2,'0.D2]
T2 --> 'CII['0.D1,'0.D1]
T3 --> 'N['0.D1,'0.D1]

No more solutions.
states: 6  rewrites: 134 in 12ms cpu (13ms real) (10940 rewrites/second)
```

We can check the trace of the search using the command `show path 5` and Maude would return the states visited during the search:

```
Maude> show path 5 .
state 0, GroundTerm: '__['N['0.D1,'0.D1],'CII['0.D1,'0.D1],
'CI['0.D2,'0.D2],'Cro['0.D3,'0.D3]]===[ rl '__[T1,T2,T3,T4] =>
next('__[T1,T2,T3,T4]) [label step] . ]===>
state 1, GroundTerm: '__['CI['1.D2,'1.D2],'CII['0.D1,'0.D1],
'Cro['1.D3,'1.D3],'N['1.D1,'1.D1]] ===[ rl '__[T1,T2,T3,T4] =>
next('__[T1,T2,T3,T4]) [label step] . ]===>
state 2, GroundTerm: '__['CI['0.D2,'0.D2],'CII['1.D1,'1.D1],
'Cro['2.D3,'2.D3],'N['0.D1,'0.D1]]===[ rl '__[T1,T2,T3,T4] =>
next('__[T1,T2,T3,T4]) [label step] . ]===>
state 3, GroundTerm: '__['CI['1.D2,'1.D2],'CII['0.D1,'0.D1],
'Cro['3.D3,'3.D3],'N['0.D1,'0.D1]]===[ rl '__[T1,T2,T3,T4] =>
next('__[T1,T2,T3,T4]) [label step] . ]===>
state 4, GroundTerm: '__['CI['0.D2,'0.D2] ,'CII['0.D1,'0.D1],
'Cro['2.D3,'2.D3],'N['0.D1,'0.D1]]===[ rl '__[T1,T2,T3,T4] =>
next('__[T1,T2,T3,T4]) [label step] . ]===>
state 5, GroundTerm: '__['CI['0.D2,'0.D2],'CII['0.D1,'0.D1],
'Cro['3.D3,'3.D3],'N['0.D1,'0.D1]]
```

We can check if a state is reachable from a certain starting state. For example: we can check if the state 0020 is reachable starting from state 1000:

```
search '__['N['1.D1,'1.D1],'CII['0.D1,'0.D1],'CI['0.D2,'0.D2],
'Cro['0.D3,'0.D3]] =>+ '__['CI['2.D2,'2.D2],'CII['0.D1,'0.D1],
'Cro['0.D3,'0.D3],'N['0.D1,'0.D1]] .
```

Maude returns a single solution in state 4:

```
state 4, GroundTerm: '__['CI['2.D2,'2.D2],'CII['0.D1,'0.D1],
'Cro['0.D3,'0.D3],'N['0.D1,'0.D1]]
```

Identifying the point attractors for a model is an important part of the analysis. This was straightforward in the asynchronous case since point attractors represented deadlocked global states. However, in the synchronous case all traces are infinite and so additional work is needed. One way to address this is to introduce a Boolean operator `repState` at the metalevel that indicates if a global state remains unchanged after a synchronous update step. (Note that this again illustrates the expressiveness provided by Maude's metalevel capabilities). This can be defined equationally as follows:

```
op same : Term Term -> Bool .
eq same('__[T1,T2,T3,T4],'__[T1,T2,T3,T4]) = true .
eq same(T1,T2) = false [owise] .

op repState : Term -> Bool .
eq repState(T) = same(T,next(T)) .
```

where `same` is an equationally defined function that returns true only if two metalevel state terms contain the same entities in the same states. We can make use of this function We can then use this together with the `search` command to find point attractors. For example, the following search shows that state 1010 will eventually enter the point attractor $0020, 0020, \ldots$ (note the use of `such that` to place a condition on the result of the search).

```
search '__['N['1.D1,'1.D1],'CII['0.D1,'0.D1],'CI['1.D2,'1.D2],
'Cro['0.D3,'0.D3]] =>+ T:Term such that repState(T) .
```

Performing the same search again starting from state results in no solution as the trace does not end in a point attractor:

```
search '__['N['0.D1,'0.D1],'CII['0.D1,'0.D1],'CI['0.D2,'0.D2],
'Cro['0.D3,'0.D3]] =>+ T:Term such that repState(T) .
```

This returns no solution which is correct.

4.4.4 LTL Model Checking

The above examples give an initial idea of the wide range of analysis checks possible of our RL model. Further analysis of *PL4* can be done by again utilizing the LTL model checker provided by Maude. This can be applied in a similar way to that illustrated in Section 3.4.4 but in this case it will work at the metalevel.

To illustrate this, consider checking the hypothesis that when *CI* becomes permanently inactive then *Cro* must continually be able to reach full activation. This can be checked using the following model checking instruction:

```
red modelCheck('__['CI['1.D2,'1.D2],'CII['1.D1,'1.D1],'Cro['2.D3,'2.D3],
'N['0.D1,'0.D1]], <> [] atCI('0) -> []<> atCro('3)) .
```

This holds for the given initial state 0112, and further tests indicate this is a potential invariant.

We can check the hypothesis that starting with *Cro* at full activation then at some point ending up with *CI* always at state 1:

```
Maude> red modelCheck('__['CI['0.D2,'0.D2],'CII['0.D1,'0.D1],
'Cro['3.D3,'3.D3],'N['0.D1,'0.D1]], [] (<> [] atCI(1))) .
reduce in SWITCH-CHECK : modelCheck('__['CI['0.D2,'0.D2]
,'CII['0.D1,'0.D1],'Cro['3.D3,'3.D3],'N['0.D1,'0.D1]], []<> []atCI(1)) .
rewrites: 55 in 3ms cpu (5ms real) (16917 rewrites/second)
result ModelCheckResult: counterexample(nil, {'__['CI['0.D2,'0.D2],
'CII['0.D1,'0.D1],'Cro['3.D3,'3.D3],'N['0.D1,'0.D1]],'step}
{'__['CI['0.D2,'0.D2],'CII['0.D1,'0.D1],'Cro['2.D3,'2.D3],
'N['0.D1,'0.D1]],'step})
```

Maude returns a counter example showing that the property does not hold for starting state 0030. The counter example is a trace that starts from the initial state 0030 and stops when the hypothesis no longer holds (in this case, the trace stops at state 0020 with *Cro* going back to state 2).

We can check the hypothesis that if we eventually have *CI* always at 0, then its always possible to have *Cro* at 3 (Maude returns true in this case):

```
Maude> red modelCheck('__['CI['1.D2,'1.D2],'CII['1.D1,'1.D1],
'Cro['2.D3,'2.D3],'N['0.D1,'0.D1]], <> [] atCI('0) -> []<> atCro('3)) .
```

```
reduce in SWITCH-CHECK : modelCheck('__['CI['1.D2,'1.D2],
'CII['1.D1,'1.D1],'Cro['2.D3,'2.D3],'N['0.D1,'0.D1]], <> []atCI('0) ->
[]<> atCro('3)) .
rewrites: 135 in 9ms cpu (10ms real) (14475 rewrites/second)
result Bool: true
```

We can check whether *Cro* can always eventually be at state 3 starting from state 0000:

```
Maude> red modelCheck('__['CI['0.D2,'0.D2],'CII['0.D1,'0.D1],
'Cro['0.D3,'0.D3],'N['0.D1,'0.D1]], []<> atCro('3)) .
reduce in SWITCH-CHECK : modelCheck('__['CI['0.D2,'0.D2],
'CII['0.D1,'0.D1],'Cro['0.D3,'0.D3],'N['0.D1,'0.D1]],
[]<> atCro('3)) .
rewrites: 147 in 13ms cpu (14ms real) (10925 rewrites/second)
result Bool: true
```

We can further extend our analysis capabilities by developing our own metalevel operators which can be used to define interesting atomic propositions. As an example, consider using the metalevel Boolean operator `repState` to define an atomic proposition `rept : -> Prop` as follows:

```
eq  T |= rept = repState(T) .
```

where T is a variable of type *Term*. This atomic proposition can the be used to form interesting LTL formulas for model checking. For example, suppose we want to check whether *N* and *CII* becoming simultaneously active is a predictor for a point attractor. The following model checking instruction shows the property is true for initial state 1000.

```
red modelCheck('__['CI['0.D2,'0.D2],'CII['0.D1,'0.D1],
'Cro['0.D3,'0.D3],'N['0.D1,'0.D1]],
(<> (atN('1) /\ atCII('1))) -> <> rept) .
```

Checking a further sample of initial states shows that the property is potentially an invariant of the model.

4.5 Conclusions

Following on chapter 3, in this chapter we aimed to strengthen the tool support available for MVNs by linking synchronous MVNs to RL and so enabling the application of the support tools available for RL. The challenge here lied in the ability to coordinate update steps, we made use of rewriting strategies implemented using Maude's metalevel capabilities to achieve this using a two phase update protocol. We define a rewriting strategy to apply a system synchronous update step which is defined in two parts: a metalevel operation applying the synchronous rewrite rules,then a metalevel operation that resets the current state after applying the first phase. We developed an RL model for synchronous MVNs and within that we represented the next state function associated with each entity and the associated synchronous update rules.

We provided a formal correctness argument for this translation approach where we formally showed that our translation from an asynchronous MVN to an RL model given in the previous section is correct. That was done by showing and proving the soundness and completeness of our translation process. To do this we show that: 1) (*soundness*) each global state transition possible in our RL model represents a corresponding asynchronous update in the original MVN; and 2) (*completeness*) every asynchronous update possible in an MVN is specified in our RL model.

We illustrated the RL framework developed by presenting a case study based on an existing MVN model for the genetic regulatory network controlling the lysis–lysogeny switch in the bacteriophage λ [25, 4]. The case study helped motivate our RL framework by illustrating the analysis techniques and tools when using RL and the tool *Maude*. We formally investigated the model using a range of powerful analysis tools provided by Maude, which was one of the motivations for choosing to use RL. We started our analysis using basic rewrite commands before we moved onto making use of the `search` command and performed some basic model checking for certain properties in Section 4.4.3. We then extended our analysis by utilizing the LTL model checker provided by Maude in Section 4.4.4 where we showed how that is done at the metalevel in synchronous MVNs and how certain hypothesis can be proved, and we showed how Maude provides us with a counter example for hypothesis that do not hold.

A prototype tool has been developed that given an XML description of an MVN will automatically translate a model into an RL model. We can then use that module to perform some analysis using Maude and LTL. In order to illustrate the practical application of tools and techniques developed in this chapter, we carry out a larger case study in Chapter 6 by introducing models of sizes 13 and 22 using a scalable test model based on the gene regulatory network of the segment polarity gene family which is at the basis of *Drosophila* embryonic development.

Again, one important aspect is the performance of the techniques developed in this chapter. We use a scalable test model to investigate that in Chapter 5 where we start with a multi-valued network of five entities. Using the developed techniques and tools we start by presenting an asynchronous version of a scalable test model, we perform a range of analysis using the Maude's search command and the LTL model checker. We then introduce the synchronous version of the model alongside an analysis of its behaviour. We perform some LTL model checking before we scale the model in increments of five and test the performance of our techniques and the RL framework before we provide some analysis summary tables.

Chapter 5

Performance Evaluation

5.1 Introduction

The case studies presented in Section 3.4.1 and Section 4.4 provide a good illustration of the practical application of the RL techniques we have developed. However, they provide little indication of how the developed RL approach would scale when applied to larger MVN models and what impact the well–known state space explosion problem would have. In this section we set out to address this by investigating how the our RL framework performs as the MVN size (i.e. number of entities) increases. The approach we take is to define an artificial, scalable test MVN and then use this to produce an incremental set of test models. Note that the ease with which this scalable test model can be implemented in our RL framework illustrates how versatile it is.

We start by introducing our performance test model using a basic model ABCDI which consists four Boolean entities A,C,D and I and one multi-valued entity B with a state space of size 3 (0, 1 or 2). We then give a brief analysis of the ABCDI model behaviour. We then explain the approach we took in scaling the model and discuss our testing approach in detail and give a summary of the different model checking tests applied to test the performance of our RL framework. Using the developed techniques and tool support we present both asynchronous and synchronous versions of the basic model of five entities and perform a range of analysis using our testing approach where we start by performing three test searches as well as two LTL model checking commands using LTL formulas. We then extend our model in increments of 5 entities and perform the same set of test searches and LTL formulas to test the performance of our RL framework as the size of our model increases. We provide summary tables for the different sizes of our scalable test model showing the results of our test searches as well as introduce two graphs summarizing the performance of our RL framework with this model in terms of time and the number of rewrites performed.

This chapter is organised as follows, in Section 5.2 we present our basic test model of 5 entities and explain the approach we took scaling the model. We then introduce our testing approach in Section 5.3. We introduce the asynchronous version of our model in Section 5.4 and produce a set of test models in increments of 5 entities. We test each model using Maude's model checking tools (both search command and the LTL model checker). In Section 5.5 we introduce the synchronous version of our model and analyse it using a similar approach to the asynchronous version. Finally, we conclude the chapter by providing some reflection on what was done in Section 5.6.

5.2 Performance Test Model

In order to allow a range of model sizes to be considered we define an artificial, scalable MVN model *ABCDI*, that is a multi-valued network of 5 entities which contains four Boolean entities *A*,*C*,*D* and *I* and one multi-valued entity *B* with a state space of size 3 (0, 1 or 2). Note that the entity *I* can be viewed as simply an input entity that remains in its initial state and is used to introduce different behaviour in a block when connecting to a neighbouring block of entities. The idea is that *ABCDI* can be used as a basic building block and instances of these basic blocks can be composed to make models of size 10, 15, 20, 25, etc. The basic building block MVN *ABCDI* is presented in Figure 5.1 with the following equations representing the behaviour of the model:

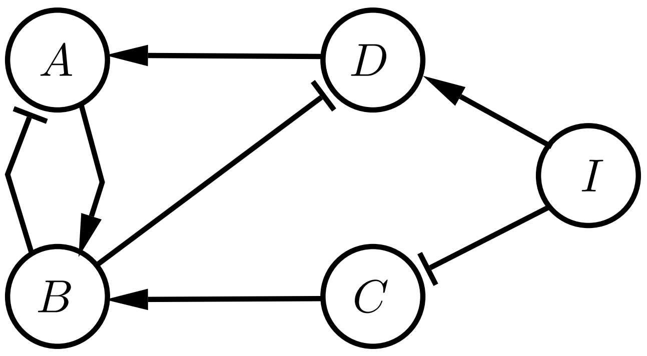


Fig. 5.1 An MVN model *ABCDI* which is used as a basic building block to construct models of increasing size.

$$\begin{aligned} A\{0\} &= B\{0\}D\{0\} + B\{1,2\} \\ A\{1\} &= B\{0\}D\{1\} \end{aligned}$$

$$\begin{aligned} B\{0\} &= A\{0\}C\{0\} \\ B\{1\} &= A\{0\}C\{1\} + A\{1\}C\{0\} \\ B\{2\} &= A\{1\}C\{1\} \end{aligned}$$

$$\begin{aligned} C\{0\} &= I\{1\} \\ C\{1\} &= I\{0\} \end{aligned}$$

$$\begin{aligned} D\{0\} &= I\{0\} + B\{2\} \\ D\{1\} &= B\{0,1\}I\{1\} \end{aligned}$$

$$\begin{aligned} I\{0\} &= I\{0\} \\ I\{1\} &= I\{1\} \end{aligned}$$

This model has a state space of 48 global states and 2 attractors, a point attractor in state 01100, and one of size 4: $00011- > 10011- > 11011- > 01011- > 00011$. The state graph for the basic *ABCDI* model of 5 entities is given in Figure 5.2.

To allow multiple instances of this model to be easily composed we represent each entity as a family of entities. This is straightforward to do in RL by associating a natural number parameter to each entity as the following Maude excerpt shows for the asynchronous case:

```
ops A C D I : Nat D1 -> Entity .
op  B : Nat D2 -> Entity .
```

We can then have general rewrite rules for the MVN as the following rules modelling the asynchronous behaviour of entity *A* illustrate (where `i` is the index of the block):

```
rl [A0] :  A(i,1) B(i,0) D(i,0) => A(i,0) B(i,0) D(i,0) .
rl [A0] :  A(i,1) B(i,1) => A(i,0) B(i,1) .
rl [A0] :  A(i,1) B(i,2) => A(i,0) B(i,2) .
rl [A1] :  A(i,0) B(i,0) D(i,1) => A(i,1) B(i,0) D(i,1) .
```

The full list of rewrite rules for both asynchronous and synchronous version of this scalable model is given in Sections 5.4 and 5.5, respectively.

To compose instances of *ABCDI* together we simply link them by allowing one instance to influence the state of entity *I* in the other instance. For example, to create a model of size 10 we could compose two instances, referred to as instance 1 and 2, by having entity *A* in instance 2 activating entity *I* in instance 1. This is straightforward to do by adding the following rules for *I* in instance 1:

```
rl [I10]:  I(1,1) A(2,0) => I(1,0) A(2,0) .
rl [I11]:  I(1,0) A(2,1) => I(1,1) A(2,1) .
```

Using the test model generation approach presented above we were able to create a series of MVN test models in incremental steps of 5 entities as depicted in Figure 5.3. For every new test model we add, we connect an entity of the new model to the entity `I` in the previous test model. We alternate between activation and inhibition for the rewrite rules of `I` to add complexity and introduce different behaviours as the model scales. For example: when adding the second test model, `I(1)` takes the value of `A(2)` as an input, and when we add the third test model, `I(2)` takes the value of `B(3)` as an input and so on. We now give the rewrite rules for the first four instances of `I` as follows:

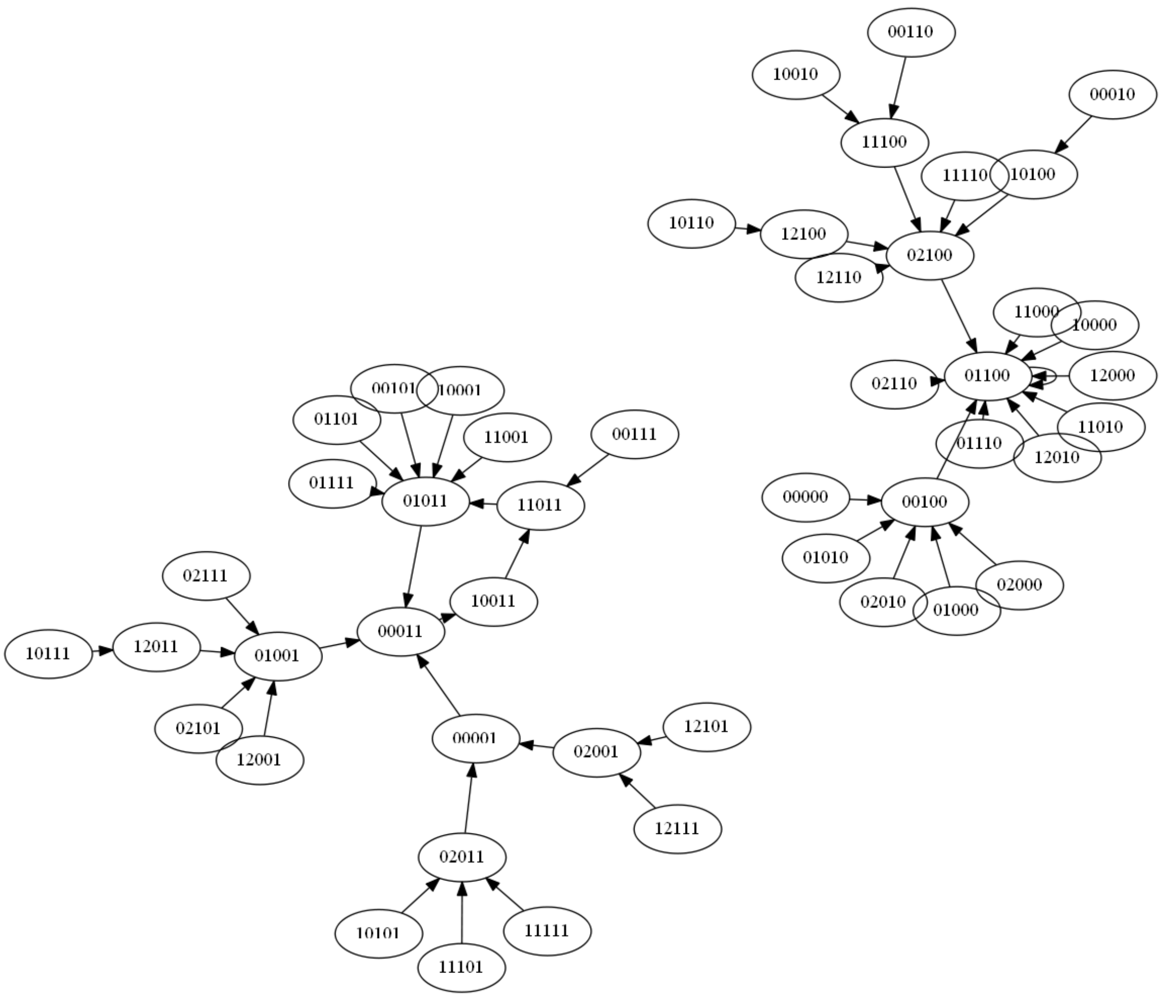


Fig. 5.2 The State Graph of MVN model *ABCDI* which is used as a basic building block to construct models of increasing size.

