# The Construction and Verification of Asynchronous Components Built from Chemical Reaction Networks


Max Whitby

Keble College, University of Oxford

Supervisor

Professor Marta Kwiatkowska

A thesis submitted for the degree of

*Master of Science in Computer Science*

Trinity term, 2015

# Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Marta Kwiatkowska, for the insight and guidance she has given throughout this project. Without her vision this project would never have been possible. I would also like to thank Professor Luca Cardelli for establishing most of the theory on which this thesis is based and for our meetings.

I would like to thank Dr.Stefan Dantchev for his trust and confidence in my ability, for providing me with valuable research experience and for aiding in my path to Oxford. Finally I would like to thank my parents, Lynne and Michael, for the overwhelming moral and financial support throughout my masters degree.

# Abstract

The formalism of Chemical Reaction Networks (CRNs) was traditionally used to capture the behaviour of chemical reactions. Recent realisation of CRNs in DNA strand displacement systems makes it possible to view CRNs as a programming language for DNA computing devices. It can thus be reasoned that the construction of logic gates, control flow elements and even algorithms, if realised within CRNs, can be eventually implemented within a laboratory. The problem lies in the feasibility of DNA computation without the use of a global synchronisation method. This thesis proposes novel designs of asynchronous components implemented in CRNs such that they could be realised in DNA without the use of biological oscillators which, to this date, are unreliable for timings. The correctness of the designs is established by systematic analysis of their input-output behaviour using simulation and probabilistic model checking for uniform reaction rates and low molecular count. The proposed designs are an attractive target behaviour for a wide range of biological applications.

# Contents


**1 Literature Review and Definitions** **4**

1.1 Chemical Reaction Networks . . . . . 4

1.1.1 Stochastic Chemical Reaction Networks . . . . . 5

1.1.2 Definition of SCRN . . . . . 6

1.1.3 Restrictions on our Specific SCRN model . . . . . 8

1.1.4 A Diagrammatic Notation for CRNs . . . . . 9

1.2 The Physical Realisation of CRNs . . . . . 11

1.2.1 Non-DNA realisations . . . . . 12

1.2.2 DNA realisations . . . . . 12

1.3 Probabilistic Model Checking . . . . . 14

1.3.1 PRISM . . . . . 15

1.3.2 Definition of a CTMC . . . . . 16

1.3.3 From CRN to CTMC . . . . . 17

1.3.4 CSL . . . . . 17

1.4 Asynchronous Systems . . . . . 18

1.4.1 Asynchronous Architecture . . . . . 19

1.4.2 The C-Element . . . . . 19

1.4.3 Pipelining . . . . . 20

1.4.4 Classification of Asynchronous Circuits . . . . . 22

**2 Methodology** **24**

2.1 CRN Design . . . . . 24

2.1.1 The use of Microsoft's GEC Tool . . . . . 26

2.1.2 The PRISM Model . . . . . 26

2.2 The Verification Process . . . . . 27

2.2.1 Verification of an Isolated Component . . . . . 27

2.2.2 Non-Binary and Changing Values . . . . . 29

2.2.3 The Verification of a Larger System . . . . . 30

**3 Component Design and Verification** **32**
3.1 Rendez-Vous Elements . . . 33
3.1.1 Atomic Operations . . . 33
3.1.2 The Retention of Information . . . 35
3.1.3 Latches . . . 38
3.1.4 Evaluation of Latch Designs . . . 39
3.1.5 Muller-C Element . . . 41
3.1.6 Merging . . . 43
3.2 Arbiter Elements . . . 44
3.2.1 Forks . . . 44
3.2.2 Amplification and Muting . . . 46
3.2.3 Biased Arbiters . . . 47
3.3 Control Flow Elements . . . 48
3.3.1 Conditional Statements . . . 48
3.3.2 Loops . . . 49
3.4 Logic and Numbers . . . 50
3.4.1 NOT, AND and OR . . . 50
3.4.2 NOR, NAND and XOR . . . 51
3.4.3 Numbers . . . 52