```
rl [I10]:  I(1,1) A(2,0) => I(1,0) A(2,0) .

rl [I11]:  I(1,0) A(2,1) => I(1,1) A(2,1) .


rl [I20]:  I(2,1) B(3,1) => I(2,0) B(3,1) .

rl [I21]:  I(2,0) B(3,0) => I(2,1) B(3,0) .


rl [I30]:  I(3,1) C(4,0) => I(3,0) C(4,0) .

rl [I31]:  I(3,0) C(4,1) => I(3,1) C(4,1) .
rl [I40]:  I(4,1) D(5,1) => I(4,0) D(5,1) .
rl [I41]:  I(4,0) D(5,0) => I(4,1) D(5,0) .
```

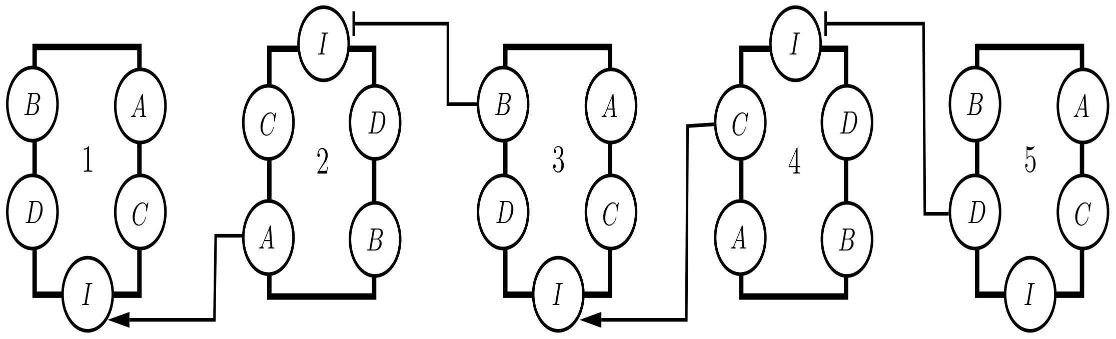


Fig. 5.3 A pictorial representation of the five test MVN models constructed incrementally from the basic building block MVN given in Figure 5.1. Note that the arcs ending in arrows represent activation whereas the arcs ending in bars represent inhibition.

5.3 Testing Approach

The scalable *ABCDI* model (both the synchronous and asynchronous versions) was used to test the performance of our RL framework. The two key performance measures used were the number of explored states and the command run time. Our testing approach consisted of using Maude's search command to perform three search tests and two LTL model checking commands using predicates. For each test model we used three simple types of test searches to see how the model performed:

1) Search for a point attractor.
2) Search for a state in which all input places become inactive.
3) Check whether the entities could all go from being active to being inactive.

We also used LTL model checking to check the following properties:
1) Starting off with the first instance of `B` at full activation then at some point ending up with the last instance of `A` always at state 1.

2) The last instance of `C` can always eventually be at state 1 starting from an initial state where all instances of `A` are at state 0.

We use these test searches to compare the performance results for our scalable test model based on time in milliseconds and the number of rewrites performed. The tests were carried out using Maude on a computer with a 2.7 GHz Intel Core i5 processor and containing 16 GB 1333 MHz DDR3 memory.

5.4 Asynchronous Model Performance Evaluation

We start by considering the asynchronous version of the basic *ABCDI* model of size 5. We present the full set of rewrite rules for the asynchronous version of our scalable test model as follows:

```
rl [Ai0]: A(i,1)B(i,1) => A(i,0)B(i,1) .
rl [Ai0]: A(i,1)B(i,1)D(i,1) => A(i,0)B(i,1)D(i,1) .
rl [Ai0]: A(i,1)B(i,2) => A(i,0)B(i,2) .
rl [Ai1]: A(i,0)B(i,0)D(i,1) => A(i,1)B(i,0)D(i,1) .

rl [Bi0]: B(i,1)A(i,0)C(i,0) => B(i,0)A(i,0)C(i,0) .
rl [Bi0]: B(i,2)A(i,0)C(i,0) => B(i,0)A(i,0)C(i,0) .
rl [Bi1]: B(i,0)A(i,0)C(i,1) => B(i,1)A(i,0)C(i,1) .
rl [Bi1]: B(i,0)A(i,1)C(i,0) => B(i,1)A(i,1)C(i,0) .
rl [Bi2]: B(i,1)A(i,1)C(i,1) => B(i,2)A(i,1)C(i,1) .

rl [Ci0]: C(i,1)I(i,1) => C(i,0)I(i,1) .
rl [Ci1]: C(i,0)I(i,0) => C(i,1)I(i,0) .

rl [Di0]: D(i,1)B(i,2) => D(i,0)B(i,2) .
rl [Di0]: D(i,1)I(i,0) => D(i,0)I(i,0) .
rl [Di1]: D(i,0)B(i,0)I(i,1) => D(i,1)B(i,0)I(i,1) .
rl [Di1]: D(i,0)B(i,1)I(i,1) => D(i,1)B(i,1)I(i,1) .
```

With these rewrite rules we now have the set of rewrite rules we need to perform our analysis for this model (note that entity `I` is acting as an input entity and therefore does not have any rewrite rules as its value is never updated).

5.4.1 Basic Model Analysis

We start our analysis for the basic model of five entities using the three types of test searches that were introduced in Section 5.3 to see how the model performed:

1) Searching for a point attractor starting from an initial state where all entities are at their highest possible value:

```
search A(0,1) B(0,2) C(0,1) D(0,1) I(0,1) =>! A(0,tA:D1) B(0,tB:D2)
C(0,tC:D1) D(0,tD:D1) I(0,tI:D1) .
```

Maude performs 30 rewrites taking less than a millisecond to visit 17 states.

2) For our second search we start at an initial state where all entities have the value 1 and we check for the possibility of entity `I` going back to state 0:

```
search A(0,1) B(0,1) C(0,1) D(0,1) I(0,1) =>! I(0,0) Gs:GState .
```

Maude performs 33 rewrites resulting in it not finding a solution after visiting 18 states.

3) For our third test search, we start with an initial state where all entities are at state 0, exploring the possibility of all entities reaching their highest:

```
Maude> search A(0,0) B(0,0) C(0,0) D(0,0) I(0,0) =>!
A(0,1) B(0,2) C(0,1) D(0,1) I(0,1) .
```

Maude performs only two rewrites here exploring 3 states and returns no solution where all entities are at their highest value. We give a summary of results for the three test searches in Table 5.1.

We now perform our LTL checks:

1) Starting off with the first instance of `B` at full activation then at some point ending up with the last instance of `A` always at state 1.

```
Maude> red modelCheck( A(0,0) B(0,2) C(0,1) D(0,1) I(0,1), []
(<> [] hasAat(0,1))).
```

Maude performs 16 rewrites returning a counter example showing that the property does not hold:

```
rewrites: 16 in 0ms cpu (0ms real) (26315 rewrites/second)
result [ModelCheckResult]: counterexample({A(0, 0) B(0, 2) C(0, 1)
D(0, 1)I(0, 1),'Bi1} {A(0, 0) B(0, 1) C(0, 1) D(0, 1) I(0, 1),'Ci0},
{A(0, 0) B(0, 1) C(0, 0) D(0, 1) I(0,1),'Bi0} {A(0, 0) B(0, 0) C(0, 0)
D(0, 1) I(0, 1),'Ai1} {A(0, 1) B(0, 0) C(0, 0) D(0,1) I(0, 1),'Bi1}
{A(0, 1) B(0, 1) C(0, 0) D(0, 1) I(0, 1),'Ai0})
```

2) The last instance of `C` can always eventually be at state 1 starting from an initial state where all instances of `A` are at state 0.

```
Maude> red modelCheck( A(0,0) B(0,1) C(0,1) D(0,1) I(0,1),
[] <> hasCat(0,1)).
```

Maude performs 12 rewrites here returning a counter example showing that the property does not hold:

```
rewrites: 12 in 0ms cpu (0ms real) (55045 rewrites/second)
result [ModelCheckResult]: counterexample({A(0, 0) B(0, 1) C(0, 1)
D(0, 1) I(0, 1),'Ci0}, {A(0, 0) B(0, 1) C(0, 0) D(0, 1) I(0, 1),'Bi0}
{A(0, 0) B(0, 0) C(0, 0) D(0, 1) I(0, 1),'Ai1} {A(0, 1) B(0, 0) C(0, 0)
D(0, 1) I(0, 1),'Bi1} {A(0, 1) B(0, 1) C(0, 0) D(0, 1) I(0, 1),'Ai0})
```

Table 5.1 Summary of test results for performing three simple searches on the series of asynchronous test MVNs for model size 5.

Model Size	5
Rewrites	30
	33
	2
Time (ms)	0
	0
	0

5.4.2 Scaling to 10 Entities

For performance evaluation purposes, our test model can be scaled up by adding instances of A, B, C, D and I, thus doubling the system state space. The same rules for A, B, C and D can be used again by changing the value of i to represent different instances of those entities. The only part of the system that is changing now is the behaviour of the first instance of I, which is used to connect the new 5 entities to the old ones as follows:

```
rl [I10]:  I(1,1) A(2,0) => I(1,0) A(2,0) .
rl [I11]:  I(1,0) A(2,1) => I(1,1) A(2,1) .
```

This connects the second instance of `A(2)` to the first instance of `I(1)`. We now start our analysis using the three types of test searches:

1) Our first test is searching for a point attractor from an initial state where all entities are their highest possible value:

```
search A(0,1) A(1,1) B(0,2) B(1,2) C(0,1) C(1,1) D(0,1) D(1,1) I(0,1)
I(1,1) =>! A(0,tA0:D1) A(1,tA1:D1) B(0,tB0:D2) B(1,tB1:D2)
C(0,tC0:D1) C(1,tC1:D1) D(0,tD0:D1) D(1,tD1:D1) I(0,tI0:D1) I(1,tI1:D1) .
```

The number of rewrites performed has increased from 30 in the 5-entity model to 2472 now in 17 milliseconds.

2) For our second test search, we start with an initial state where all entities are at state 0, exploring the possibility of all entities reaching their highest:

```
Maude> search A(0,0) A(1,0) B(0,0) B(1,0) C(0,0) C(1,0) D(0,0) D(1,0)
I(0,0) I(1,0) =>! A(0,1) A(1,1) B(0,2) B(1,2) C(0,1) C(1,1) D(0,1)
D(1,1) I(0,1) I(1,1) .
```

Maude performs 12 rewrites in one millisecond exploring 9 network states.

3) For our third search test, we start from an initial state where all entities have the value 1 and we search for the possibility of both instances of `I` going back to state 0:

```
search A(0,1) A(1,1) B(0,1) B(1,1) C(0,1) C(1,1) D(0,1) D(1,1) I(0,1)
I(1,1) =>! I(0,0)I(1,0) Gs:GState .
```

The number of rewrites performed has increased from 33 in the 5-entity model to 2632 rewrites now with the search taking 17 milliseconds to finish.

We now give a summary table of the search results for the 3 considered tests in Table 5.2. The table shows a clear jump in both the number of rewrites and the search time which is to be expected after doubling the state space of our model.

We now perform our LTL checks:
1) Starting off with the first instance of `B` at full activation then at some point ending up with the last instance of `A` always at state 1.

```
Maude> red modelCheck( A(0,0) A(1,0) B(0,2) B(0,1) C(0,1)
C(0,1) D(0,1) D(1,1) I(0,1) I(1,1) , [] (<> [] hasAat1(1,1))).
```

Maude now performs 106 rewrites in 2 milliseconds returning a counter example showing that the property does not hold.

2) The last instance of `C` can always eventually be at state 1 starting from an initial state where all instances of `A` are at state 0.

```
Maude> red modelCheck( A(0,0) A(1,0) B(0,2) B(0,1) C(0,1)
C(0,1) D(0,1) D(1,1) I(0,1) I(1,1) , [] <> hasCat(1,1)).
```

Maude performs 104 rewrites here returning a counter example.

Table 5.2 Summary of test results for performing three simple searches on the series of asynchronous test MVNs for model sizes 5 and 10.

Model Size	5	10
Rewrites	30	2472
	33	2632
	2	12
Time (ms)	0	17
	0	17
	0	1

5.4.3 Further Scaling Analysis

We further scaled our model to sizes 15, 20 and 25 and performed some analysis using the Maude search command and our three types of test searches:

1) Searching for a point attractor from an initial state where all entities are their highest possible value in the model of size 15:

```
search A(0,1) A(1,1) A(2,1) B(0,2) B(1,2) B(2,2) C(0,1) C(1,1) C(2,1)
D(0,1) D(1,1) D(2,1) I(0,1) I(1,1) I(2,1)  =>! A(0,tA0:D1) A(1,tA1:D1)
A(2,tA2:D1) B(0,tB0:D2) B(1,tB1:D2) B(2,tB2:D2) C(0,tC0:D1)
C(1,tC1:D1) C(2,tC2:D1) D(0,tD0:D1) D(1,tD1:D1) D(2,tD2:D1)
I(0,tI0:D1) I(1,tI1:D1) I(2,tI2:D1) .
```

The result returned by Maude shows that a number of performed rewrites of 150133 which is a big jump from the model of 10 entities and that result was generated in 1040 milliseconds.

For the model of size 20 those numbers again had a huge jump to 8374315 rewrites in 10 minutes.

2) For our second search test, starting from an initial state where all entities have the value 1 we search for the possibility of all instances of `I` going back to state 0 in the model of size 15:

```
search A(0,1) A(1,1) A(2,1) B(0,1) B(1,1) B(2,1) C(0,1) C(1,1) C(2,1)
D(0,1) D(1,1) D(2,1) I(0,1) I(1,1) I(2,1) =>! I(0,0) I(1,0) I(2,0)
Gs:GState .
```

Maude performs 292228 rewrites in 2234 milliseconds in the 15-entity model. Those numbers jump to 12696152 in 26 minutes for the model of size 20 before Maude fails to produce any results for this search command in the 25-entity model.

3) We now explore the possibility of all entities going back to 0 starting from an initial state where all entities are at their highest value with respect to their state spaces in the model of size 15:

```
search A(0,0) A(1,0) A(2,0) B(0,0) B(1,0) B(2,0) C(0,0) C(1,0) C(2,0)
D(0,0) D(1,0) D(2,0) I(0,0) I(1,0) I(2,0) =>! A(0,1) A(1,1) A(2,1)
B(0,2) B(1,2) B(2,2) C(0,1) C(1,1) C(2,1) D(0,1) D(1,1) D(2,1)
I(0,1) I(1,1) I(2,1) .
```

The result returned by Maude shows a number of performed rewrites of 16092 which is a big jump from the model of 10 entities and that result was generated in 177 milliseconds.

For the model of size 20 those numbers again had a huge jump to 431891 rewrites in 4.72 seconds. A summary of the test results produced is shown in Table 5.3.

We now perform our LTL checks on the model of size 15:

1) Starting off with the first instance of `B` at full activation then at some point ending up with the last instance of `A` always at state 1.

```
Maude> red modelCheck( A(0,0) A(1,0) A(2,0) B(0,2) B(1,1) B(2,1)
C(0,1) C(0,1) C(2,1) D(0,1) D(1,1) D(2,1) I(0,1) I(1,1) I(2,1),
[] (<> [] hasAat1(2,1))) .
```

Maude now performs 184 rewrites in 3 milliseconds before returning the counter example. For the model of size 20, the number of rewrites jumped to 727 in 9 milliseconds before returning the counter example.
2) The last instance of `C` can always eventually be at state 1 starting from an initial state where all instances of `A` are at state 0.

```
Maude> red modelCheck( A(0,0) A(1,0) A(2,0) B(0,2) B(1,1) B(2,1)
C(0,1) C(0,1) C(2,1) D(0,1) D(1,1) D(2,1) I(0,1) I(1,1) I(2,1),
[] <> hasCat(2,1)) .
```

Maude performs 182 rewrites here in 3 milliseconds before returning a counter example. For the model of size 20, the number of rewrites increased to 725 in 10 milliseconds.

Table 5.3 Summary of test results for performing three simple searches on the series of asynchronous test MVNs for Models of sizes 5 to 25.

Model Size	5	10	15	20	25
Rewrites	30	2472	150133	8374315	-
	33	2632	292228	12696152	-
	2	12	16092	431891	-
Time (ms)	0	17	1040	10.01mins	-
	0	17	2234	26mins	-
	0	1	177	4728	-

5.4.4 Evaluating Test Results

We started our analysis for our scalable model with the basic model of 5 entities. Then we extended it repeatedly by adding a new block of 5 entities and performed some tests using the search command to come up with some performance results for our RL framework before we gave a summary of those results in Table 5.3. Figure 5.4 gives a visual representation of the number of rewrites with relation to different model sizes for test searches 1,2 and 3, while Figure 5.5 gives a visual representation of the `log` of search time.

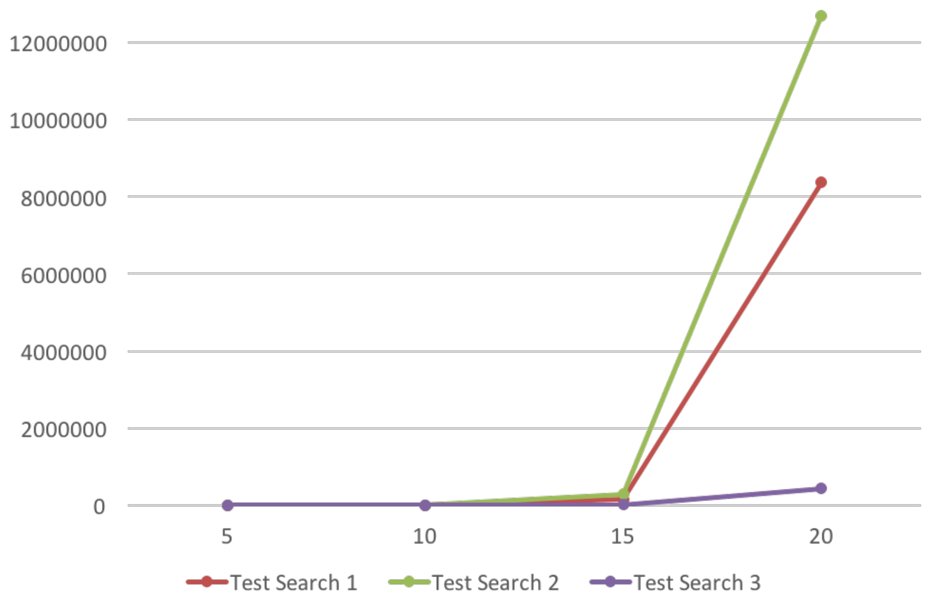


Fig. 5.4 Number of Rewrites in Relation to Model Size for Test Searches 1,2 and 3.

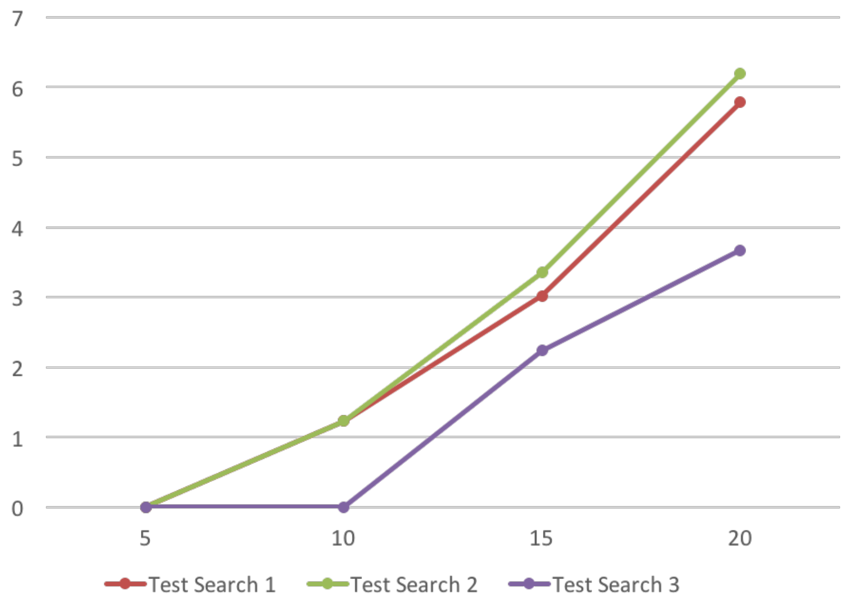


Fig. 5.5 A Representation of the log of Search Time for Test Searches 1,2 and 3.

It can be seen that the largest test model that was able to produce results in the asynchronous case was the model of 20 entities. The test model with 25 entities did not complete the searches even after an extended period. This highlights one of the key limitations of MVNs given the state space explosion problem. In order to allow larger models to be analysed new ways to compositional construct and analyse MVNs are needed and we discuss this further in the conclusions section at the end of this chapter.

5.5 Synchronous Model Performance Evaluation

For the synchronous version of this performance test model, entities now have three values instead of just two; `i` representing the index; a value representing the current state; and another value representing the next state of the entity. Note that we now use the meta-level operators to represent different model states. We now present the full list of synchronous rewrite rules for the module:

```
rl [Ai0]:  A(i,1,1)B(i,1,S2) => A(i,1,0)B(i,1,S2) .
rl [Ai0]:  A(i,1,1)B(i,1,S2)D(i,1,S4) => A(i,1,0)B(i,1,S2)D(i,1,S4) .
rl [Ai0]:  A(i,1,1)B(i,2,S2) => A(i,1,0)B(i,2,S2) .
rl [Ai1]:  A(i,0,0)B(i,0,S2)D(i,1,S4) => A(i,0,1)B(i,0,S2)D(i,1,S4) .

rl [Bi0]:  B(i,1,1)A(i,0,S1)C(i,0,S3) => B(i,1,0)A(i,0,S1)C(i,0,S3) .
rl [Bi0]:  B(i,2,2)A(i,0,S1)C(i,0,S3) => B(i,0,2)A(i,0,S1)C(i,0,S3) .
rl [Bi1]:  B(i,0,0)A(i,0,S1)C(i,1,S3) => B(i,0,1)A(i,0,S1)C(i,1,S3) .
rl [Bi1]:  B(i,0,0)A(i,1,S1)C(i,0,S3) => B(i,0,1)A(i,1,S1)C(i,0,S3) .
rl [Bi2]:  B(i,1,1)A(i,1,S1)C(i,1,S3) => B(i,1,2)A(i,1,S1)C(i,1,S3) .

rl [Ci0]:  C(i,1,1)I(i,1,S5) => C(i,1,0)I(i,1,S5) .
rl [Ci1]:  C(i,0,0)I(i,0,S5) => C(i,0,1)I(i,0,S5) .

rl [Di0]:  D(i,1,1)B(i,2,S2) => D(i,1,0)B(i,2,S2) .
rl [Di0]:  D(i,1,1)I(i,0,S5) => D(i,1,0)I(i,0,S5) .
rl [Di1]:  D(i,0,0)B(i,0,S2)I(i,1,S5) => D(i,0,1)B(i,0,S2)I(i,1,S5) .
rl [Di1]:  D(i,0,0)B(i,1,S2)I(i,1,S5) => D(i,0,1)B(i,1,S2)I(i,1,S5) .
```

We now have the necessary set of rewrite rules we need to start our analysis of our basic model of 5 entities.

5.5.1 Basic Model Analysis

We start our analysis for the synchronous version of our basic model of five entities using the three types of test searches that were introduced in Section 5.3 to see how the model performed:

1) Our first search is searching for a point attractor:

```
search '__['A['0.Nat,'1.D1,'1.D1],'B['0.Nat,'0.D2,'0.D2],
'C['0.Nat,'0.D1,'0.D1], 'D['0.Nat,'0.D1,'0.D1],
'I['0.Nat,'1.D1,'1.D1]] =>! '__[T:Term] .
```

Maude returns no solution here after performing 120 rewrites in 6 milliseconds.

2) We now explore the possibility of all entities going back to 0 starting from an initial state where all entities are at their highest value (where `upTerm` is used to left the initial state to the metalevel representation of the state):

```
search upTerm(init1) =>! '__['A['0.Nat,'0,'0],'B['0.Nat,'0,'0],
'C['0.Nat,'0,'0],'D['0.Nat, '0,'0],'I['0.Nat,'0,'0]] .
```

Maude returns no solution here after exploring 9 states and performing 182 rewrites in 8 milliseconds.

3) For our third and final search here, starting from an initial state of `10101`, we search

for the possibility of B reaching the value 2 (which is its highest), with C simultaneously going back to state 0:

```
search upTerm(init2) =>! '__['B['0.Nat,'2,'2],'C['0.Nat,'0,'0],T:Term] .
```

No solution is found here after Maude have explored 7 states and performed 142 rewrites in 7 milliseconds. Table 5.4 gives a summary of the test results for the three searches performed for the model of size 5.

We now perform our LTL checks:
1) Starting off with the first instance of B at full activation then at some point ending up with the last instance of A always at state 1.

```
red modelCheck( '__['A['0.Zero,'0.D1,'0.D1],'B['0.Zero,'2.D2,'2.D2],
'C['0.Zero,'1.D1,'1.D1], 'D['0.Zero,'1.D1,'1.D1],'I['0.Zero,'1.D1,
'1.D1]] , []  (<> [] atA(0,'1.D1)) ) .
```

Maude performs 181 rewrites in 12 milliseconds returning a counter example showing that the property does not hold.

2) The last instance of C can always eventually be at state 1 starting from an initial state where all instances of A are at state 0.

```
Maude> red modelCheck( '__['A['0.Zero,'0.D1,'0.D1],'B['0.Zero,'2.D2,
'2.D2],'C['0.Zero,'1.D1,'1.D1], 'D['0.Zero,'1.D1,'1.D1],
'I['0.Zero,'1.D1,'1.D1]] , [] <> atC(0,'1.D1) ) .
```

Maude performs 179 rewrites here in 12 milliseconds returning a counter example showing that the property does not hold.

5.5.2 Scaling to 10 Entities

For performance evaluation purposes, we now double the system state space (following the same approach in Section 5.4.2. We now present our test searches:

Table 5.4 Summary of test results for performing three simple searches on synchronous test MVN of Size 5.

Model Size	5
Rewrites	120 182 142
Time (ms)	6 8 7

1) We perform our first search for this size of our scalable model by searching for a point attractor:

```
Maude> search  '__['A['0.Int,'1.D1,'1.D1],'B['0.Int,'0.D2,'0.D2],
'C['0.Int,'0.D1,'0.D1], 'D['0.Int,'0.D1,'0.D1],'I['0.Int,'1.D1,'1.D1],
 'A['s_^1['0.Zero],'1.D1,'1.D1],'B['s_^1['0.Zero],'0.D2,'0.D2],
'C['s_^1['0.Zero],'0.D1,'0.D1],'D['s_^1['0.Zero],'0.D1,'0.D1],
'I['s_^1['0.Zero],'1.D1,'1.D1]] =>! '__[T:Term] .
```

No solution was found here after Maude performed 400 rewrites compared to 120 in smaller model of size 5. Maude also explored 16 states here compared to 6 in the 5 entity model and now has taken 14 milliseconds to perform this search instead of 6.

2) For our second search (exploring the possibility of all entities going back to 0 starting from an initial state where all entities are at their highest value):

```
Maude> search upTerm(init1) =>! '__['A['0.Int, '0,'0],'B['0.Int, '0,'0],
'C['0.Int, '0,'0],'D['0.Int, '0,'0],'I['0.Int, '0,'0],'A['s_^1['0.Zero],
'0,'0],'B['s_^1['0.Zero], '0,'0],'C['s_^1['0.Zero], '0,'0],
'D['s_^1['0.Zero], '0,'0],'I['s_^1['0.Zero], '0,'0]] .
```

Maude performed 477 rewrites here compared to 182 in the 5 entity model. The number of explored states has increased from 9 to 19, and the time taken to perform the search has increased from 8 milliseconds to 19.

3) For our final search here, we perform the same third search we did in the 5 entity model which checks whether both instances of `B` can become active from a given initial state:

```
Maude> search upTerm(init2) =>! '__['B['0.Int, '2,'2],'C['0.Int, '0,'0],
'B['s_^1['0.Zero], '2,'2], 'C['s_^1['0.Zero], '0,'0], T:Term] .
```

The number of rewrites has more than trebled here after Maude performed 452 rewrites in 19 milliseconds. Table 5.5 gives a summary of the test results for the three searches performed for the model of size 10. We now perform our LTL checks:

Table 5.5 Summary of the performance of the three test searches on the synchronous test MVN of Size 10.

Model Size	5	10
Rewrites	120 182 142	400 477 452
Time (ms)	6 8 7	14 19 19

1) Starting off with the first instance of `B` at full activation then at some point ending up with the last instance of `A` always at state 1.

```
red modelCheck( '__['A['0.Zero,'0.D1,'0.D1],'B['0.Zero,'2.D2,'2.D2],
'C['0.Zero,'1.D1,'1.D1],'D['0.Zero,'1.D1,'1.D1],'I['0.Zero,'1.D1,'1.D1],
'A['s_^1['0.Zero],'0.D1,'0.D1],'B['s_^1['0.Zero],'2.D2,'2.D2],
'C['s_^1['0.Zero],'1.D1,'1.D1], 'D['s_^1['0.Zero],'1.D1,'1.D1],
'I['s_^1['0.Zero],'1.D1,'1.D1]] , [] ( <> [] atA(1,'1.D1)) ).
```

Maude performs 455 rewrites in 22 milliseconds returning a counter example showing that the property does not hold.

2) The last instance of `C` can always eventually be at state 1 starting from an initial state where all instances of `A` are at state 0.

```
Maude> red modelCheck( '__['A['0.Zero,'0.D1,'0.D1],'B['0.Zero,'2.D2,
'2.D2],'C['0.Zero,'1.D1,'1.D1],'D['0.Zero,'1.D1,'1.D1],'I['0.Zero,
'1.D1,'1.D1],'A['s_^1['0.Zero],'0.D1,'0.D1],'B['s_^1['0.Zero],'2.D2,
'2.D2],'C['s_^1['0.Zero],'1.D1,'1.D1],'D['s_^1['0.Zero],'1.D1,
'1.D1],'I['s_^1['0.Zero],'1.D1,'1.D1]] , [] <> atC(1,'1.D1) ).
```

Maude performs 453 rewrites here in 26 milliseconds returning a counter example showing that the property does not hold.

5.5.3 Further Scaling Analysis

We now extend our analysis to models of sizes 15,20,25 and 30 to give an idea on the performance of our RL framework with regards to this scalable test model. Below are the three test searches:

1) We search for a point attractor for our first test in the model of size 15:

```
search  '__['A['0.Int,'1.D1,'1.D1],'B['0.Int,'0.D2,'0.D2],'C['0.Int,
'0.D1,'0.D1], 'D['0.Int,'0.D1,'0.D1],'I['0.Int,'1.D1,'1.D1],
'A['s_^1['0.Zero],'1.D1,'1.D1],'B['s_^1['0.Zero],'0.D2,'0.D2],
'C['s_^1['0.Zero],'0.D1,'0.D1],'D['s_^1['0.Zero],'0.D1,'0.D1],
'I['s_^1['0.Zero],'1.D1,'1.D1],'A['s_^2['0.Zero],'1.D1,'1.D1],
'B['s_^2['0.Zero],'0.D2,'0.D2],'C['s_^2['0.Zero],'0.D1,'0.D1],
'D['s_^2['0.Zero],'0.D1,'0.D1], 'I['s_^2['0.Zero],'1.D1,'1.D1]]
=>! '__[T:Term] .
```

this command now results in 510 rewrites being performed by Maude compared to 400 in the model of size 10 taking 19 milliseconds to finish (compared to 14 in the model of size 10). For the model of size 20 the number of rewrite rules jumps to 805, then to 1080 in the 25-entity model and 1395 in the model of size 30.

2) For our second test, we introduce the following search for the 15-entities model:

```
search upTerm(init1) =>! '__['A['0.Int, '0,'0],'B['0.Int, '0,'0],
'C['0.Int,'0,'0],'D['0.Int, '0,'0],'I['0.Int, '0,'0],'A['s_^1['0.Zero],
'0,'0],'B['s_^1['0.Zero], '0,'0],'C['s_^1['0.Zero], '0,'0],
'D['s_^1['0.Zero], '0,'0],'I['s_^1['0.Zero], '0,'0],'A['s_^2['0.Zero],
'0,'0],'B['s_^2['0.Zero], '0,'0],'C['s_^2['0.Zero], '0,'0],
'D['s_^2['0.Zero], '0,'0],'I['s_^2['0.Zero], '0,'0]].
```

the number of rewrites performed by Maude for this command has gone up to 752 rewrites that were performed in 28 milliseconds. That number jumps to 807 in 28 milliseconds in the model of size 20; then to 1082 in 37 for the 25-entity model and finally 1397 in 48 milliseconds for the model of size 30.

3) For our third search that checks if all instances of entity `B` can become active we have the following command for the 15-entity model:

```
search upTerm(init2) =>! '__['B['0.Int, '2,'2],'C['0.Int, '0,'0],
'B['s_^1['0.Zero], '2,'2], 'C['s_^1['0.Zero], '0,'0],
'B['s_^2['0.Zero], '2,'2],'C['s_^2['0.Zero], '0,'0], T:Term].
```

the number of rewrites has gone up from 452 to 602 for this command taking 3 more milliseconds compared to the 10 entity model. It increases to 772 in the model of size 20 then to 962 in the 25-entity model then finally 1307 in the model of size 30, with times being 27,35 and 45 milliseconds respectively.

We now perform our LTL checks:

1) Starting off with the first instance of `B` at full activation then at some point ending up with the last instance of `A` always at state 1.

```
red modelCheck( '__['A['0.Zero,'0.D1,'0.D1],'B['0.Zero,'2.D2,'2.D2],
'C['0.Zero,'1.D1,'1.D1],'D['0.Zero,'1.D1,'1.D1],'I['0.Zero,'1.D1,'1.D1],
'A['s_^1['0.Zero],'0.D1,'0.D1],'B['s_^1['0.Zero],'2.D2,'2.D2],
'C['s_^1['0.Zero],'1.D1,'1.D1], 'D['s_^1['0.Zero],'1.D1,'1.D1],
'I['s_^1['0.Zero],'1.D1,'1.D1],'A['s_^2['0.Zero],'0.D1,'0.D1],
'B['s_^2['0.Zero],'2.D2,'2.D2],'C['s_^2['0.Zero],'1.D1,'1.D1],
'D['s_^2['0.Zero],'1.D1,'1.D1],'I['s_^2['0.Zero],'1.D1,'1.D1]] ,
[] ( <> [] atA(2,'1.D1)) ).
```

Maude performs 687 rewrites in 34 milliseconds returning a counter example showing that the property does not hold.

2) The last instance of `C` can always eventually be at state 1 starting from an initial state where all instances of `A` are at state 0.