**4 Systems, Pipelining and Algorithms** **56**
4.1 Muller Pipeline . . . 56
4.1.1 Construction . . . 57
4.1.2 The Queue . . . 59
4.1.3 Ripple-Carry Adder . . . 60
4.1.4 Loops . . . 61
4.2 Protocols . . . 62
4.2.1 4-phase Dual Rail Protocol . . . 62
4.2.2 2-phase dual-rail Protocol . . . 63

**5 Evaluation** **66**
5.1 Limitations . . . 66
5.1.1 State Space Explosion . . . 66
5.1.2 Automated Verification . . . 67
5.1.3 Multi-purpose circuit design . . . 67
5.1.4 Implementation Issues . . . 67
5.2 Further Work . . . 68

5.2.1 High-Level Programming Language . . . . . 68
5.2.2 Proofs and Exhaustive Testing . . . . . 69
5.2.3 Physical Implementation . . . . . 70

**6 Conclusion** **71**

**A Appendix - Example code** **73**
A.1 AND-Gate Example . . . . . 73
A.1.1 LBS CRN code for an AND gate . . . . . 73
A.1.2 AND-gate SBML File . . . . . 74
A.1.3 AND-gate PRISM File . . . . . 79
A.1.4 AND-Gate Properties for Simulation . . . . . 83
A.2 CRN Designs . . . . . 83
A.2.1 Latches . . . . . 83
A.2.2 C-element . . . . . 85
A.2.3 Adder . . . . . 86
A.2.4 Muller C Pipeline CRN . . . . . 89
A.2.5 Queue Structure . . . . . 91
A.2.6 Loop PRISM model . . . . . 94
A.2.7 Four-Phase Protocol CRN . . . . . 108

**Bibliography** **114**

# Introduction

Chemical Reaction Networks were traditionally used to capture the behaviour of inorganic and organic chemical reactions [42, 60]. Recently, a paradigm shift in the scientific community has seen the use of CRNs extend to that of a high-level programming language [28, 49, 50]. Chemical Reaction Networks are used as a target behaviour for low-level biological systems [51, 54]. As a good example the work of *Chen et al.* has shown the realisation of three types of chemical reaction using DNA strand displacement [14].

With the link between DNA computation and the CRN formalism thus established, it can be reasoned that the construction of logic gates, control flow elements and even algorithms, if realised within CRNs, can be eventually implemented within a laboratory, using CRNs as a target behaviour. Given that DNA computation can interface directly with biological systems, it has been postulated that such research has direct application in medicine, waste removal and other fields [41].

A major problem in the construction of biological systems is the need for a synchronisation methods to ensure that systems perform computation with strict timing. Biological oscillators of any sort are difficult to achieve [24, 59]. Work on the construction of asynchronous components, be it for control flow or for computational logic, has been little researched with few examples of CRNs more complex than just a couple of reactions [51].

This thesis provides novel CRN designs for the construction of asynchronous logic and control flow elements such that this thesis concludes that any asynchronous system can be realised using the components and mechanisms provided. All designs are robust for low molecular count, which is important as it means that implementations without an isochronous fork can also be partially realised. All components are produced with simple reactions and uniform reaction rates, making implementation

of these components even more attractive. Moreover, any design provided in this thesis could potentially be realised within DNA. This is much more concrete than *Magnasco's* existing claim which provides insight into why the realisation of CRNs are Turing Universal [37].

The verification and simulation of our designs are made achievable by the model checker PRISM. PRISM is a model checking tool which allows for the construction of both Discrete-Time Markov Chain (DTMC) and Continuous-Time Markov Chain (CTMC) models [30], the equivalence of which to CRNs has been proven by [8]. PRISM is widely regarded as one of the best free and well documented model checking tools by the community. The constructed CTMC models can be queried by PRISM's own query language based upon Continuous Stochastic Logic (CSL). Implemented CRNs within this thesis are sketched out using Microsoft's GEC tool before they are passed to PRISM [44]. Once our models are constructed in PRISM as a CTMC, we use CSL to query them to confirm that the CRNs exhibit desired behaviour.

The diagrammatic CRN notation used within this thesis is based upon the diagrammatic language constructed by *Luca Cardelli* [10], which this thesis promotes as a concise and convenient way to reason about chemical systems rather than a long list of reactions or processes.