```
Maude> red modelCheck( '__['A['0.Zero,'0.D1,'0.D1],'B['0.Zero,'2.D2,
'2.D2],'C['0.Zero,'1.D1,'1.D1],'D['0.Zero,'1.D1,'1.D1],'I['0.Zero,
'1.D1,'1.D1],'A['s_^1['0.Zero],'0.D1,'0.D1],'B['s_^1['0.Zero],'2.D2,
'2.D2],'C['s_^1['0.Zero],'1.D1,'1.D1],'D['s_^1['0.Zero],'1.D1,
'1.D1],'I['s_^1['0.Zero],'1.D1,'1.D1],A['s_^2['0.Zero],'0.D1,'0.D1],
'B['s_^2['0.Zero],'2.D2,'2.D2],'C['s_^2['0.Zero],'1.D1,'1.D1],
 'D['s_^2['0.Zero],'1.D1,'1.D1],'I['s_^2['0.Zero],'1.D1,'1.D1]] ,
 [] <> atC(2,'1.D1) ).
```

Maude performs 675 rewrites here in 37 milliseconds returning a counter example showing that the property does not hold.

A full summary of the test results produced using our 3 test searches for the synchronous version of our model is shown in Table 5.6 below for model sizes 15,20,25 and 30.

Table 5.6 Summary of test results for performing three simple searches on the series of synchronous test MVNs.

Model Size	5	10	15	20	25	30
Rewrites	120	400	510	805	1080	1395
	182	477	752	807	1082	1397
	142	452	602	772	962	1307
Time (ms)	6	14	19	29	37	48
	8	19	28	28	37	48
	7	19	22	27	35	47

5.5.4 Testing Conclusions

We started our analysis for our scalable model with the basic model of 5 entities. Then we extended it repeatedly by adding a new block of 5 entities using the same approach we used to extend the asynchronous version of our test model and performed some tests using the search command to come up with some performance results for our RL framework before we gave a summary of those results in Table 5.6. Figure 5.6 gives a visual representation of the number of rewrites with relation to different model sizes for test searches 1,2 and 3, while Figure 5.7 gives that in terms of search time in milliseconds.

In the synchronous case test results could be obtained for a much larger set of models (up to size 30 and beyond) and the resources needed to carry out the tests scaled well with respect to the MVN size. This is perhaps not surprising; while the synchronous RL framework is more complex due to the need to use meta–level strategies the actual dynamics of synchronous MVNs is far less complex (recall traces are deterministic) than their asynchronous counterparts.

5.6 Conclusions

The RL framework developed for asynchronous and synchronous MVNs in Chapters 3 and 4 needed further testing with regards to its performance. In order to address that we developed a scalable test model in this chapter to test its scalability as the number of entities and the

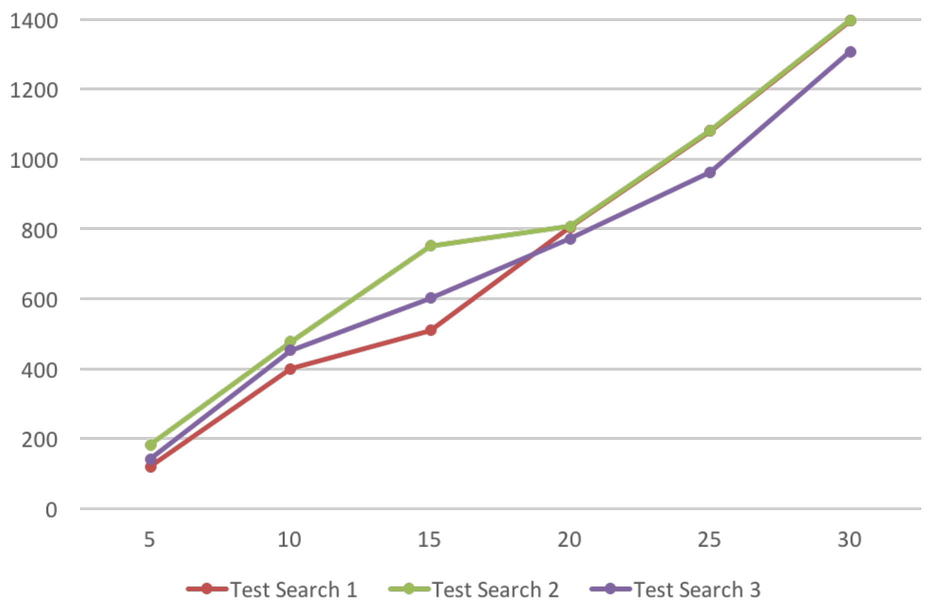


Fig. 5.6 Number of Rewrites in Relation to Model Size for Test Searches 1,2 and 3.

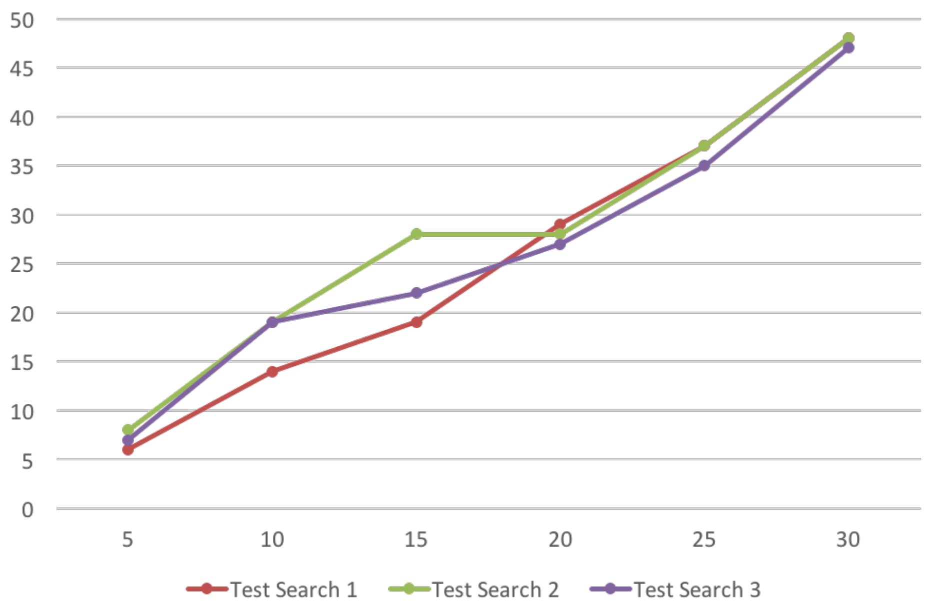


Fig. 5.7 Search Time in Relation to Model Size for Test Searches 1,2 and 3.

states in the model increases. In particular, using the metalevel with synchronous update semantics with its costs and implications.

We started off by introducing a test model of 5 entities in 5.2 as the main building block of our scalable model. We then scaled the model in increments of five by adding a new block of entities and connecting one of its entities to the connecting entity in the first block which we determined to be `I`. We focused on two factors in our scalable model test which were the number of performed rewrites and the time it took to perform those rewrites. We started off by doubling the number of entities in our model by adding a new block then we kept on adding a new block of 5 entities until Maude was not able to produce search results.

We first presented the asynchronous version of the model in Section 5.4, and we performed some analysis using the Maude's search command by performing three different types of test searches. We then moved into the next level of our analysis which was to use some LTL model checking formulas starting with the main building block of 5 entities. We then extended our model analysis by adding a new block of entities at a time and performing the same set of test searches and LTL model checking formulas. After that, we introduced the synchronous version of the test model in Section 5.5 alongside an analysis of the synchronous behaviour. We followed a similar structure to our scaling and testing to the one for the asynchronous model in Section 5.4, and discussed using metalevel with synchronous update semantics with its costs and implications.

In order to illustrate the practical applications of tools and techniques developed in Chapters 3 and 4, we carry out a larger case study in Chapter 6. We introduce models of sizes 13 and 22 using a larger scalable test model based on the gene regulatory network of the segment polarity gene family which is at the basis of *Drosophila* embryonic development. We perform soma analysis using the Maude search command as well as LTL model checking formulas in a similar manner to the the simpler scalable model presented in this chapter.

Finally, we note that while the performance evaluation of our RL framework presented in this chapter does provide some insight into the scalability of the developed RL framework, it is limited in its scope. Much further work is needed here and in particular, we aim to develop a set of benchmark scalable test models which can be used to evaluate MVN approaches in the literature.

Chapter 6

Case Study

6.1 Introduction

Following on the scalable case study in Chapter 5, in this chapter we consider the gene regulatory network of the segment polarity gene family [95, 94] as a larger case study from the literature. This model was selected as a larger case study in order to evaluate the developed techniques and tool support as it is a larger and more complex model compared to our scalable test model. This regulatory network is at the basis of *Drosophila* embryonic development[92, 95, 93, 94]. The network has been investigated through mathematical modelling to determine the network's capacity for generating and maintaining a specific gene expression pattern. During the initial stages of development of the fruit fly [96], three families of genes are successfully activated [95]: the gap genes; the pair-rule genes; and the segment polarity genes. A first mathematical model was proposed by [85], the model was proposed in an attempt to understand and study this network and its properties. Some improvements to the model were introduced in [86], including an alternative mathematical description and analysis of its properties have been presented since [93]. Those studies reached a common conclusion that the interconnections among genes and proteins that make the segment polarity network are the crucial factor for the robustness of the expression pattern with respect to small biological perturbations.

We start by introducing the boolean version of the model where we discuss the two different entities within the model. The model has five different polarity genes and their corresponding proteins. We discuss the behaviour of model entities and the relationship between them by giving a detailed introduction to the interactions happening within a cell of entities. We then introduce a proposed simplified version of the model that reduces the number of entities from 13 to 9 entities and explain the decision behind the simplification.

We then introduce a version of the model where we have a single cell with fixed input values for connecting entities from neighbouring cells. We alternate the inputs for those input entities and discuss the effect of doing so by giving a brief analysis of the model using Maude. We then extend our model to two cells with connecting entities to neighbouring cells increasing the number of entities to 22 where we have two cells of size 9 and four connecting entities to neighbouring cells. We then introduce the set of rewrite rules generated for our model using tool support for both asynchronous and synchronous versions of the model and perform a range of analysis for both models of one and two cells using Maude by performing some searches and LTL model checking formulas.

This Chapter is organised as follows, in Section 6.2 we introduce the original boolean model of 13 entities. In Section 6.2 we discuss the original Boolean model and our simplified version of the model with 9 Entities for a single cell before with 4 connecting entities to neighbouring cells, we then extend our model by adding a new cell which takes the size of the model to 22 entities. We discuss the asynchronous semantics of the model in Section 6.3 starting with the model of one cell in Section 6.3.1, we perform a set of test searches before we extend our model in Section 6.3.2 by adding a new cell and carrying out a set of test searches. We then introduce the synchronous version of the model in Section 6.4 starting with the basic model of one cell in Section 6.4.1 before extending the model by adding a second cell in Section 6.4.2. Finally we give a brief summary to this chapter in Section 6.5.

6.2 The Model

A boolean version of the continuous model described in [85] was proposed in [86]. There are 5 different main polarity genes in the boolean version shown in Figure 6.1, *wingless(wg), engrailed(en), hedgehog(hh), patched(ptc) and cubitus interruptus(ci)* [87, 88, 95, 97]. These genes code for their corresponding proteins (respectively represented by symbols WG, EN, HH, PTC and *CI*). Protein Cubitus interruptus can be converted to a transcriptional activator (*CIA*), or may be cleaved to form a transcriptional repressor (*CIR*) [97]. Proteins *EN*, *CIA* and *CIR* are transcription factors. While *WG* and *HH* are secreted proteins[98, 99]. *PTC* is a transmembrane receptor protein [100].

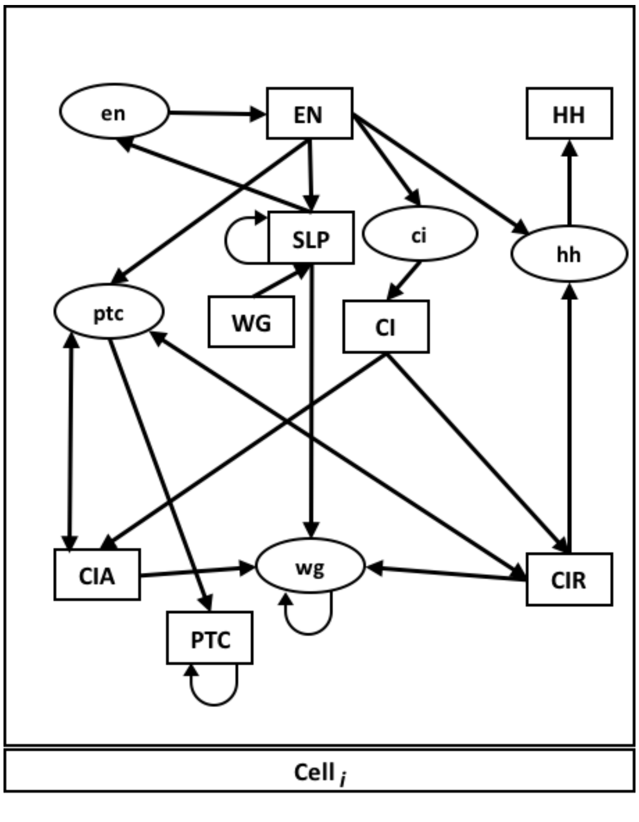


Fig. 6.1 The Wiring Diagram for Cell i

The pair-rule gene *SLP* activates *wg* transcription and represses *en* transcription, where *wg* is secreted from the cells synthesizing it [87, 88] and initiates a signalling cascade leading to the transcription of *en*. Gene product *EN* promotes the transcription of the gene *hh* and represses the transcription of *ci* [89], possibly *ptc* [90]. Gene *HH* is also secreted, and binds to the *HH* receptor *PTC* on a neighbouring cell [91]. We propose an Initial simplified model of 9 entities that simplifies the behaviour of four original model entities (minimizing the state space for the model from 2^13 to 2^9). After that, a two-cell model with 22 entities (2 cells of size 9 and 4 connecting entities to cells *i-1* and *i+2*) is presented and analysed.

We start by introducing the simplified boolean model that reduces the network state by simplifying entities WG_i, EN_i, HH_i and CI_i, reducing the number of network entities from 13 down to 9 (thus reducing the network's state space size from 2^{13} to 2^9). These entities hold the values of wg_i, en_i, hh_i and ci_i from the previous iteration. The list of boolean updating functions is given below:

Entity	Boolean Updating Function
SLP_i	$SLP_i(k+1) = (WG_i(k)$ and not $EN_i(k))$ or $SLP_i(k)$
wg_i	$wg_i(k+1) = (CIA_i(k))$ and $SLP_i(k)$ and not $CIR_i(k))$ or
	$[wg_i(k)$ and $(CIA_i(k))\ SLP_i(k))$ and not $CIR_i(k))]$
en_i	$en_i(k+1) =$ not $SLP_i(k)$
hh_i	$hh_i(k+1) = EN_i(k)$ and not $CIR_i(k)$
ptc_i	$ptc_i(k+1) = CIA_i(k)$ and not $EN_i(k))$ and not $CIR_i(k)$
PTC_i	$PTC_i(k+1) = ptc_i(k)$ or $PTC_i(k)$
ci_i	$ci_i(k+1) =$ not $EN_i(k)$
CIA_i	$CIA_i(k+1) = CI_i(k)$ and not $PTC_i(k)$
CIR_i	$CIR_i(k+1) = CI_i(k)$ and $PTC_i(k)$

Figure 6.2 shows the structure of entities contained in a single cell for the simplified version of the model. The model has two single attractors in states 001111000 and 100001101. We simulate the model using tool support and produce different types of state graphs, those can help in finding different properties within the network structure and can give some insight into the behaviour of the network. Those insights can help determine some model properties that can be checked using the Maude's search command and LTL model checker.

Cell *i* can be more descriptive of the network behaviour when it's fully represented with inter-cell connection entities from cells *i-1* and *i+1* where The values of entities wg_{i-1}, hh_{i-1}, wg_{i+1} and hh_{i+1} are set to different combinations of fixed values (For example: $wg_{i-1} = 0$,

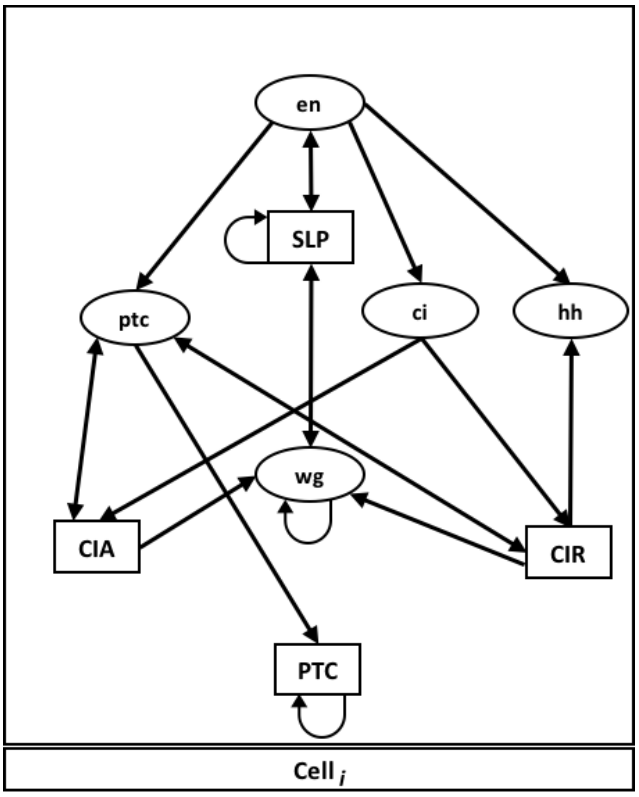


Fig. 6.2 The Simplified Wiring Diagram for Cell *i*.

$hh_{i-1} = 1$, $wg_{i+1} = 1$ and $hh_{i+1} = 1$) increasing the state space of the cell division model by 2^4 states and allowing us to use those 4 values as a testing environment. Adding connecting entities from cells *i-1* and *i+1* ($wg_{i-1}, wg_{i+1}, hh_{i-1}$ *and* hh_{i+1}) and setting them to fixed values will change the look of the network attractors to: 000001101(0000) and 100001101(0000), Where the values in brackets range from 0000 to 1111 for connecting entities resulting in 32 attractors (original 2 * 2^4).

Based on the simplified boolean model, a list of simplified entities equations can be generated using tool support:

$$SLP_i\{0\} = SLP\{0\}wg\{0\} + SLP\{0\}wg\{1\} + SLP\{1\}wg\{1\}$$
$$SLP_i\{1\} = SLP\{1\}wg\{0\}$$

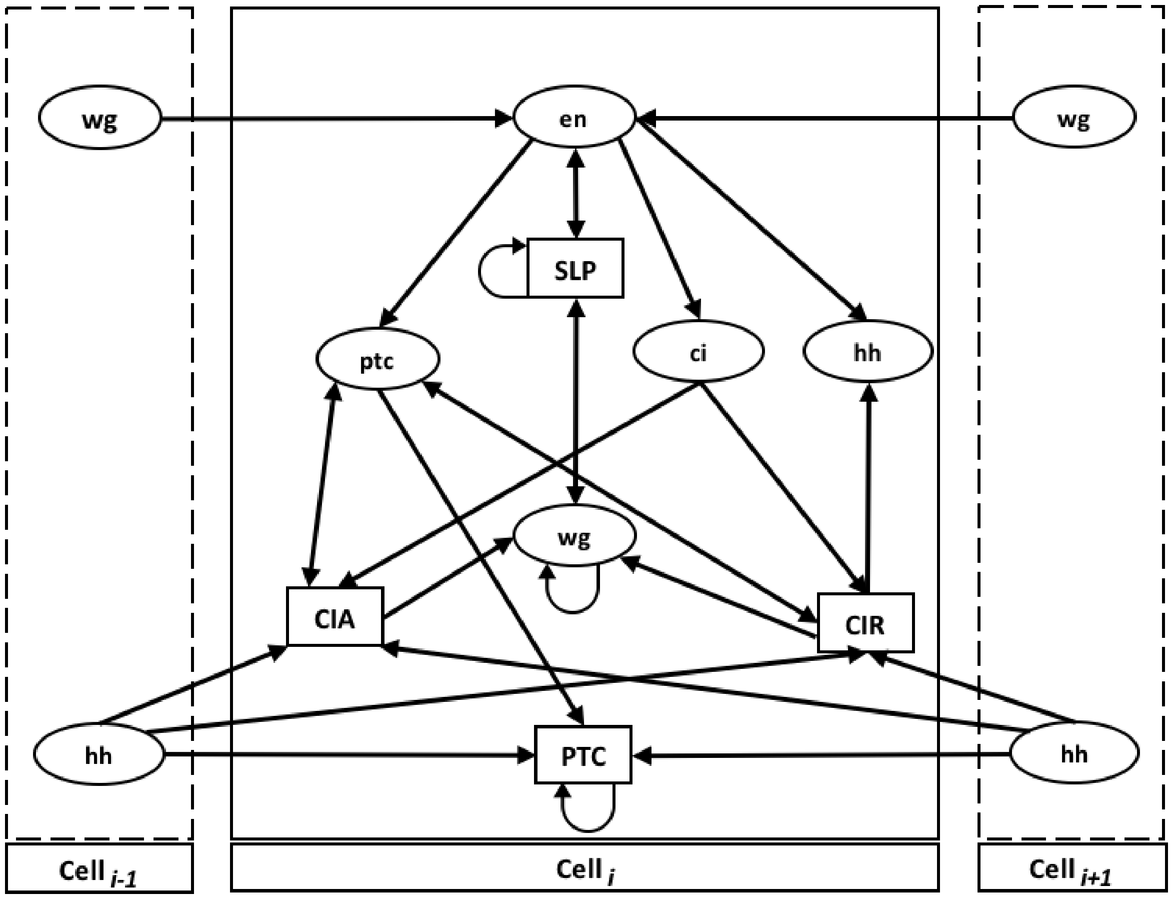


Fig. 6.3 The Simplified Model Showing Connections to Cells *i-1* and *i+1*.

$$wg_i\{0\} = SLP\{0\}wg\{1\}CIA\{0\}CIR\{0\} + SLP\{0\}wg\{1\}CIA\{1\}$$
$$+SLP\{1\}wg\{1\}CIA\{1\}$$
$$wg_i\{1\} = SLP\{1\}wg\{0\}ci\{0\}CIR\{1\}$$

$$en_i\{0\} = SLP\{1\}$$
$$en_i\{1\} = SLP\{0\}$$

$$hh_i\{0\} = en\{0\} + en\{1\}CIR\{1\}$$
$$hh_i\{1\} = en\{1\}CIR\{0\}$$

$$ptc_i\{0\} = en\{0\}CIA\{0\} + en\{0\}CIA\{1\}CIR\{1\}$$
$$ptc_i\{1\} = en\{0\}CIA\{1\}CIR\{0\} + en\{1\}CIA\{0\} + en\{1\}CIA\{1\}$$

$PTC_i\{0\} = PTC\{1\} + ptc\{0\}PTC\{0\}$

$PTC_i\{1\} = ptc\{1\}PTC\{0\}$

$ci_i\{1\} = en\{0\}$

$ci_i\{0\} = en\{1\}$

$CIA_i\{0\} = PTC\{0\}ci\{0\} + PTC\{1\}$

$CIA_i\{1\} = PTC\{0\}ci\{1\}$

$CIR_i\{0\} = PTC\{0\} + PTC\{1\}ci\{0\}$

$CIR_i\{1\} = PTC\{1\}ci\{1\}$

We now present the set of equations for the simplified model with the four connecting entities:

Entity	Boolean Updating Function
SLP_i	$SLP_i(k+1) = (WG_i(k)$ and not $EN_i(k))$ or $SLP_i(k)$
wg_i	$wg_i(k+1) = (CIA_i(k))$ and $SLP_i(k)$ and not $CIR_i(k))$ or
	$[wg_i(k)$ and $(CIA_i(k))\ SLP_i(k))$ and not $CIR_i(k))]$
en_i	$en_i(k+1) = wg_{i-1}(k)$ or $wg_{i+1}(k)$ and not $SLP_i(k)$
hh_i	$hh_i(k+1) = EN_i(k)$ and not $CIR_i(k)$
ptc_i	$ptc_i(k+1) = CIA_i(k)$ and not $EN_i(k))$ and not $CIR_i(k)$
PTC_i	$PTC_i(k+1) = ptc_i(k)$ or $(PTC_i(k)$ and not $hh_{i-1}(k)$ and not $hh_{i+1}(k))$
ci_i	$ci_i(k+1) =$ not $EN_i(k)$
CIA_i	$CIA_i(k+1) = CI_i(k)$ and [not $PTC_i(k)$ or $hh_{i-1}(k)$ or $hh_{i+1}(k)$]
CIR_i	$CIR_i(k+1) = CI_i(k)$ and $PTC_i(k)$ and not $hh_{i-1}(k)$ and not $hh_{i+1}(k)$

Once a model has been developed for a biological system then the next stage is to analyse its behaviour. The idea is to validate the model by checking that it has known biological properties and to produce important new insights that can then be experimentally investigated by biologists. We illustrate the wide range of analysis possible using our RL framework and the support tool Maude by providing a selection of analysis examples for the simplified cell division network of 9 entities with 4 input entities from neighbouring cells representing the setting in which our testing will be carried out.

After we introduce and analyse our model of size 13 using the search command and LTL operators. We extend our model by adding a new cell of 9 entities to study the behavioural changes to the model as its state space grows. Figure 6.4 shows the structure of the model after adding a new cell.

We can now use the list of equations for cells in the model to create an RL model representing the behaviour of the model and generate a set of rewrite rules using tool support.

6.3 Asynchronous Semantics

6.3.1 Basic model of a Single Cell

A list of asynchronous rewrite rules can be generated using tool support and the developed RL framework. We now present the set of rewrite rules for our model:

```
rl [SLP1] : SLP(i,0)wg(i,1)en(i,0) => SLP(i,1)wg(i,1)en(i,0) .
rl [wg0] : wg(i,1)CIA(i,0)SLP(i,0) => wg(i,0)CIA(i,0)SLP(i,0).
rl [wg0] : wg(i,1)CIA(i,1)CIR(i,1) => wg(i,0)CIA(i,1)CIR(i,1) .
rl [wg0] : wg(i,1)CIA(i,0)SLP(i,1)CIR(i,0) =>
wg(i,0)CIA(i,0)SLP(i,1)CIR(i,0) .
rl [wg0] : wg(i,1)CIA(i,1)SLP(i,0)CIR(i,0) =>
wg(i,0)CIA(i,1)SLP(i,0)CIR(i,0) .
rl [wg1] : wg(i,0)CIA(i,1)SLP(i,1)CIR(i,0) =>
wg(i,1)CIA(i,1)SLP(i,1)CIR(i,0) .
rl [wg1] : wg(i,0)CIA(i,0)SLP(i,1)CIR(i,0)wg(i,1) =>
wg(i,1)CIA(i,0)SLP(i,1)CIR(i,0)wg(i,1) .
rl [wg1] : wg(i,0)CIA(i,1)SLP(i,0)CIR(i,0)wg(i,1) =>
```

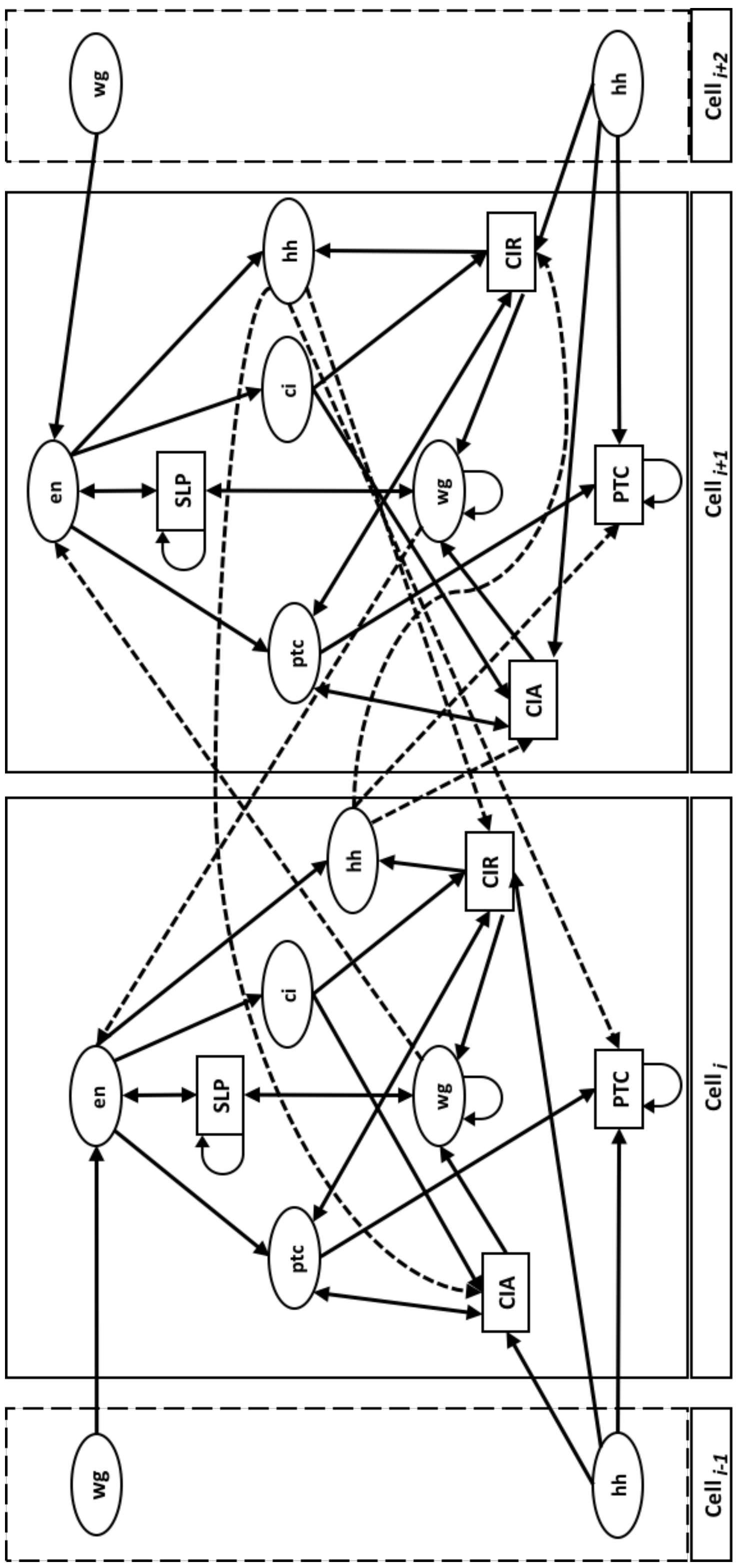


Fig. 6.4 a 2-Cell Model Showing Connections to Cells *i-1* and *i+2*.

```
wg(i,1)CIA(i,1)SLP(i,0)CIR(i,0)wg(i,1) .
rl [en0] : en(i,1)wg(i - s(0),0)wg(i + s(0),0) =>
en(i,1)wg(i - s(0),0)wg(i + s(0),0).
rl [en0] : en(i,1)wg(i - s(0),1)wg(i + s(0),1) =>
en(i,1)wg(i - s(0),1)wg(i + s(0),1).
rl [en0] : en(i,1)wg(i - s(0),0)wg(i + s(0),1)SLP(i,1) =>
en(i,0)wg(i - s(0),0)wg(i + s(0),1)SLP(i,1).
rl [en0] : en(i,1)wg(i - s(0),1)wg(i + s(0),0)SLP(i,1) =>
en(i,1)wg(i - s(0),1)wg(i + s(0),0)SLP(i,1).
rl [en1] : en(i,0)wg(i - s(0),0)wg(i + s(0),1)SLP(i,0) =>
en(i,1)wg(i - s(0),0)wg(i + s(0),1)SLP(i,0) .
rl [en1] : en(i,0)wg(i - s(0),1)wg(i + s(0),0)SLP(i,0) =>
en(i,1)wg(i - s(0),1)wg(i + s(0),0)SLP(i,0) .
rl [hh0] : hh(i,1)en(i,0) => hh(i,0)en(i,0) .
rl [hh0] : hh(i,1)en(i,1)CIR(i,1) => hh(i,0)en(i,1)CIR(i,1) .
rl [hh1] : hh(i,0)en(i,1)CIR(i,0) => hh(i,1)en(i,1)CIR(i,0) .
rl [ptc0] : ptc(i,1)CIA(i,0) => ptc(i,0)CIA(i,0) .
rl [ptc0] : ptc(i,1)CIA(i,1)en(i,0)CIR(i,1) =>
ptc(i,0)CIA(i,1)en(i,0)CIR(i,1) .
rl [ptc0] : ptc(i,1)CIA(i,1)en(i,1) =>
ptc(i,0)CIA(i,1)en(i,1).
rl [ptc1] : ptc(i,0)CIA(i,1)en(i,0)CIR(i,0) =>
ptc(i,1)CIA(i,1)en(i,0)CIR(i,0) .
rl [PTC0] : PTC(i,1)ptc(i,0)hh(i + s(0),1) =>
PTC(i,0)ptc(i,0)hh(i + s(0),1) .
rl [PTC0] : PTC(i,1)ptc(i,0)hh(i - s(0),1)hh(i + s(0),0) =>
PTC(i,0)ptc(i,0)hh(i - s(0),1)hh(i + s(0),0) .
rl [PTC1] : PTC(i,0)ptc(i,1) => PTC(i,1)ptc(i,1) .
rl [ci0] : ci(i,1)en(i,1) => ci(i,0)en(i,1) .
rl [ci1] : ci(i,0)en(i,0) => ci(i,1)en(i,0) .
rl [CIA0] : CIA(i,1)ci(i,0) => CIA(i,0)ci(i,0) .
rl [CIA0] : CIA(i,1)ci(i,1)PTC(i,1)hh(i - s(0),0)hh(i +
s(0),0) => CIA(i,0)ci(i,1)PTC(i,1)hh(i - s(0),0)hh(i + s(0),0).
rl [CIA1] : CIA(i,0)ci(i,1)PTC(i,0) => CIA(i,1)ci(i,1)PTC(i,0) .
rl [CIA1] : CIA(i,0)ci(i,1)PTC(i,1)hh(i - s(0),0)hh(i + s(0),1)
=> CIA(i,1)ci(i,1)PTC(i,1)hh(i - s(0),0)hh(i + s(0),1).
rl [CIA1] : CIA(i,0)ci(i,1)PTC(i,1)hh(i - s(0),1) =>
CIA(i,1)ci(i,1)PTC(i,1)hh(i - s(0),1) .
rl [CIR0] : CIR(i,1)ci(i,0)PTC(i,1)hh(i - s(0),0) hh(i + s(0)
,0) => CIR(i,0)ci(i,0)PTC(i,1)hh(i - s(0),0) hh(i + s(0),0) .
rl [CIR1] : CIR(i,0)ci(i,1)PTC(i,1)hh(i - s(0),0)hh(i + s(0)
,0) => CIR(i,1)ci(i,1)PTC(i,1)hh(i - s(0),0)hh(i + s(0),0).
```

We can test certain properties using this set of rewrite rules for the simplified model. For example: searching for entity `CIA` switching off from a given initial state:

```
search SLP(1,0)wg(0,1)wg(1,1)wg(2,1)en(1,0)hh(0,0)hh(1,1)
hh(2,0)ptc(1,1)PTC(1,0)ci(1,0)CIA(1,1)CIR(1,1) =>!
SLP(1,tSLP:D1)wg(0,twgp:D1)wg(1,twg:D1)wg(2,twgn:D1)
en(1,ten:D1)hh(0,thhp:D1)hh(1,thh:D1)hh(2,thhn:D1)ptc(1,
tptc:D1)PTC(1,tPTC:D1)ci(1,tci:D1)CIA(1,0)CIR(1,tCIR:D1) .
```

Maude returns three solutions in states 125,126 and 127 indicating that `CIA` does switch off in three states in the trace from the chosen initial state:

```
Solution 1 (state 125)
states: 128  rewrites: 352 in 10ms cpu (10ms real)
(32854 rewrites/second)
twgp:D1 --> (1).D1
twg:D1 --> (1).D1
twgn:D1 --> (1).D1
```

```
ten:D1 --> (0).D1
thhp:D1 --> (0).D1
thh:D1 --> (0).D1
thhn:D1 --> (0).D1
tptc:D1 --> (0).D1
tPTC:D1 --> (1).D1
tci:D1 --> (1).D1
tCIR:D1 --> (1).D1
tSLP:D1 --> (1).D1

Solution 2 (state 126)
states: 128  rewrites: 352 in 10ms cpu (10ms real)
(32263 rewrites/second)
twgp:D1 --> (1).D1
twg:D1 --> (0).D1
twgn:D1 --> (1).D1
ten:D1 --> (0).D1
thhp:D1 --> (0).D1
thh:D1 --> (0).D1
thhn:D1 --> (0).D1
tptc:D1 --> (0).D1
tPTC:D1 --> (1).D1
tci:D1 --> (1).D1
tCIR:D1 --> (1).D1
tSLP:D1 --> (0).D1

Solution 3 (state 127)
states: 128  rewrites: 352 in 11ms cpu (11ms real)
(31734 rewrites/second)
```

```
twgp:D1 --> (1).D1
twg:D1 --> (0).D1
twgn:D1 --> (1).D1
ten:D1 --> (0).D1
thhp:D1 --> (0).D1
thh:D1 --> (0).D1
thhn:D1 --> (0).D1
tptc:D1 --> (0).D1
tPTC:D1 --> (1).D1
tci:D1 --> (1).D1
tCIR:D1 --> (1).D1
tSLP:D1 --> (1).D1
```

One property we can check is *CIA* turning on from an initial state where all entities are at state `0` using the following search command:

```
Maude> search SLP(1,0)wg(0,0)wg(1,0)wg(2,0)en(1,0)hh(0,0)
hh(1,0)hh(2,0)ptc(1,0)PTC(1,0)ci(1,0)CIA(1,0)CIR(1,0) =>!
SLP(1,tSLP:D1)wg(0,twgp:D1)wg(1,twg:D1)wg(2,twgn:D1)
en(1,ten:D1)hh(0,thhp:D1)hh(1,thh:D1)hh(2,thhn:D1)
ptc(1,tptc:D1)PTC(1,tPTC:D1)ci(1,tci:D1)CIA(1,1)CIR(1,tCIR:D1) .
```

Maude returns one solution where *CIA* turns on from the given initial state which is found at the 4th state of the trace (we now start using tables to present solution states for brevity and show the full solutions as returned by Maude in Appendix A):

Table 6.1 The Global State Representing the Solution found by Maude.

wg_0	wg_1	wg_2	en_1	hh_0	hh_1	hh_2	ptc_1	PTC_1	ci_1	CIR_1	SLP_1
0	0	0	0	0	0	0	1	1	1	0	0

```
Solution 1 (state 4)
```

```
states: 5  rewrites: 4 in 0ms cpu (0ms real)
(8714 rewrites/second)
```

Another property we can check is that `CIR` is typically absent in cells expressing `wg`. `CIR`=0 when `wg` =1. We can check that property using the following search command starting from an initial state where all instances of `wg` are at state 1:

```
Maude> search SLP(1,0)wg(0,1)wg(1,1)wg(2,1)en(1,0)hh(0,0)
hh(1,0)hh(2,0)ptc(1,0)PTC(1,0)ci(1,0)CIA(1,0)CIR(1,0) =>!
SLP(1,tSLP:D1)wg(0,twgp:D1)wg(1,twg:D1)wg(2,twgn:D1)en(1,
ten:D1)hh(0,thhp:D1)hh(1,thh:D1)hh(2,thhn:D1)ptc(1,tptc:D1)
PTC(1,tPTC:D1)ci(1,tci:D1)CIA(1,tCIA:D1)CIR(1,0) .
```

Maude returns two solutions here at states 17 and 18 of the trace where the property holds:

Table 6.2 The Global States Representing the two Solutions found by Maude.

wg_0	wg_1	wg_2	en_1	hh_0	hh_1	hh_2	ptc_1	PTC_1	ci_1	CIA_1	SLP_1
1	1	1	0	0	0	0	1	1	1	1	1
1	0	1	0	0	0	0	1	1	1	1	0

```
Solution 1 (state 17)
states: 20  rewrites: 30 in 1ms cpu (1ms real)
(15923 rewrites/second)


Solution 2 (state 18)
states: 20  rewrites: 30 in 2ms cpu (2ms real)
(13210 rewrites/second)
```

Another property we can check is that either `CIA` is absent or `CIR` is present in cells not expressing `wg`. `CIA`=0 or `CIR`=1 when `wg`=0. We can check this property using the following search command:

```
Maude> search SLP(1,0)wg(0,0)wg(1,0)wg(2,0)en(1,0)hh(0,0)
hh(1,0)hh(2,0)ptc(1,0)PTC(1,0)ci(1,1)CIA(1,0)CIR(1,0) =>!
SLP(1,tSLP:D1)wg(0,twgp:D1)wg(1,twg:D1)wg(2,twgn:D1)en(1,
ten:D1)hh(0,thhp:D1)hh(1,thh:D1)hh(2,thhn:D1)ptc(1,tptc:D1)
PTC(1,tPTC:D1)ci(1,tci:D1)CIA(1,tCIA:D1)CIR(1,1) .
```

Maude returns a solution at the third state where CIA is absent and CIR is present and both properties hold from the given initial state.

Table 6.3 The Global State Representing the single Solution found by Maude.

wg_0	wg_1	wg_2	en_1	hh_0	hh_1	hh_2	ptc_1	PTC_1	ci_1	CIA_1	SLP_1
0	0	0	0	0	0	0	1	1	1	0	0

```
Solution 1 (state 3)
states: 4  rewrites: 3 in 0ms cpu (0ms real)
(6593 rewrites/second)
```

We now make use of Maude model checking capabilities to check for certain properties. We define an atomic proposition `atCIA: Nat D1 -> Prop` , where `atCIA(s)` is true only if `CIA` is in state s.

```
eq CIA(i,s1) s |= atCIA(i,s1) = true .
eq s |= atCIA(i,s1) = false [owise] .
```

We define a similar proposition for entity `wg` , We can then model check a range of interesting dynamic properties expressed using the temporal operators of LTL as illustrated by the example below for the cell division model:

```
red modelCheck(SLP(1,0)wg(0,0)wg(1,0)wg(2,0)en(1,0)hh(0,0)
hh(1,0)hh(2,0)ptc(1,0)PTC(1,0)ci(1,1)CIA(1,0)CIR(1,0), [] <>
(atCIA(1,1) \/ atwg(1,1)).
```

performing this model checking command, Maude returns the following output:

```
rewrites: 27 in 0ms cpu (0ms real)
(188811 rewrites/second)
result Bool: true
```

which tells that the property holds from the given starting state and Maude returns true. Starting from an initial state where `ci`,`CIA` and `CIR` are all active, we check for the possibility of ending up with `en` always at state `1` using the following model checking command:

```
Maude> red modelCheck(SLP(1,0)wg(0,1)wg(1,0)wg(2,0)en(1,0)
hh(0,0)hh(1,0)hh(2,0)ptc(1,0)PTC(1,0)ci(1,1)CIA(1,1)
CIR(1,1), [] (<> [] aten(1,1))) .
```

Maude returns a counter example showing that no state matching the LTL formula condition is reachable from the given initial state and returns a counter example as follows:

```
rewrites: 13 in 0ms cpu (0ms real) (28697 rewrites/second)
result [ModelCheckResult]: counterexample(nil, {wg(0, 1)
wg(1, 0) wg(2, 0) en(1, 0) hh(0, 0) hh(1, 0) hh(2, 0) ptc(1,
0) PTC(1, 0)ci(1, 1) CIA(1, 1)CIR(1, 1) SLP(1, 0),deadlock})
```

The counter example shows the LTL checker reaching a deadlock with `en` at state `0`.

6.3.2 Extending the Model to 2 Cells

Building on the single cell model that was presented in Section 6.3.1, we now add a new cell to the model without making any changes to the rewrite rules or the model definition as we are working with indexes. We now add a new cell with the index 2 that is using the same set of rewrite rules as the cell in the previous model of one cell.

We now present our first search for this model which looks to the entity `CIA` in the second cell turning off, from a starting state that is now represented using 22 entities (two cells of size 9 and four connecting entities to neighbouring cells):

```
Maude> search SLP(1,0)SLP(2,0)wg(0,0)wg(1,0)wg(2,0)wg(3,0)
en(1,0)en(2,1)hh(0,0)hh(1,0)hh(2,0)hh(3,0)ptc(1,0)ptc(2,0)
```

```
PTC(1,0)PTC(2,1)ci(1,1)ci(2,1)CIA(1,0)CIA(2,1)CIR(1,0)
CIR(2,0) =>! SLP(1,tSLP:D1)SLP(2,tSLP2:D1)wg(0,twgp:D1)
wg(1,twg:D1)wg(2,twg2:D1)wg(3,twgn:D1)en(1,ten:D1)
en(2,ten2:D1)hh(0,thhp:D1)hh(1,thh:D1)hh(2,thh2:D1)
hh(3,thhn:D1)ptc(1,tptc:D1)ptc(2,tptc2:D1)PTC(1,tPTC:D1)
PTC(2,tPTC2:D1)ci(1,tci:D1)ci(2,tci2:D1)CIA(1,tCIA1:D1)
CIA(2,0)CIR(1,tCIR:D1)CIR(2,tCIR2:D1) .
```

Maude returns a single solution which was found on the 23rd state of the trace as follows:

```
Solution 1 (state 23)
states: 24  rewrites: 46 in 3ms cpu (4ms real)
(12182 rewrites/second)
twgp:D1 --> (0).D1
twg:D1 --> (0).D1
twg2:D1 --> (0).D1
twgn:D1 --> (0).D1
ten:D1 --> (0).D1
ten2:D1 --> (1).D1
thhp:D1 --> (0).D1
thh:D1 --> (0).D1
thh2:D1 --> (1).D1
thhn:D1 --> (0).D1
tptc:D1 --> (1).D1
tptc2:D1 --> (0).D1
tPTC:D1 --> (1).D1
tPTC2:D1 --> (1).D1
tci:D1 --> (1).D1
tci2:D1 --> (0).D1
tCIA1:D1 --> (1).D1
tCIR:D1 --> (0).D1
```

```
tCIR2:D1 --> (0).D1
tSLP:D1 --> (0).D1
tSLP2:D1 --> (0).D1
```

For brevity and similar to the representation of solutions introduced earlier in Section 6.3.1, we present search solutions found by Maude using tables from now on. Table 6.4 represents the solution found using the first search command for this model.

Table 6.4 The Global State Representing the single Solution found by Maude.

wg_0	SLP_1	wg_1	en_1	hh_1	ptc_1	PTC_1	ci_1	CIA_1	CIR_1	wg_3
0	0	0	0	0	0	1	1	1	0	0
hh_0	SLP_2	wg_2	en_2	hh_2	ptc_2	PTC_2	ci_2	CIA_2	CIR_2	hh_3
0	0	0	1	1	0	1	0	0	0	0

We now take a look at the behavioural changes to entities `hh` and `wg` within cell `i`. We start from an initial state where they are both at state `1` and check if they reach a point where they are both at state `0` using the following search command:

```
search init1 =>! SLP(1,tSLP:D1)SLP(2,tSLP2:D1)wg(0,twgp:D1)
wg(1,0)wg(2,twg2:D1)wg(3,twgn:D1)en(1,ten:D1)en(2,ten2:D1)
hh(0,thhp:D1)hh(1,0)hh(2,thh2:D1)hh(3,thhn:D1)
ptc(1,tptc:D1)ptc(2,tptc2:D1)PTC(1,tPTC:D1)PTC(2,tPTC2:D1)
ci(1,tci:D1)ci(2,tci2:D1)CIA(1,tCIA1:D1)CIA(2,tCIA2:D1)
CIR(1,tCIR:D1)CIR(2,tCIR2:D1) .
```

Maude returns a single solution where `hh(1)` and `wg(1)` are both at state `0` that was reachable after performing 609 rewrites. Table 6.5 represents the solution found using this search command for this model.

```
Solution 1 (state 169)


states: 212  rewrites: 688 in 21ms cpu (23ms real)
(32544 rewrites/second)
```

Table 6.5 The Global State Representing the single Solution found by Maude.

wg_0	SLP_1	wg_1	en_1	hh_1	ptc_1	PTC_1	ci_1	CIA_1	CIR_1	wg_3
1	0	0	0	0	1	1	1	1	0	1
hh_0	SLP_2	wg_2	en_2	hh_2	ptc_2	PTC_2	ci_2	CIA_2	CIR_2	hh_3
1	0	0	1	0	0	1	0	0	1	1

We now perform two model checking tests using LTL formulas using the same initial state and only changing We can check the hypothesis that when `en(1)` becomes permanently inactive then `en(2)` must continually be able to be on. This can be done using the following model checking instruction:

```
Maude> red modelCheck(SLP(1,0)SLP(2,1)wg(0,1)wg(1,1)wg(2,1)
wg(3,1)en(1,0)en(2,0)hh(0,0)hh(1,0)hh(2,0)hh(3,0)ptc(1,0)
ptc(2,0)PTC(1,0)PTC(2,0)ci(1,0)ci(2,1)CIA(1,0)CIA(2,1)
CIR(1,0)CIR(2,0), <> [] aten(1,1) -> [] <>  aten(2,1)) .
```

This holds for the given initial state and Maude returns the following:

```
rewrites: 162 in 7ms cpu (8ms real) (20349 rewrites/second)
result Bool: true
```

Furthermore we can now check how the interaction between the two cells affects the original behaviour of entity `ptc` as an example, we can check whether `en(2)` being inactive can lead to the activation of `ptc(2)` using the following model checking instruction:

```
Maude> red modelCheck(SLP(1,1)SLP(2,1)wg(0,0)wg(1,0)wg(2,0)
wg(3,0)en(1,1)en(2,1)hh(0,1)hh(1,0)hh(2,0)hh(3,1)ptc(1,0)
ptc(2,0)PTC(1,0)PTC(2,0)ci(1,0)ci(2,0)CIA(1,0)CIA(2,1)
CIR(1,0)CIR(2,0), <> [] aten(2,0) -> [] atptc(1,1)) .
```

Maude returns true from the given initial state showing the effect of setting the value `hh(3)` to `1` on the behaviour of the `ptc` instance in cell `1`.