The methodology behind this thesis can be broken down as follows. First, we construct the chemical reaction networks which describe the various asynchronous elements. We then transform these CRNs into an equivalent CTMC model using PRISM's modelling language. We then use these models to simulate and verify stochastically that with high probability the original CRN behaves in an equivalent way to our theoretical logical or control element. This is repeated for all core components listed in *Furber's* book on asynchronous circuit design [55].

The main body of this thesis is explained through an extensive literature review on the current state of the art in relation to this work. Following this, Chapter 2 covers the methodology and an example to explain, in detail, how each stage is realised. Chapter 3 covers the construction of basic asynchronous components which are used in systems in Chapter 4. Chapter 4 covers the constructions and applications of the Muller-C pipeline. The thesis concludes with an extensive evaluation of the work

done and the feasibility of its implementation.

The justification for this work is apparent in its application. In providing components which are independent of a universal clock and operate through a series of local handshakes we aid in the construction and realisation of simpler systems, be it in DNA or in cell-cycle systems. Because of the lack of feasible oscillators, it could be argued that this work provides the first feasible implementation of asynchronous computational components as CRNs, and it is certainly the case that most components discussed within this thesis are completely novel CRN constructions.

# Chapter 1

# Literature Review and Definitions

This chapter gives the necessary background to understand the methodology and relevance of this thesis, explained in subsequent chapters. The chapter starts with an introduction to Chemical Reaction Networks. This allows one to understand the designs presented in Chapter 3: *Component Design and Verification* and Chapter 4: *Systems, Pipelining and Algorithms*. The theory behind Stochastic Chemical Reaction Networks used within this project, along with a definition of diagrammatic notation, is presented and discussed in detail. We go on to discuss the relevance of this by discussing the implementations of CRNs in DNA-based computation. The specific components targeted within this thesis are discussed within the section on *Asynchronous Systems*. The chapter concludes by outlining a framework to convert CRNs into an equivalent Continuous-Time Markov Chain, thus justifying our use of PRISM as a model checker for this project.

## 1.1 Chemical Reaction Networks

Chemical Reaction Networks provide a framework which underpins most of the construction and designs within this thesis. The theory of reaction kinetics can be described very simply: a set of *rules* are imposed upon *reactants* which determine the *products* they form at a rate determined by the present number of reactants [42, 60]. Therefore, reactions provide us with a basic, fundamental computational process: *input* chemicals are transformed into *output* via reaction instruction.

Reaction kinetics is a cornerstone for much of our understanding in both biological and chemical systems. Because of this, work has traditionally focused on *analysis*: the study of trying to simulate and model the behaviour of natural reactions [21, 23]. For instance, *Guptasarma* shows that 80% of the genes in the E. coli chromosome are

expressed at fewer than a hundred copies per cell [26]. At best, we could say that a genetic engineer's job is to modify existing functionality to fit desired functionality, however the computer science approach is that of *synthesis*. Synthesis is the design of specific reactionary behaviour which enacts our target behaviour.

Several works have explored the computational power and limitations of CRNs. *Magnasco* demonstrated that chemical reactions can compute anything that digital circuits can compute, albeit in a brief and informal manner [37]. *Soloveichik et al.* demonstrated that chemical reactions are Turing Universal, meaning that they can compute anything that a computer algorithm can compute [53], and *Soloveichik* himself has contributed much of the work in the formalism of CRNs [53, 17, 13]. What is important to mention is that memory is simulated by polymer memory rather than being included in the CRN, therefore CRNs, within this framework, are not completely Turing complete. In Chapter 3 we expand on this problem by providing reaction networks for memory components.

Such prior work considered the computational power of chemical reactions from a deductive point of view. An important work that considers a ground up approach is [49]. They propose a constructive method for designing specific computational modules: an inverter, an incrementer, a decrementer, a copier, a comparator, a multiplier, an exponentiator, a raise-to-a-power operation, and a logarithm operation in base two. Other work shows the practical construction of *'for'* and *'while'* loops [50]. Some works even go as far as to show signal processing operations such as filtering [28]. The problem with these constructions is that they depend on specific rate categories for reactions. They are also, for the most part, without rewritable memory which could simplify a lot of their designs and constructions.