```
rewrites: 81 in 0ms cpu (2ms real) (97826 rewrites/second)
result Bool: true
```

The two model checking instructions show an increase in the number of rewrites performed compared to the single-cell model. Maude is performing less rewrites per second as a result of the new model size which is expected.

6.4 Synchronous Semantics

6.4.1 Basic Model of a single Cell

We now move onto the synchronous semantics of the single cell model. A list of synchronous rewrite rules are generated using the RL framework and tool support. The Introduction of variables s3 and s4 is needed as the next state of entities ptc and hh may or may have not been updated at this stage:

```
rl [PTC0] : PTC(i,1,1)ptc(i,0,S4)hh(i + s(0),1,S3) =>
PTC(i,1,0)ptc(i,0,S4)hh(i + s(0),1,S3) .
rl [PTC0] : PTC(i,1,1)ptc(i,0,S4)hh(i - s(0),1,S3)
hh(i + s(0),0,S3) => PTC(i,1,0)ptc(i,0,S4)hh(i - s(0),1,S3)
hh(i + s(0),0,S3) .
rl [PTC1] : PTC(i,0,0)ptc(i,1,S4) => PTC(i,0,1)ptc(i,1,S4) .
```

The meta-level term representing a global state is as follows:

```
'__['wg['0.Zero,'0.D1,'0.D1],'wg['s_['0.Zero],'0.D1,'0.D1],
'wg['s_^2['0.Zero],'0.D1,'0.D1],'en['s_['0.Zero],'0.D1,'0.D1],
'hh['0.Zero,'0.D1,'0.D1],'hh['s_['0.Zero],'0.D1,'0.D1],
'hh['s_^2['0.Zero],'0.D1,'0.D1],'ptc['s_['0.Zero],'0.D1,'0.D1],
'PTC['s_['0.Zero],'0.D1,'0.D1],'ci['s_['0.Zero],'1.D1,'0.D1],
'CIA['s_['0.Zero],'0.D1,'0.D1],'CIR['s_['0.Zero],
'0.D1,'0.D1],'SLP['s_['0.Zero],'0.D1,'0.D1]]
```

We define this meta-level term as an initial state and name it `init` to be used in the following searches.

We now perform a few tests using the search command starting from the initial state `init`. We search for both `CIA` and `ptc` both turning on at the same time (under settings `hh(0,1) hh(2,0) wg(1,1) wg(2,0)`:

```
search init =>+ '__['CIA['s_['0.Zero],'1.D1,'1.D1],T1:Term,
T2:Term,T3:Term,T4:Term,T5:Term,T6:Term,T7:Term,T8:Term,
'ptc['s_['0.Zero],'1.D1,'1.D1],T10:Term,T11:Term,T12:Term] .
```

Maude returns a single solution at the fourth state of the trace after performing 149 rewrites in 16 milliseconds:

```
Solution 1 (state 4)
states: 5  rewrites: 122 in 14ms cpu (14ms real)
(8691 rewrites/second)
T1 --> 'CIR['s_['0.Zero],'0.D1,'0.D1]
T2 --> 'PTC['s_['0.Zero],'1.D1,'1.D1]
T3 --> 'SLP['s_['0.Zero],'1.D1,'1.D1]
T4 --> 'ci['s_['0.Zero],'1.D1,'1.D1]
T5 --> 'en['s_['0.Zero],'0.D1,'0.D1]
T6 --> 'hh['0.Zero,'1.D1,'1.D1]
T7 --> 'hh['s_['0.Zero],'0.D1,'0.D1]
T8 --> 'hh['s_^2['0.Zero],'0.D1,'0.D1]
T10 --> 'wg['0.Zero,'0.D1,'0.D1]
T11 --> 'wg['s_['0.Zero],'1.D1,'1.D1]
T12 --> 'wg['s_^2['0.Zero],'0.D1,'0.D1]
```

Now that we presented the Meta-level representation of solutions found by Maude, we start using tables to present solutions for brevity (we provide the full solutions as returned by Maude in Appendix A):

Table 6.6 The Global State Representing the Solution found by Maude.

CIR_1	PTC_1	SLP_1	ci_1	en_1	hh_0	hh_1	hh_2	wg_0	wg_1	wg_2
0	1	1	1	0	1	0	0	0	1	0

We now change the initial state of our search to have `PTC` at `1` and we set the four connecting entities to `1` and check if `SLP` can turn on from the given initial state:

```
search init =>+ '__[T0:Term,T1:Term,T2:Term,
'SLP['s_['0.Zero],'1.D1,'1.D1],T4:Term,T5:Term,
T6:Term,T7:Term,T8:Term,T9:Term,T10:Term,T11:Term,T12:Term] .
```

Maude returns a single solution after performing 58 rewrites in total reaching the solution after performing 31 rewrites:

Table 6.7 The Global State Representing the Solution found by Maude.

CIA_1	CIR_1	PTC_1	ci_1	en_1	hh_0	hh_1	hh_2	ptc_1	wg_0	wg_1	wg_2
0	0	1	1	0	1	0	1	0	0	0	1

```
Solution 1 (state 1)
states: 2  rewrites: 31 in 5ms cpu (5ms real)
(5819 rewrites/second)
```

We now reset our initial state to have all internal cell entities at state `0` while the four connecting entities (`hh(i-1)`,`hh(i+1)`,`wg(i-1)` and `wg(i+1)`) are at state `1` and check if we can have four entities (`CIA, PTC, SLP and ci`) all turning on at the same time:

```
search init =>+ '__['CIA['s_['0.Zero],'1.D1,'1.D1],T1:Term,
'PTC['s_['0.Zero],'1.D1,'1.D1],'SLP['s_['0.Zero],'1.D1,
'1.D1],'ci['s_['0.Zero],'1.D1,'1.D1],T5:Term,T6:Term,
T7:Term,T8:Term,T9:Term,T10:Term,T11:Term,T12:Term] .
```

Maude finds a single solution where all four entities are at state 1 at the same time and returns the following output:

Table 6.8 The Global State Representing the Solution found by Maude.

CIR_1	en_1	hh_0	hh_1	hh_2	ptc_1	wg_0	wg_1	wg_2
0	0	0	0	1	1	0	1	1

```
Solution 1 (state 4)
states: 5  rewrites: 117 in 13ms cpu (13ms real)
(8760 rewrites/second)
```

We can check if SLP and CIA can become simultaneously active from an initial state where they are inactive using the following model checking instruction:

```
Maude> red modelCheck( init , <> [] (atSLP(1,'1.D1) /\  atCIA(1,'1.D1))) .
```

Maude returns a counter example showing that no state meeting the LTL formula can be reached from the given initial state:

```
rewrites: 163 in 15ms cpu (17ms real) (10324 rewrites/second)
result ModelCheckResult: counterexample({'__['wg['0.Zero,'0.D1,'0.D1],
'wg['s_['0.Zero],'1.D1,'1.D1],'wg['s_^2['0.Zero],'1.D1,'1.D1],
'en['s_['0.Zero],'0.D1,'0.D1],'hh['0.Zero,'0.D1,'0.D1],'hh['s_['0.Zero],
'1.D1,'1.D1],'hh['s_^2['0.Zero],'1.D1,'1.D1],'ptc['s_['0.Zero],'0.D1,
'0.D1],'PTC['s_['0.Zero],'0.D1,'0.D1],
```

We now look into extending our synchronous model by adding a second cell taking and performing a range of tests using the search command and the LTL model checker.

6.4.2 Extending the Model to 2 Cells

We now add a second cell to the model taking the number of entities within the model to 22. We define the following meta level term as the starting point for our search and give it the name `init`:

```
'__['wg['0.Zero,'0.D1,'0.D1],'wg['s_['0.Zero],'0.D1,'0.D1],
'wg['s_^2['0.Zero],'0.D1,'0.D1],'wg['s_^3['0.Zero],'0.D1,'0.D1],
'en['s_['0.Zero],'0.D1,'0.D1],'en['s_^2['0.Zero],'1.D1,'1.D1],
'hh['0.Zero,'0.D1,'0.D1],'hh['s_['0.Zero],'0.D1,'0.D1],
'hh['s_^2['0.Zero],'0.D1,'0.D1],'hh['s_^3['0.Zero],'0.D1,'0.D1],
'ptc['s_['0.Zero],'0.D1,'0.D1],'ptc['s_^2['0.Zero],
'0.D1,'0.D1],'PTC['s_['0.Zero],'0.D1,'0.D1],
'PTC['s_^2['0.Zero],'1.D1,'1.D1],'ci['s_['0.Zero],'1.D1,'1.D1],
'ci['s_^2['0.Zero],'1.D1,'1.D1],'CIA['s_['0.Zero],'0.D1,'0.D1],
'CIA['s_^2['0.Zero],'1.D1,'1.D1],'CIR['s_['0.Zero],'0.D1,'0.D1],
'CIR['s_^2['0.Zero],'0.D1,'0.D1],'SLP['s_['0.Zero],
'0.D1,'0.D1],'SLP['s_^2['0.Zero],'0.D1,'0.D1]] .
```

Our first search checks if both instances of `CIA` and both instances of `PTC` can be active at the same time:

```
search init =>+ '__['CIA['s_['0.Zero],'1.D1,'1.D1],
'CIA['s_^2['0.Zero],'0.D1,'0.D1],T2:Term,T3:Term,
'PTC['s_['0.Zero],'1.D1,'1.D1],'PTC['s_^2['0.Zero],'1.D1,'1.D1],
T6:Term,T7:Term,T8:Term,T9:Term,T10:Term,T11:Term,T12:Term,
T13:Term,T14:Term,T15:Term,T16:Term,T17:Term,T18:Term,
T19:Term,T20:Term,T21:Term] .
```

Maude returns a single solution at the third state of the trace reaching the solution after performing 115 rewrites:

```
Solution 1 (state 3)
states: 4  rewrites: 115 in 12ms cpu (12ms real)
(9200 rewrites/second)
T2 --> 'CIR['s_['0.Zero],'0.D1,'0.D1]
T3 --> 'CIR['s_^2['0.Zero],'0.D1,'0.D1]
T6 --> 'SLP['s_['0.Zero],'0.D1,'0.D1]
```

```
T7 --> 'SLP['s_^2['0.Zero],'0.D1,'0.D1]
T8 --> 'ci['s_['0.Zero],'1.D1,'1.D1]
T9 --> 'ci['s_^2['0.Zero],'0.D1,'0.D1]
T10 --> 'en['s_['0.Zero],'0.D1,'0.D1]
T11 --> 'en['s_^2['0.Zero],'1.D1,'1.D1]
T12 --> 'hh['0.Zero,'0.D1,'0.D1]
T13 --> 'hh['s_['0.Zero],'0.D1,'0.D1]
T14 --> 'hh['s_^2['0.Zero],'1.D1,'1.D1]
T15 --> 'hh['s_^3['0.Zero],'0.D1,'0.D1]
T16 --> 'ptc['s_['0.Zero],'1.D1,'1.D1]
T17 --> 'ptc['s_^2['0.Zero],'0.D1,'0.D1]
T18 --> 'wg['0.Zero,'0.D1,'0.D1]
T19 --> 'wg['s_['0.Zero],'0.D1,'0.D1]
T20 --> 'wg['s_^2['0.Zero],'0.D1,'0.D1]
T21 --> 'wg['s_^3['0.Zero],'0.D1,'0.D1]
```

We now search with the aim of finding a state where both instances of PTC from cells 1 and 2 are at state 1 using the following search:

```
Maude> search init =>+ '__[T0:Term,T1:Term,T2:Term,T3:Term,
'PTC['s_['0.Zero],'1.D1,'1.D1],'PTC['s_^2['0.Zero],'1.D1,'1.D1],
T6:Term,T7:Term,T8:Term,T9:Term,T10:Term,T11:Term,T12:Term,
T13:Term,T14:Term,T15:Term,T16:Term,T17:Term,T18:Term,
T19:Term,T20:Term,T21:Term] .
```

Maude returns two solutions after performing 120 rewrites and Table 6.9 shows the two solution states.

We now make use of Maude model checking capabilities to check for certain properties. We define an atomic proposition atPTC, where it is true only if PTC is in state 0.

```
op atPTC : Nat Qid -> Prop .
eq  '__[T1:Term,T2:Term,T3:Term,'PTC['s_['0.Zero],'0.D1,'0.D1],
```

Table 6.9 The Global State Representing the 2 Solutions found by Maude.

wg_0	SLP_1	wg_1	en_1	hh_1	ptc_1	PTC_1	ci_1	CIA_1	CIR_1	wg_3
1	0	0	0	0	0	1	1	1	0	0
hh_0	SLP_2	wg_2	en_2	hh_2	ptc_2	PTC_2	ci_2	CIA_2	CIR_2	hh_3
0	0	0	1	1	0	1	0	1	0	0

wg_0	SLP_1	wg_1	en_1	hh_1	ptc_1	PTC_1	ci_1	CIA_1	CIR_1	wg_3
1	0	0	0	0	1	1	1	1	0	0
hh_0	SLP_2	wg_2	en_2	hh_2	ptc_2	PTC_2	ci_2	CIA_2	CIR_2	hh_3
1	0	0	1	1	0	1	0	0	0	0

```
T5:Term,T6:Term,T7:Term,T8:Term,T9:Term,T10:Term,T11:Term,
T12:Term,T13:Term,T14:Term,T15:Term,T16:Term,T17:Term,
T18:Term,T19:Term,T20:Term,T21:Term] |= atPTC(i,'1) = true .
eq  T |= atPTC(i,Q1) = false [owise] .
```

We can then model check a range of interesting dynamic properties expressed using the temporal operators of LTL as illustrated by the example below for the cell division model of two cells:

```
red modelCheck( init , <> [] atPTC(1,'1.D1) ) .
```

Maude returns a counter example in this case for the check starting from the starting state `init`(the full trace of the counter example is omitted for brevity and is shown in full in Appendix A):

```
rewrites: 166 in 21ms cpu (23ms real) (7851 rewrites/second)
result ModelCheckResult: counterexample({'__['wg['0.Zero,'0.D1,
'0.D1],'wg['s_['0.Zero],'0.D1,'0.D1],'wg['s_^2['0.Zero],'0.D1,
'0.D1],'wg['s_^3['0.Zero],'0.D1,'0.D1], .....
```

We can check if `SLP` and `CIA` instances in the second cell can become simultaneously active from an initial state where their instances in the first cell are inactive using the following model checking instruction:

```
Maude> red modelCheck( init , <> [] (atSLP(2,'1.D1) /\  atCIA(2,'1.D1)) ) .
```

Maude returns a counter example showing that the property does not hold and no state meeting the LTL formula is reachable from the initial state `init` (the full trace of the counter example is omitted for brevity and is shown in full in Appendix A):