### 1.1.1 Stochastic Chemical Reaction Networks

A *Chemical Reaction Network (CRN)* is a discrete model of chemical kinetics. The focus within this project will be a subset of this discrete model: *Stochastic Chemical Reaction Networks* (SCRNs). Many results on the formalism of SCRNs have been established [17, 53, 42]; however, we will cover a brief summary of the specific model we use within this thesis. Stochastic Chemical Reaction Networks are closely related to other computational models and indeed other works have shown complete reductions from one to another. Such models include Fractran [16], Vector Addition

Systems (VASs) [29], Petri nets [22], and Register Machines [39], and for many of these systems we can also consider stochastic or non-deterministic variants. Later in the chapter we look at an important result showing a bijection between the CRN framework and Continuous-Time Markov Chains.

In *Cook et al.* an interesting observation about CRNs is made [17]. Given the importance of stochastic behavior in Chemical Reaction Networks, it is particularly interesting that most questions of possibility, concerning the behaviour of CRN models are decidable [29], the corresponding questions of probability are undecidable [62, 53]. This result derives from showing that Stochastic Chemical Reaction Networks can simulate Register Machines [36] efficiently [3] within a known error bound that is independent of the unknown number of steps prior to halting [53]. This results is a conclusion that: when answers must be guaranteed to be correct, computational power is limited, but when an arbitrarily small error probability can be tolerated, the computational power is dramatically increased. This observation is extremely important for the feasibility of this thesis as we observe chemical reaction networks as probabilistic models. In doing so we allow for small tolerance, making most verification questions we ask within this thesis decidable.

### 1.1.2 Definition of SCRN

Stoichiometry is defined as a non-negative number of copies of each species required for the reaction to take place, or produced when the reaction does take place [2, 53].

An SCRN $C$ is a finite set $R$ of reactions acting on a finite number $S$ of species. A reaction is a triple written in the form $< r \in R, \rightarrow_{k_\alpha}, p \in R >$, where r and p are the multisets of species reactants and products, respectively, and $k\alpha > 0$ is the reaction rate. Each reaction $\alpha$ is defined as a vector of non-negative integers specifying the stoichiometry of the reactants, $r_\alpha = (r_{\alpha,1}, \ldots, r_{\alpha,n})$, together with another vector of non-negative integers specifying the stoichiometry of the products, $p_\alpha = (p_{\alpha,1}, \ldots, p_{\alpha,n})$ [2, 53].

As an example take the catalytic reaction:

$$A + B \rightarrow_{k_\alpha} 2C + B \tag{1.1}$$

This means that a consumption of one molecule of species A and one molecule of species B will produce two new molecules of species C and a molecule of species B. $k_\alpha$ is the reaction rate constant where $k_\alpha > 0$. The rate of every reaction $\alpha$ is proportional to the concentrations of reactants so $k$ can be viewed as a constant of proportionality. If this is omitted we can assume the rate constant is uniform. We say that B is *catalytic* as it needs to be present for the reaction to occur but is actually preserved under this reaction. There are two other categories of reaction to be considered: basic e.g. $A + B \to C$ and auto-catalytic e.g. $A + B \to C + 2B$ [2, 53].

A reaction is of the form $A \xrightarrow{X} B$ if there is some reaction $X$ in the SCRN ($C$) that can change A to B, similarly we will use $A \xrightarrow{\cdots} B$ to represent transitive closure [2, 53]. $Pr[A \xrightarrow{X} B]$ indicates the probability that, given that the state is initially A, the next reaction will transition to the state to B. Similarly $Pr[A \xrightarrow{\cdots} B]$ can be seen as the transitive probability that there is a set of reactions which will transform the reactant A into the species B.

The state of the network is defined as a vector of non-negative integers specifying the quantities present of each species, $S = (q_1, ..., q_m)$. A reaction is possible in state $S$ only if there are enough reactants present, that is, $\forall i, q_i \geq r_{\alpha,i}$. When reaction $\alpha$ occurs in state $S$, the reactant molecules are used up and the products are produced. The new state is $S' = S \times \alpha = (q_1 r_{\alpha,1} + p_{\alpha,1}, ..., q_m r_{\alpha,m} + p_{\alpha,m})$. For a given volume $V$, for any state $S = (q_1, \ldots, q_m)$, the rate of reaction $\alpha$ in that state is [2, 53]:

$$\rho_\sigma(S) = k_\alpha V \prod_{i=1}^{m} \frac{(q_i)^{\underline{r_{\alpha,i}}}}{V^{r_{\alpha,i}}} \text{ where } q^{\underline{r}} \stackrel{def}{=} \frac{q!}{(q-r)!} \tag{1.2}$$