```
rewrites: 137 in 9ms cpu (13ms real) (14204 rewrites/second)
result ModelCheckResult: counterexample({'__['wg['0.Zero,'1.D1,'1.D1],
'wg['s_['0.Zero],'1.D1,'1.D1],'wg['s_^2['0.Zero],'1.D1,'1.D1],
'wg['s_^3['0.Zero],'0.D1,'0.D1],'en['s_['0.Zero],'0.D1,'0.D1],
'en['s_^2['0.Zero],'1.D1,'1.D1],'hh['0.Zero,'0.D1,'0.D1],
'hh['s_['0.Zero],'0.D1,'0.D1],'hh['s_^2['0.Zero],'0.D1,'0.D1],
'hh['s_^3['0.Zero],'0.D1,'0.D1],'ptc['s_['0.Zero],'1.D1, '1.D1], .....
```

6.5 Conclusions

In this chapter, an analysis of the effect of cell division on expression patterns of the segment polarity genes was produced as a larger case study that builds on the scalable test model introduced in Chapter 5. We started with a simplified version of a single cell model that reduced the model's state space from 2^13 to 2^9 by simplifying the behaviour of 4 entities. After that, a proposed model with of size 13 was introduced by adding the 2 connecting entities from neighbouring cells $i-1$ and $i+1$, those entities were set to fixed values at first to study the effect they had added from the original 9 entities model. Lastly, a model of size 22 was introduced that represented two cells and their 4 connecting entities to cells $i-1$ and $i+2$.

We started by introducing an adapted boolean version of the model where we introduced the dynamics of the system which has five different polarity genes and their corresponding proteins. We discussed the behaviour of model entities and the relationship between them by giving a detailed introduction to the interactions happening within a cell of entities. We then introduced a proposed simplified version of the model that had 9 entities instead of 13 and explained the decision behind the simplification.

We then proposed a version of the model where we have a single cell with four fixed input values for connecting entities from neighbouring cells. We alternate the inputs for those input entities and performed a range of analysis using the search command starting from an initial state with different input values from neighbouring cells. We then extended our analysis by performing some LTL checks using certain properties that were true in the original network and presented a few counter examples where those properties do not hold in our adapted version of the model. We then extended our model to two cells with connecting entities to neighbouring cells increasing the number of entities to 22 and performed a range of checks using the search command where the number of rewrites per second was increasing, and the LTL model checker where it was apparent that Maude was taking longer to come up with counter examples for our LTL checks. After that, we introduced the synchronous versions of of the two model sizes and performed a range of analysis for both models using Maude by performing some searches and LTL model checking formulas.

The case study presented in this chapter have concluded the technical content of this thesis by introducing a larger case studies than the ones considered in Chapters 3,4 and 5 and have proven to be a good test to the developed RL framework and tool support by starting the analysis of the model from traces and attractors to modelling it using RL and Maude and producing a range of interesting biological properties.

Chapter 7

Concluding Remarks

7.1 Summary

At the start of this thesis, we stated that **"This thesis aims to investigate the application of RL to modelling and analysing MVNs, and to develop a set of tools and techniques to support this"**. We have set out to develop and evaluate tools and techniques for MVNs based on using RL and Maude. Rewriting logic has formed the core of this work, owing to its numerous modelling advantages, including: (i) the ability of the framework to model and analyse the behaviour of dynamic, concurrent systems; (ii) the success RL had in being used to model a wide range of different formalisms and systems such as: process algebras, Petri nets and their application in biological systems; (iii) the ability of RL to model the dynamic behaviour of a given system using rewrite rules that can capture the non–deterministic state transitions that occur in such systems; and (iv) rewriting strategies that can control the application of rewrite rules and so allow an RL model to capture subtle aspects of the behaviour of a dynamic system.

We developed an RL model for asynchronous MVNs based on their asynchronous update semantics. A translation approach that translates an MVN into an RL model was presented and discussed. The translation approach dealt with straightforward translations taking a single equation part and producing a matching rewrite rule. More importantly, the translation approach dealt with more complex translations when an entity is missing from an equation and can hold more than one value hence the need for two or more equations. The translation approach also had to make sure to add missing terms in case they are not present. We provided a formal correctness argument where we formally showed that our translation approach for both asynchronous and synchronous MVNs to an RL model was correct. In our argument we proved the soundness and completeness of our approach. Techniques and the developed RL framework were then illustrated using case studies for both the asynchronous and synchronous semantics. The synchronous mechanisms were more challenging as we needed to introduce and make use of rewriting strategies and the metalevel representation of terms within those models. Those case studies have helped motivate our RL framework by illustrating the analysis techniques and flexibility available when using RL and Maude. We formally investigated those case studies using a range of powerful analysis tools provided by Maude, which was one of the motivations for choosing to use RL.

The case studies presented for our RL framework for asynchronous and synchronous MVNs provided a good illustration of the practical application of the RL techniques we have developed. However, as they provided little indication of how the developed RL approach would scale when applied to larger MVN models and what impact the well–known state space explosion problem would have. In order to evaluate the performance and the scalability of our RL framework, we addressed this by defining an artificial, scalable MVN model in order to allow a range of model sizes to be considered and we investigated how our RL framework performed as the MVN size (i.e. number of entities) increased. We started off by introducing a test model of 5 entities as the main building block, then we scaled the model in increments of five and explained how blocks are connected when it comes to scaling our test model. We started off by doubling the number of entities in our model by adding a new block and performed some analysis on what kind of behaviour that adds to our test model. Using the developed techniques and tools we presented both asynchronous and synchronous versions of our test model and performed a systematic analysis. We started our analysis using the Maude's search command where we identified three types of test searches to be carried out. Following that we defined two model checking commands using LTL model checking formulas. We started off by testing the basic version of the model that had 5 entities before scaling the test models in increments of 5 and performing the same test searches and LTL model checking commands. The results of those tests gave as an idea on the scalability of our framework, as those test results were summarised and categorised as the models increased in size.

A larger case study from the literature which was the gene regulatory network of the segment polarity gene family at the basis of *Drosophila* embryonic development was also introduced to be analysed and tested using the developed techniques and the RL framework. After we analysed the case study and ran a range of tests using our RL framework we were able to give some insights into how well it coped with a larger case study. We started with a simplified version of a single cell model that reduced the model's state space from 2^{13} to 2^9 by simplifying the behaviour of 4 entities. After that, a proposed model with of size 13 was introduced by adding the 2 connecting entities from neighbouring cells $i-1$ and $i+1$, those entities were set to fixed values at first to study the effect they had added from the original 9 entities model. Lastly, a model of size 22 was introduced that represented two cells and their 4 connecting entities to cells $i-1$ and $i+2$. The RL framework was able to cope with the different model sizes and we were able to generate RL models that we analysed using Maude's search command and the LTL model checker under different sets of settings using certain properties that were true in the original network and presented a few counter examples where those properties do not hold in our adapted version of the model with no issues regarding performance.

7.2 What has been achieved

Looking back at the list of contributions we identified in Section 1.2, we have focused on developing these aspects of RL with a specific application to MVNs. In particular, this thesis has resulted in the following key contributions:

1. Defining a new semantic translation from an asynchronous MVN into an RL model that appears to be the first such translation in the literature (presented in Chapter 3).

Importantly, we formally proved the correctness of our translation approach in Section 3.3.

2. Defining a new semantic translation from a synchronous MVN into an RL model by the use of rewriting strategies that appears to be the first such translation in the literature (presented in Chapter 4). Importantly, we formally proved the correctness of our translation approach in Section 4.3.

3. Developing a scalable test model to carry out a formal investigation into the scalability of the developed RL framework and it's performance (presented in Chapter 5).

4. Developing an integrated toolset to support the developed RL framework and perform automatic translation from an MVN to an RL model and vice versa (presented in Chapter 2 and used for case studies presented in Chapters 3,4,5 and 6).

5. Undertaking a biologically relevant case study from the literature to investigate applying the developed RL framework and illustrate the practical applications and support the application of new RL techniques and tools using a parametric model. This case study provides a a starting point for anyone who would like to use the developed techniques and the RL framework (presented in Chapter 6).

7.3 Future Work

This thesis has investigated the support tools available for MVNs and RL. With our RL framework and the developed tools and techniques, a number of interesting areas of future research can be done to take this work forward. We represent those as a prioritised list as follows:

1. Evaluate the use of RL in modelling and analysing MVNs by considering larger case studies from the literature to further test the performance of our RL framework with regards to limitations and required computer power.

2. Evaluate the developed tools and techniques and aim to enhance the performance of our RL framework. While a range of case studies were undertaken during the course of this thesis, case studies from different areas could be considered to enable further development of the RL framework.

3. Integrate the developed tools and techniques into a compositional approach (for example, see [102]) allowing a range of larger compositional models to be considered and tested .

4. Extend the developed tool support to hide the use of Maude at the meta-level, and develop meta-level search facilities that are tailored to MVN's. This would simplify the process of testing and analysing synchronous MVNs using the developed RL framework.

5. Investigate Extending MVN modelling approach with new concepts such as prioritising entities and support entity location based update which can lead to a wider range of models from different areas to be analysed and tested.

6. Investigate further practical applications for the developed tools and techniques. Such as: work with biologists to develop a range of synthetic models to support synthetic biology [110] automating as a new area where the applicability of the developed framework can be tested and improved, and to develop a range of adaptation techniques for different sets of requirements.

References

[1] Bartocci, E. and Lió, P. (2016). Computational modelling, formal analysis and tools for systems biology. *PLOS Computational Biology*, 12(1), pp. e1004591.

[2] Rudell, R. and Sangiovanni-Vincentelli, A. (1987). Multiple-Valued Minimization for PLA Optimization. *IEEE Transactions on Computer-Aided Design*, CAD-6.

[3] Thomas, R. and D'Ari, R. (1990). *Biological Feedback*, CRC Press.

[4] Chaouiya C., Remy, E. and Thieffry, D. (2008). Petri Net Modelling of Biological Regulatory Networks. *Journal of Discrete Algorithms*, 6(2):165–177.

[5] Banks, R. and Steggles, L. J. (2007). A High-Level Petri Net Framework for Multi-Valued Genetic Regulatory Networks. *Journal of Integrative Bioinformatics*, 4(3):60.

[6] Kauffman, S. A. (1969). Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22(3):437–467.

[7] Wuensch, A. (2002). Basins of Attraction in Network Dynamics: A Conceptual Framework for Biomolecular Networks, In: G.Schlosser and G.P.Wagner (Eds), *Modularity in Development and Evolution*, pages 288-311, Chicago University Press.

[8] Harvey, I. and Bossomaier, T. (1997). Time Out of Joint: Attractors in Asynchronous Random Boolean Networks. In: P. Husbands and I. Harvey (eds.), *Proc. of ECAL97*, pages 67–75, MIT Press.

[9] Meseguer, J. (1992). Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(2):73–155.

[10] Martí-Oliet, N. and Meseguer, J. (2002). Rewriting logic as a logical and semantic framework. In: D.M.Gabbay and F.Guenthner (eds), *Handbook of Philosophical Logic (Second Edition)*, Vol. 9, pages 1–87, Kluwer Academic Publishers.

[11] Ciobanu, G., Koutny, M. and Steggles, L. J. (2014). Strategy based semantics for mobility with time and access permissions. *Formal Aspects of Computing*, 27(3):525–549.

[12] Stehr, M-O., Meseguer, J. and P. C. Ölveczky. (2001). Rewriting Logic As a Unifying Framework for Petri Nets. In: H. Ehrig, *et al.* (eds), Unifying Petri Nets: Advances in Petri Nets, LNCS 2128, pages 250–303, Springer Verlag.

[13] Steggles, L. J. (2001). Rewriting Logic and Elan: Prototyping Tools for Petri Nets with Time. Applications and Theory of Petri Nets 2001, LNCS 2075, pages 363-381, Springer Verlag.

[14] Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J. and Sonmez, K. (2004). Pathway Logic: Executable models of biological networks. In: F. Gadducci, U. Montanari (eds.), *Proc. of WRLA 2002*, *Electronic Notes in Theoretical Computer Science*, 71:144–161.

[15] Nigam, V., Donaldson, R., Knapp, M., McCarthy, T. and Talcott, C. (2015). Inferring Executable Models from Formalized Experimental Evidence. In: Computational Methods in Systems Biology 9308, pages 90–103, Springer Verlag.

[16] Clavel, M., Durán,F., Eker, S., Meseguer, J. and Lincoln, P. (1998). An introduction to Maude (beta version). Manuscript, SRI International (March).

[17] Clavel, M. (2002). Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285(2):187–243, .

[18] Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.–E. and Ringeissen, C. (1998). An overview of ELAN. In: C. Kirchner and H. Kirchner (eds), Proc. of WRLA ’98, *Electronic Notes in Theoretical Computer Science*, 15.

[19] Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E. and Reilles, A. (2007). Tom: Piggybacking rewriting on java. In: RTA'07, LNCS 4533, pages 36–47, Springer Verlag.

[20] Eker, S., Martí-Oliet, N., Meseguer, J. and Verdejo, A. (2007). Deduction, Strategies, and Rewriting. *Proc. of STRATEGIES 2006*, *Electronic Notes in Theoretical Computer Science*, 174(11):3–25.

[21] Sen, A. K. and Liu, W. (1990). Dynamic analysis of genetic control and regulation of amino acid synthesis: The tryptophan operon in Escherichia coli. *Biotechnology and Bioengineering*, 35(2):185–194.

[22] Santillán, M. and Mackey, M. C. (2001). Dynamic regulation of the tryptophan operon: A modelling study and comparison with experimental data. *PNAS*, 98(4): 1364-1369.

[23] Simão, E., Remy, E., Thieffry, D. and Chaouiya, C. (2005). Qualitative modelling of regulated metabolic pathways: application to the tryptophan biosynthesis in E. Coli. *Bioinformatics*, 21:190-196.

[24] Lahdesmki, H., Shmulevich, I. and Yli-Harja, 0. (2003). On learning gene regulatory networks under the boolean network model. Machine Learning, 52:147-167.

[25] Thieffry, D. and Thomas, R. (1995). Dynamical behaviour of biological regulatory networks - II. Immunity control in bacteriophage lambda. *Bulletin of Mathematical Biology*, 57:277–295.

[26] *Pathway Logic*, http://pl.csl.sri.com/. Accessed: 30/12/2016.

[27] Eker, S., Meseguer, J., Sridharanarayanan, A. (2004). The Maude LTL Model Checker. In: F. Gadducci, U. Montanari (eds.), *Proc. of WRLA 2002*, *Electronic Notes in Theoretical Computer Science*, 71:162–187.

[28] Clavel, M., *et al. Maude Manual (Version 2.7)* http://maude.lcc.uma.es/manual/maude-manual.html Accessed April 2016.

[29] Schaub, M., Henzinger, T. and Fisher, J. (2007). Qualitative networks: A symbolic approach to analyze biological signaling networks. *BMC Systems Biology*, 1:4.

[30] Mishchenko, A. and Brayton, R. (2002). Simplification of non-deterministic multi-valued networks. In: *ICCAD '02: Proc. of the 2002 IEEE/ACM Int. Conference on Computer-aided design*, pages 557–562.

[31] Oppenheim, A.B., Kobiler, O., Stavans, J., Court, D. L. and Adhya, S. L. (2005). Switches in bacteriophage λ development. *Annual Review of Genetics*, 39:4470–4475.

[32] Thomas, R. (1990). Boolean formalization of genetic control circuits. *Journal of Theoretical Biology*, 42:563–585.

[33] Kauffman, S. A. (1993). *The origins of order: Self-organization and selection in evolution.* Oxford University Press, New York, January.

[34] Tournier, L. and Chaves, M. (2009). Uncovering operational interactions in genetic networks using asynchronous Boolean dynamics. *Journal of Theoretical Biology*, 260:196–209.

[35] Saadatpour, A., Albert, I. and Albert, R. (2010). Attractor analysis of asynchronous Boolean models of signal transduction networks. *Journal of Theoretical Biology*, 266:641–656.

[36] *MVSIS Group. UC Berkeley*: https://embedded.eecs.berkeley.edu/mvsis/, Accessed: 30/12/2016.

[37] Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems – Specification.* Springer.

[38] Steggles, L. J., Banks, R., Shaw, O. and Wipat, A. (2006). Qualitatively modelling and analysing genetic regulatory networks: a Petri net approach. *Bioinformatics*, 23(3):336-343.

[39] Chaouiya, C., Naldi, A. and Thieffry, D. (2012). Logical Modelling of Gene Regulatory Networks with GINsim. Methods in molecular biology (Clifton, N.J.), 804:463-79.

[40] Meseguer, J. and Montanari, U. (1990). Petri nets are monoids. *Information and Computation*, 88(2):105-155.

[41] Bruni, R., Meseguer, J., and Montanari, U. (1998). Internal strategies in a rewriting implementation of tile systems. Electronic Notes in Theoretical Computer Science. Volume 15, Pages 331-352.

[42] Alhumaidan, A. and Steggles, L. J. (2017). Modelling and Analysing Qualitative Biological Models using Rewriting Logic. *Fundamenta Informaticae*, vol. 153, no. 1-2, pp. 1-28.

[43] Chaouiya, C., Naldi, A. and Thieffry, D. (2012). *Logical Modelling of Gene Regulatory Networks with GINsim.*. Methods in molecular biology (Clifton, N.J.). 804:463-479.

[44] Akutsu, T., Kuhara, S., Maruyama, 0. and Miyano, S. (1998). A system for identifying genetic networks from gene expression patterns produced by gene disruptions. Genome Informatics, 9:151-160.

[45] McMillan, K.L. (1993). Symbolic Model Checking. Kluwer Academic Publishers, Boston, MA.

[46] Murata, T. (1989). *Petri nets: Properties, analysis and applications*. Proceedings of the IEEE, 77:541-580.

[47] Shatz, S., Tu, S., Murata, T. and Duri, S. (1996). *An application of petri net reduction for ada tasking deadlockanalysis*. Parallel and Distributed Systems, 7(12):1307-1322.

[48] Nazareth, D. (1993). Investigating the applicability of petri nets for rule-based system verification. IEEE Transactions on Knowledge and Data Engineering, 5(3):402.

[49] Roig, O., Cortadella, J. and Pastor, E. (1995). *Verification of asynchronous circuits by bdd- based model checking of petri nets*. 16th Int. Conf. on Application and Theory of Petri Nets, 935:374-391.

[50] Will, J. and Heiner, M. (2002). *Petri nets in biology, chemistry, and medicine-bibliography*. Computer, pages 1-36, Jan.

[51] Goss, P. J. E. and Peccoud, J. (1998). *Quantitative modelling of stochastic systems in molecular biology by using stochastic petri nets*. Proceedings of the National Academy of Sciences, 95:6750-6755.

[52] Shmulevich, I., Dougherty, E. and Zhang, W. (2002). From boolean to probabilistic boolean networks as models of genetic regulatory networks. Proceedings of the IEEE, 90:1778-1792.

[53] Comet, J. P., Klaudel, H. and Liauzu, S. (2005). *Modeling multi-valued genetic regulatory networks using high-level petri nets*. LNCS, 3536:208-227.

[54] *Petri Net World*, www.informatik.uni-hamburg.de/TGI/PetriNets. Accessed on: 10/1/2017.

[55] Reisig, W. and Rozenberg, G. (1998). *Lectures on Petri nets I: basic models. Advances in Petri Nets*, Lecture Notes in Computer Science 1491, Springer-Verlag.

[56] Murata, T. (1989). *Petri nets: properties, analysis and applications*. Proc. IEEE ,77,5414580.

[57] JUNG, Java Universal Network/Graph Framework. http://jung.sourceforge.net/. Accessed on: 15/3/2017.

[58] Comet, J. P., Klaudel, H. and Liauzu, S. (2004). *Modeling Multi-Valued Genetic Regulatory Networks Using High-Level Petri Nets*. Rapport de Recherche.

[59] Gonzalez Gonzalez, A., Naldi, A., sanchez, L. S., Thieffry, D. and Chaouiya, C. (2006). *GINsim: A software suite for the qualitative modelling, simulation and analysis of regulatory networks*. BioSystems 84, 91-100.

[60] Chaouiya, C. (1995). Dynamical behaviour of biological regulatory networks–II. Immunity control in bacteriophage lambda, ginsim.org/node/47.

[61] Steggles, L. J., Banks, R., Shaw, O. and Wipat, A. (2007). Qualitatively modelling and analysing genetic regulatory networks: a Petri net approach. BIOINFORMATICS, Vol. 23 no. 3 , pages 336–343.

[62] Gschwind, A., Zwick, E., Prenzel, N., Leserer, M. and Ullrich, A. (2001). Cell communication networks: epidermal growth factor receptor transactivation as the paradigm for interreceptor signal transmission. Oncogene, 20:1594-1600.

[63] Kwiatkowska, M., Norman, G. and Parker. D. (2009). PRISM: Probabilistic Model Checking for Performance and Reliability Analysis. Oxford University Computing Laboratory.

[64] Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J. and Sonmez, K. (2002). Pathway Logic: Symbolic Analysis of Biological Signalling. Pacific Symposium on Biocomputing, January 3-7, p400-412.

[65] Chaouiya, C., Naldi, A., Remy, E. and Thieffry, D. (2011). Petri net representation of multi-valued logical regulatory graphs. *Natural Computing*, 10(2):727–750.

[66] Drossel, B., Mihaljev, T. and Greil, F. (2005). Number and length of attractors in a critical Kauffman model with connectivity one. *Physical Review Letters*, 94(8).

[67] D'Silva, V., Kroening, D. and Weissenbacher,G. (2008). A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178.

[68] Emerson, E.A. (1990). Temporal and modal logic. Handbook of Theoretical Computer Science, Chapter 16, the MIT Press.

[69] William, L. K. (2005). Hardware Design Verification: Simulation and Formal Method-Based Approaches.

[70] Clavel, M. and Durán, F. and Eker, S. and Lincoln, P. and Martí-Oliet, N. and Meseguer, J. . Metalevel Computation in Maude, Electronic Notes in Theoretical Computer Science. Volume 15.

[71] Emerson, E. A. (1990). Temporal and modal logic, Handbook of Theoretical Computer Science, Chapter 16.

[72] D'haeseleer, P., Liang, S. and Somogyi, R. (2000). Genetic network inference: from co-expression clustering to reverse engineering. Bioinformatics, 16:707-726.

[73] Hilbert, D. and Ackermann, W. (1928). Principles of Mathematical Logic. Springer-Verlag , ISBN 0-8218-2024-9.

[74] Someren, E., Wessels, L., Backer, E. and Reinders, M. (2002). Genetic network modeling. Pharmacogenomics, 3(4):507-525.

[75] Kauffman, S. (1969). Metabolic stability and epigenesis in randomly constructed genetic nets. J Theor Biol, 2(3):437-467.

[76] Akutsu,T., Miyano, S. and Kuhara, S. (2000). Algorithms for Identifying Boolean Networks and Related Biological Networks based on Matrix Multiplication and Fingerprint Function.

[77] Thomas, R. (1973). Boolean formalization of genetic control circuits. Journal of theoretical biology. Dec. 42:563-585.

[78] Smolen, P., Baxter, D. and Byrne, J. (2000). Modeling transcriptional control in gene networks-methods, recent results, and future directions. Bulletin of Mathematical Biology, 62:247-292, Jan.

[79] Harvey, I. and Bossomaier, T. (1997). Proceedings of the Fourth European Conference on Artificial Life (ECAL97).

[80] McAdams, H. and Arkin, A. (1998). Simulation of prokaryotic genetic circuits. Annual Review of Biophysics and Biomolecular Structure, 27:199-224, Jan.

[81] Klemm, K. and Bornholdt, S. (2005). Stable and unstable attractors in boolean networks. Physical Review E, Jan.

[82] Wuensche, A. (1998) Genomic regulation modelled as a network with basins of attraction. Pacific Symposium on Biocomputing, 3:89-102.

[83] Akutsu, T., Miyano, S. and Kuhara, S. (1999). Identification of genetic networks from a small number of gene expression patterns under the boolean network model. Pacific Symposium on Biocomputing., pages 17-28.

[84] de Jong, H. (2002). Modeling and simulation of genetic regulatory systems: a literature review.. J Comput Biol. 9(1):67-103.

[85] Dassow, v., Meir, G., Munro, E., M., E. and Odell, G. M. (2000). The segment polarity network is a robust developmental module. Nature 406, 188–192.

[86] Albert, R. and Othmer, H. G. (2003). The topology of the regulatory interactions predicts the expression pattern of the Drosophila segment polarity genes. J. Theor. Biol. 223, 1–18.

[87] Hooper, J. E. and Scott, M. P. (1989). The Drosophila patched gene encodes a putative membrane protein required for segmental patterning. Cell 59, 751–765.

[88] Hooper, J. E. and Scott, M. P. (1992). The molecular genetic basis of positional information in insect segments. In Early embryonic development of animals (ed. W. Hennig), pp. 1–49. Berlin, Germany: Springer.

[89] Eaton, S. and Kornberg, T. B. (1990). Repression of ci–d in posterior compartments of Drosophila by engrailed. Genes Dev. 4, 1068–1077.

[90] Taylor, A. M., Nakano, Y., Mohler, J. and Ingham, P. W. (1993). Contrasting distributions of patched and hedgehog proteins in the Drosophila embryo. Mech. Dev. 42, 89–96.

[91] Ingham, P. W. and McMahon, A. P. (2001). Hedgehog signaling in animal development: paradigms and principles. Genes Dev. 15, 3059–3087.

[92] Chaves, M. and Albert, R. (2008). Studying the effect of cell division on expression patterns of the segment polarity genes. J. R. Soc. Interface (2008) 5, S71–S84.

[93] Ingolia, N. T. (2004). Topology and robustness in the Drosophila segment polarity network. PLoS Biol. 2, 0805–0815.

[94] Chaves, M., Sontag, E. D. and Albert, R. (2006). Methods of robustness analysis for boolean models of gene control networks. IEE Proc. Syst. Biol. 153, 154–167.

[95] Sanson, B. (2001). Generating patterns from fields of cells. Examples from Drosophila segmentation. EMBO Rep. 21, 1083–1088.

[96] Chaves, M., Albert, R. and Sontag, E. D. (2005). Robustness and fragility of boolean models for genetic regulatory networks. J. Theor. Biol. 235, 431–449.

[97] Aza-Blanc, P. and Kornberg, T. B. (1999). Ci a complex transducer of the hedgehog signal. Trends Genet. 15, 458–462.

[98] Baker, N. E. (1988). Transcription of the segment-polarity gene wingless in the imaginal discs of Drosophila, and the phenotype of a pupal-lethal wg mutation. The Company of Biologists Limited.

[99] Nosslein-Volhard, C. and Wleschaus, E. (1980). Mutations affecting segment number and polarity in Drosophila. Nature, Lond. 287, 795-801.

[100] Jacinto, A., Alexandre, C. and Ingham, P. W. (1996). Transcriptional activation of hedgehog target genes in Drosophila is mediated directly by the Cubitus interruptus

protein, a member of the GLI family of zinc finger DNA-binding proteins. Genes Dev. 10, 2003–2013.

[101] Jean-Louis, G. ,Grant M. and Olivier M. (2004). Rewriting systems and the modelling of biological systems. Comparative and Functional Genomics, Comp Funct Genom 5: 95–99.

[102] H. Alkhudhayr and J. Steggles. (2017) A Formal Framework for Composing Qualitative Models of Biological Systems. In: C. Martín-Vide, R. Neruda and M. A. Vega-Rodríguez, TPNC 2017, Lecture Notes in Computer Science 10687, pp. 25–36, Springer.

[103] BooleSim. https://rumo.biologie.hu-berlin.de/boolesim/. Accessed on: 15/7/2017.

[104] Biographer.Simulator. https://github.com/biographer/biographer.simulator. Accessed on: 21/9/2017.

[105] The Systems Biology Graphical Notation. https://github.com/sbgn. Accessed on: 18/3/2018.

[106] Data-Driven Documents. https://d3js.org/. Accessed on: 20/3/2018.

[107] Alhumaidan, A. A Support Tool for BooleanNetworks. (2014). CSC8499 Individual Project (Master's Dissertation), Advanced Computer Science MSc. School of Computing Science, Newcastle University.

[108] Macneil, LT, Walhout, AJ. (2011). Gene regulatory networks and the role of robustness and stochasticity in the control of gene expression. Genome Res.c21(5):645-57.

[109] Davidson, E., Levine M. (2005). Gene regulatory networks. Proc Natl Acad Sci, 102:4935.

[110] Elowitz, MB, Leible,r S. (2000). A synthetic oscillatory network of transcriptional regulators. Nature. Jan 20;403(6767):8-335.

Appendix A

Appendix A

In this section we provide search results for the tests we carried out in Chapter 6 in full and as returned by Maude.

A.1 Asynchronous Semantics

A.1.1 Basic model of a Single Cell

CIA turning on from an initial state where all entities are at state 0 using the following search command:

```
Maude> search SLP(1,0)wg(0,0)wg(1,0)wg(2,0)en(1,0)hh(0,0)hh(1,0)hh(2,0)
ptc(1,0)PTC(1,0)ci(1,0)CIA(1,0)CIR(1,0) =>! SLP(1,tSLP:D1)wg(0,twgp:D1)
wg(1,twg:D1)wg(2,twgn:D1)en(1,ten:D1)hh(0,thhp:D1)hh(1,thh:D1)
hh(2,thhn:D1)ptc(1,tptc:D1)PTC(1,tPTC:D1)ci(1,tci:D1)CIA(1,1)CIR(1,tCIR:D1) .
search in StartCellDivisionIndex : ((((((((((CIA(1, 0) CIR(1, 0)) ci(1, 0))
PTC(1, 0)) ptc(1, 0)) hh(2, 0)) hh(1, 0)) hh(0, 0)) en(1, 0)) wg(2, 0))
wg(1, 0)) wg(0, 0)) SLP(1, 0) =>! wg(0, twgp:D1) wg(1, twg:D1) wg(2, twgn:D1)
en(1, ten:D1) hh(0, thhp:D1) hh(1, thh:D1) hh(2, thhn:D1) ptc(1, tptc:D1)
PTC(1, tPTC:D1) ci(1, tci:D1) CIA(1, 1) CIR(1, tCIR:D1) SLP(1, tSLP:D1) .
```

```
Solution 1 (state 4)
states: 5  rewrites: 4 in 0ms cpu (0ms real) (6700 rewrites/second)
twgp:D1 --> (0).D1
twg:D1 --> (0).D1
twgn:D1 --> (0).D1
ten:D1 --> (0).D1
thhp:D1 --> (0).D1
thh:D1 --> (0).D1
thhn:D1 --> (0).D1
tptc:D1 --> (1).D1
tPTC:D1 --> (1).D1
tci:D1 --> (1).D1
tCIR:D1 --> (0).D1
tSLP:D1 --> (0).D1

No more solutions.
states: 5  rewrites: 4 in 0ms cpu (0ms real) (5571 rewrites/second)
```

CIR is typically absent in cells expressing wg. CIR=0 when wg =1. We can check that property using the following search command starting from an initial state where all instances of wg are at state 1:

```
Maude> search SLP(1,0)wg(0,1)wg(1,1)wg(2,1)en(1,0)hh(0,0)hh(1,0)hh(2,0)
ptc(1,0)PTC(1,0)ci(1,0)CIA(1,0)CIR(1,0) =>! SLP(1,tSLP:D1)wg(0,twgp:D1)
wg(1,twg:D1)wg(2,twgn:D1)en(1,ten:D1)hh(0,thhp:D1)hh(1,thh:D1)
hh(2,thhn:D1)ptc(1,tptc:D1)PTC(1,tPTC:D1)ci(1,tci:D1)CIA(1,tCIA:D1)CIR(1,0) .
search in StartCellDivisionIndex : ((((((((((CIA(1, 0) CIR(1, 0)) ci(1, 0))
PTC(1, 0)) ptc(1, 0)) hh(2, 0)) hh(1, 0)) hh(0, 0)) en(1, 0)) wg(2, 1)) wg(
1, 1)) wg(0, 1)) SLP(1, 0) =>! wg(0, twgp:D1) wg(1, twg:D1) wg(2, twgn:D1)
en(1, ten:D1) hh(0, thhp:D1) hh(1, thh:D1) hh(2, thhn:D1) ptc(1, tptc:D1)
```

```
PTC(1, tPTC:D1) ci(1, tci:D1) CIA(1, tCIA:D1) CIR(1, 0) SLP(1, tSLP:D1) .

Solution 1 (state 17)
states: 20  rewrites: 30 in 1ms cpu (1ms real) (15552 rewrites/second)
twgp:D1 --> (1).D1
twg:D1 --> (1).D1
twgn:D1 --> (1).D1
ten:D1 --> (0).D1
thhp:D1 --> (0).D1
thh:D1 --> (0).D1
thhn:D1 --> (0).D1
tptc:D1 --> (1).D1
tPTC:D1 --> (1).D1
tci:D1 --> (1).D1
tCIA:D1 --> (1).D1
tSLP:D1 --> (1).D1

Solution 2 (state 18)
states: 20  rewrites: 30 in 2ms cpu (2ms real) (13692 rewrites/second)
twgp:D1 --> (1).D1
twg:D1 --> (0).D1
twgn:D1 --> (1).D1
ten:D1 --> (0).D1
thhp:D1 --> (0).D1
thh:D1 --> (0).D1
thhn:D1 --> (0).D1
tptc:D1 --> (1).D1
tPTC:D1 --> (1).D1
tci:D1 --> (1).D1
```

```
tCIA:D1 --> (1).D1
tSLP:D1 --> (0).D1


No more solutions.
states: 20  rewrites: 31 in 2ms cpu (2ms real) (12815 rewrites/second)
```

Starting from an initial state where `ci`,`CIA` and `CIR` are all active, we check for the possibility of ending up with en always at state `1` using the following model checking command:

```
Maude> red modelCheck(SLP(1,0)wg(0,1)wg(1,0)wg(2,0)en(1,0)hh(0,0)hh(1,0)
hh(2,0)ptc(1,0)PTC(1,0)ci(1,1)CIA(1,1)CIR(1,1), [] (<> [] aten(1,1))) .
reduce in StartCellDivisionIndex : modelCheck((((((((((((CIA(1, 1)
CIR(1, 1))ci(1, 1)) PTC(1, 0)) ptc(1, 0)) hh(2, 0)) hh(1, 0)) hh(0, 0))
en(1, 0)) wg(2, 0)) wg(1, 0)) wg(0, 1)) SLP(1, 0), []<> []aten(1, 1)) .
rewrites: 13 in 0ms cpu (0ms real) (26315 rewrites/second)
result [ModelCheckResult]: counterexample(nil, {wg(0, 1) wg(1, 0)
wg(2, 0) en(1, 0) hh(0, 0) hh(1, 0) hh(2, 0) ptc(1, 0) PTC(1, 0)
ci(1, 1) CIA(1, 1) CIR(1, 1) SLP(1, 0),deadlock})
```

A.1.2 Extending the Model to 2 Cells

We take a look at the behavioural changes to entities `hh` and `wg` within cell `i`. We start from an initial state where they are both at state `1` and check if they reach a point where they are both at state `0` using the following search command:

```
search init1 =>! SLP(1,tSLP:D1)SLP(2,tSLP2:D1)wg(0,twgp:D1)
wg(1,0)wg(2,twg2:D1)wg(3,twgn:D1)en(1,ten:D1)en(2,ten2:D1)
hh(0,thhp:D1)hh(1,0)hh(2,thh2:D1)hh(3,thhn:D1)
ptc(1,tptc:D1)ptc(2,tptc2:D1)PTC(1,tPTC:D1)PTC(2,tPTC2:D1)
ci(1,tci:D1)ci(2,tci2:D1)CIA(1,tCIA1:D1)CIA(2,tCIA2:D1)
CIR(1,tCIR:D1)CIR(2,tCIR2:D1) .
```

```
Solution 1 (state 169)

states: 212  rewrites: 688 in 21ms cpu (23ms real)

(32544 rewrites/second)

twgp:D1 --> (1).D1

twg2:D1 --> (0).D1

twgn:D1 --> (1).D1

ten:D1 --> (0).D1

ten2:D1 --> (1).D1

thhp:D1 --> (1).D1

thh2:D1 --> (0).D1

thhn:D1 --> (1).D1

tptc:D1 --> (1).D1

tptc2:D1 --> (0).D1

tPTC:D1 --> (1).D1

tPTC2:D1 --> (1).D1

tci:D1 --> (1).D1

tci2:D1 --> (0).D1

tCIA1:D1 --> (1).D1

tCIA2:D1 --> (0).D1

tCIR:D1 --> (0).D1

tCIR2:D1 --> (1).D1

tSLP:D1 --> (0).D1

tSLP2:D1 --> (0).D1
```

A.2 Synchronous Semantics

A.2.1 Basic model of a Single Cell

Searching to check if SLP can turn on from a given initial state init:

```
search init =>+ '__[T0:Term,T1:Term,T2:Term,
'SLP['s_['0.Zero],'1.D1,'1.D1],T4:Term,T5:Term,
T6:Term,T7:Term,T8:Term,T9:Term,T10:Term,T11:Term,T12:Term] .

Solution 1 (state 1)
states: 2  rewrites: 31 in 5ms cpu (5ms real)
(5819 rewrites/second)

T0 --> 'CIA['s_['0.Zero],'0.D1,'0.D1]
T1 --> 'CIR['s_['0.Zero],'0.D1,'0.D1]
T2 --> 'PTC['s_['0.Zero],'1.D1,'1.D1]
T4 --> 'ci['s_['0.Zero],'1.D1,'1.D1]
T5 --> 'en['s_['0.Zero],'0.D1,'0.D1]
T6 --> 'hh['0.Zero,'1.D1,'1.D1]
T7 --> 'hh['s_['0.Zero],'0.D1,'0.D1]
T8 --> 'hh['s_^2['0.Zero],'1.D1,'1.D1]
T9 --> 'ptc['s_['0.Zero],'0.D1,'0.D1]
T10 --> 'wg['0.Zero,'0.D1,'0.D1]
T11 --> 'wg['s_['0.Zero],'0.D1,'0.D1]
T12 --> 'wg['s_^2['0.Zero],'0.D1,'0.D1]
```

A.2.2 Extending the Model to 2 Cells

Searching with the aim of finding a state where both instances of PTC from cells 1 and 2 are at state 1 using the following search:

```
Maude> search init =>+ '__[T0:Term,T1:Term,T2:Term,T3:Term,
'PTC['s_['0.Zero],'1.D1,'1.D1],'PTC['s_^2['0.Zero],'1.D1,'1.D1],T6:Term,
T7:Term,T8:Term,T9:Term,T10:Term,T11:Term,T12:Term,T13:Term,
T14:Term,T15:Term,T16:Term,T17:Term,T18:Term,T19:Term,T20:Term,T21:Term] .
search in StartCellDivisionIndexSync22 : init =>+ '__[T0,T1,T2,T3,
'PTC['s_['0.Zero],'1.D1,'1.D1],'PTC['s_^2['0.Zero],'1.D1,'1.D1],T6,T7,T8,T9,
T10,T11,T12,T13,T14,T15,T16,T17,T18,T19,T20,T21] .

Solution 1 (state 1)
states: 2  rewrites: 45 in 10ms cpu (10ms real) (4286 rewrites/second)
T0 --> 'CIA['s_['0.Zero],'1.D1,'1.D1]
T1 --> 'CIA['s_^2['0.Zero],'1.D1,'1.D1]
T2 --> 'CIR['s_['0.Zero],'0.D1,'0.D1]
T3 --> 'CIR['s_^2['0.Zero],'0.D1,'0.D1]
T6 --> 'SLP['s_['0.Zero],'1.D1,'1.D1]
T7 --> 'SLP['s_^2['0.Zero],'0.D1,'0.D1]
T8 --> 'ci['s_['0.Zero],'1.D1,'1.D1]
T9 --> 'ci['s_^2['0.Zero],'0.D1,'0.D1]
T10 --> 'en['s_['0.Zero],'0.D1,'0.D1]
T11 --> 'en['s_^2['0.Zero],'1.D1,'1.D1]
T12 --> 'hh['0.Zero,'0.D1,'0.D1]
T13 --> 'hh['s_['0.Zero],'0.D1,'0.D1]
T14 --> 'hh['s_^2['0.Zero],'1.D1,'1.D1]
T15 --> 'hh['s_^3['0.Zero],'0.D1,'0.D1]
T16 --> 'ptc['s_['0.Zero],'0.D1,'0.D1]
T17 --> 'ptc['s_^2['0.Zero],'0.D1,'0.D1]
T18 --> 'wg['0.Zero,'1.D1,'1.D1]
T19 --> 'wg['s_['0.Zero],'0.D1,'0.D1]
T20 --> 'wg['s_^2['0.Zero],'0.D1,'0.D1]
```

```
T21 --> 'wg['s_^3['0.Zero],'0.D1,'0.D1]

Solution 2 (state 2)
states: 3  rewrites: 84 in 13ms cpu (13ms real) (6093 rewrites/second)
T0 --> 'CIA['s_['0.Zero],'1.D1,'1.D1]
T1 --> 'CIA['s_^2['0.Zero],'0.D1,'0.D1]
T2 --> 'CIR['s_['0.Zero],'0.D1,'0.D1]
T3 --> 'CIR['s_^2['0.Zero],'0.D1,'0.D1]
T6 --> 'SLP['s_['0.Zero],'1.D1,'1.D1]
T7 --> 'SLP['s_^2['0.Zero],'0.D1,'0.D1]
T8 --> 'ci['s_['0.Zero],'1.D1,'1.D1]
T9 --> 'ci['s_^2['0.Zero],'0.D1,'0.D1]
T10 --> 'en['s_['0.Zero],'0.D1,'0.D1]
T11 --> 'en['s_^2['0.Zero],'1.D1,'1.D1]
T12 --> 'hh['0.Zero,'0.D1,'0.D1]
T13 --> 'hh['s_['0.Zero],'0.D1,'0.D1]
T14 --> 'hh['s_^2['0.Zero],'1.D1,'1.D1]
T15 --> 'hh['s_^3['0.Zero],'0.D1,'0.D1]
T16 --> 'ptc['s_['0.Zero],'1.D1,'1.D1]
T17 --> 'ptc['s_^2['0.Zero],'0.D1,'0.D1]
T18 --> 'wg['0.Zero,'1.D1,'1.D1]
T19 --> 'wg['s_['0.Zero],'1.D1,'1.D1]
T20 --> 'wg['s_^2['0.Zero],'0.D1,'0.D1]
T21 --> 'wg['s_^3['0.Zero],'0.D1,'0.D1]

No more solutions.
states: 3  rewrites: 120 in 17ms cpu (17ms real) (6966 rewrites/second)
```

Checking if SLP and CIA instances in the second cell can become simultaneously active from an initial state where their instances in the first cell are inactive using the following model checking instruction:

```
Maude> red modelCheck( init , <> [] (atSLP(2,'1.D1) /\  atCIA(2,'1.D1)) ) .
reduce in StartCellDivisionIndexSync22 : modelCheck(init,
<> [](atSLP(2, '1.D1) /\ atCIA(2, '1.D1))) .
rewrites: 137 in 16ms cpu (17ms real) (8078 rewrites/second)
result ModelCheckResult: counterexample({'__['wg['0.Zero,'1.D1,'1.D1],
'wg['s_['0.Zero],'1.D1,'1.D1],'wg['s_^2['0.Zero],'1.D1,'1.D1],
'wg['s_^3['0.Zero],'0.D1,'0.D1],'en['s_['0.Zero],'0.D1,'0.D1],
'en['s_^2['0.Zero],'1.D1,'1.D1],'hh['0.Zero,'0.D1,'0.D1],'hh['s_['0.Zero],
'0.D1,'0.D1],'hh['s_^2['0.Zero],'0.D1,'0.D1],'hh['s_^3['0.Zero],'0.D1,'0.D1],
'ptc['s_['0.Zero],'1.D1,'1.D1],'ptc['s_^2['0.Zero],'0.D1,'0.D1],
'PTC['s_['0.Zero],'0.D1,'0.D1],'PTC['s_^2['0.Zero],'1.D1,'1.D1],
'ci['s_['0.Zero],'1.D1,'1.D1],'ci['s_^2['0.Zero],'1.D1,'1.D1],
'CIA['s_['0.Zero],'0.D1,'0.D1],'CIA['s_^2['0.Zero],
'1.D1,'1.D1],'CIR['s_['0.Zero],'0.D1,'0.D1],'CIR['s_^2['0.Zero],'0.D1,
'0.D1],'SLP['s_['0.Zero],'0.D1,'0.D1],'SLP['s_^2['0.Zero],'0.D1,'0.D1]],
'sync} {'__['CIA['s_['0.Zero],'1.D1,'1.D1],'CIA['s_^2['0.Zero],'1.D1,
'1.D1],'CIR['s_['0.Zero],'0.D1,'0.D1],'CIR['s_^2['0.Zero],'0.D1,'0.D1],
'PTC['s_['0.Zero],'1.D1,'1.D1],'PTC['s_^2['0.Zero],'1.D1,'1.D1],'SLP['s_[
'0.Zero],'1.D1,'1.D1],'SLP['s_^2['0.Zero],'0.D1,'0.D1],'ci['s_['0.Zero],
'1.D1,'1.D1],'ci['s_^2['0.Zero],'0.D1,'0.D1],'en['s_['0.Zero],'0.D1,'0.D1],
'en['s_^2['0.Zero],'1.D1,'1.D1],'hh['0.Zero,'0.D1,'0.D1],'hh['s_['0.Zero],
'0.D1,'0.D1],'hh['s_^2['0.Zero],'1.D1,'1.D1],'hh['s_^3['0.Zero],'0.D1,
'0.D1],'ptc['s_['0.Zero],'0.D1,'0.D1],'ptc['s_^2['0.Zero],'0.D1,'0.D1],'wg[
'0.Zero,'1.D1,'1.D1],'wg['s_['0.Zero],'0.D1,'0.D1],'wg['s_^2['0.Zero],
'0.D1,'0.D1],'wg['s_^3['0.Zero],'0.D1,'0.D1]],'sync}, {'__['CIA['s_[
'0.Zero],'1.D1,'1.D1],'CIA['s_^2['0.Zero],'0.D1,'0.D1],'CIR['s_['0.Zero],
'0.D1,'0.D1],'CIR['s_^2['0.Zero],'0.D1,'0.D1],'PTC['s_['0.Zero],'1.D1,
'1.D1],'PTC['s_^2['0.Zero],'1.D1,'1.D1],'SLP['s_['0.Zero],'1.D1,'1.D1],
'SLP['s_^2['0.Zero],'0.D1,'0.D1],'ci['s_['0.Zero],'1.D1,'1.D1],'ci['s_^2[
'0.Zero],'0.D1,'0.D1],'en['s_['0.Zero],'0.D1,'0.D1],'en['s_^2['0.Zero],
'1.D1,'1.D1],'hh['0.Zero,'0.D1,'0.D1],'hh['s_['0.Zero],'0.D1,'0.D1],'hh[
's_^2['0.Zero],'1.D1,'1.D1],'hh['s_^3['0.Zero],'0.D1,'0.D1],'ptc['s_[
'0.Zero],'1.D1,'1.D1],'ptc['s_^2['0.Zero],'0.D1,'0.D1],'wg['0.Zero,'1.D1,
'1.D1],'wg['s_['0.Zero],'1.D1,'1.D1],'wg['s_^2['0.Zero],'0.D1,'0.D1],'wg[
's_^3['0.Zero],'0.D1,'0.D1]],'sync})
```