Since the solution is assumed to be well-stirred, the time until a particular reaction $\alpha$ occurs in state $S$ is an exponentially distributed random variable with the rate parameter $\rho_\alpha(S)$; the dynamics of a SCRN can be modelled as a continuous-time Markov process, defined as follows:

We write $Pr[S \xrightarrow{C} S']$ to indicate the probability that, given that the state is initially $S$, the next reaction will transition to the state $S'$. These probabilities are given by [2, 53]:

$$Pr[S \xrightarrow{C} S'] = \frac{\rho_{S \to S'}}{\rho_S^{tot}} \text{ where } \rho_{S \to S'} = \sum_{\alpha s.t. S \times \alpha = S'} \rho_\alpha(S) \text{ and } \rho_S^{tot} = \sum_{S'} \rho_{S \to S'} \tag{1.3}$$

The average time for a step $S \to S'$ to occur is $\frac{1}{\rho_S^{tot}}$, and the average time for a sequence of steps is simply the sum of the average times for each step. We write $Pr[S \dashrightarrow S^*]$ to mean, by transitivity, at some point in the future the system will be in state $S^*$ [2, 53].

As well as the formal definition we provide some interesting properties about SCRNs. Well-mixed finite stochastic chemical reaction networks with a fixed number of species, with low error probability, can perform Turing-universal computation [53]. With the addition of two separate reaction rates, fast and slow, SCRNs become Turing universal and can compute any computable function without error [17]. SCRNs can compute any computable function with probability of error less than $q$ for any $q > 0$, but for $q = 0$ universal computation is impossible [3, 29, 53].

SCRNs without reaction rates dependent on the concentration of species remaining are not capable of universal computation with any fixed probability of success [17]. The time and space requirements for Stochastic Chemical Reaction Networks undertaking computation, compared to a Turing Machine, are a simple polynomial slowdown in time, but an exponential increase in space [3, 53].

Realisations of SCRNs have been simulated by partitioning of molecules into membrane compartments [6, 43]. This allows unbounded computation to be performed by molecular systems containing only limited types of enzyme and basic signal-carrying molecular components. Realisations have also been simulated via polymer [5, 46] and via cellular automaton simulation in self-assembly [47]. These approaches rely on the geometrical arrangement of a fixed set of parts to encode information.

### 1.1.3 Restrictions on our Specific SCRN model

There are several important limitations of the SCRN framework, mostly to do with its simplicity. For instance, a problem is that an SCRN, being a simplification of real-world principles, allows for reactions such as $A \to 2A$. This allows for the production of an extra $A$ from nowhere which is a violation of conservation laws.

We cannot immediately assume that our number of molecules will fit within the specific volume, therefore it is important to add a further constraint to our model. For some fixed volume $V$, the number of molecules $m$ at some arbitrary time $t$ should be less than or equal to the number of molecules $m'$ at time $t'$ where $t' < t$. If, however,

we allow our volume to increase then it should scale proportionally to the number of molecules. It is also important to note that in any SCRN, $\Pr[A \xrightarrow{X} B]$ is independent of volume and we will ignore volume as we only care about comparable computation times in this thesis rather than precise timings. This means that we overlook some problems like an increase in volume having a detrimental effect on the well-mixed assumption.

We will also restrict our model further to disallow higher-order reactions, e.g. $2A + 2C \rightarrow 2A + 2B$, as these are generally believed to be approximations of lower order binary reactions with fast reaction rates. Our model assumes that most reactions are reversible unless specifically stated otherwise.

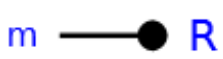
An important restriction in this thesis is that we only consider a restricted class of SCRNs where $\alpha$ is uniform. This means that our rate $k\alpha$ will be uniform throughout our CRN. This is different to previous works which have relied on a change of rate to achieve the creation of chemical components [49].

### 1.1.4 A Diagrammatic Notation for CRNs

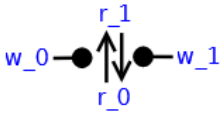
We introduce a diagrammatic notation similar to that of *Cardelli's* notation [10] to describe biological systems. This will aid us in the design and construction of circuits as it relieves the burden of long and tedious lists of CRNs. Given our previous reaction triple $< r, \rightarrow, p >$ we define a reversible reaction triple as $< m, \leftrightarrows, n >$ where:

$$\begin{aligned} < m, \leftrightarrows, n > = & < r, \rightarrow, p > \cup \\ & < p, \rightarrow, r > \end{aligned} \tag{1.4}$$

If $m$ is catalytic or a witness to some generic reaction R we define this diagrammatically as:

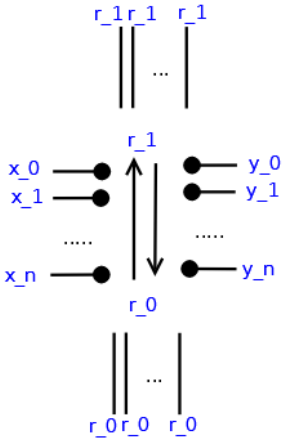



By definition, $m$ can affect a reversible reaction in one of two ways, either by being catalytic to the reaction $r_0 \rightarrow r_1 \in R$ or the reverse $r_1 \rightarrow r_0 \in R$. In the following diagram we show two witnesses $w_0, w_1$ being catalytic to the reaction $r_0 \leftrightarrows r_1$, where $w_0$ is catalytic to $r_0 \rightarrow r_1$ and $w_1$ is catalytic to $r_1 \rightarrow r_0$:




Similarly, if $m$ is a component of the reaction R we denote this as:


Therefore a generic reaction can be represented as follows:



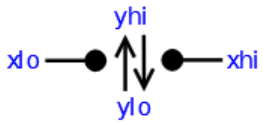

where $x_0, ..., x_n$ and $y_1, ..., y_n$ are catalytic to the reaction $r_0 \leftrightarrows r_1$. Lines attached to the labels $r_0, r_1$ represent that this species is used elsewhere in some other reaction.

The composition of a new reaction R' to a reaction R directly translates to: $R \cup R'$. Similarly, composition of reactions leads to a new state space: $S \times S'$. We define this union diagrammatically as:


Because this diagrammatic language is only defined briefly here, we run through an example. We denote the following CRN, which interestingly emulates the properties of a NOT-gate:

$$\begin{aligned} \text{xlo} + \text{ylo} &\to_{k\alpha} \text{xlo} + \text{yhi} \\ \text{xhi} + \text{yhi} &\to_{k\alpha} \text{xhi} + \text{ylo} \end{aligned} \tag{1.5}$$

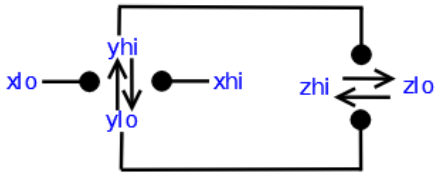
This would be represented as the following diagram:



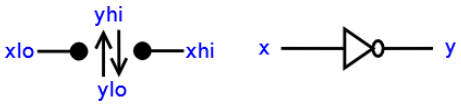

As an example of composition we note the following CRN as the composition of two NOT-gates:

$$\begin{aligned} \text{xlo} + \text{ylo} &\rightarrow_{k\alpha} \text{xlo} + \text{yhi} \\ \text{xhi} + \text{yhi} &\rightarrow_{k\alpha} \text{xhi} + \text{ylo} \\ \text{ylo} + \text{zlo} &\rightarrow_{k\alpha} \text{ylo} + \text{zhi} \\ \text{yhi} + \text{zhi} &\rightarrow_{k\alpha} \text{yhi} + \text{zlo} \end{aligned} \tag{1.6}$$

And so the diagrammatic equivalent would be:




where xlo, xhi would be considered the inputs and zhi and zlo the outputs.
The explanation as to why this emulates a NOT gate is discussed further in Chapter 3. Due to the well mixed assumption we can ignore any preconceptions one might already have about signal carry time. As Chapter 3 progresses we drop the use of specific reactions altogether in favour of standard circuit notation and so a NOT-gate would simply be represented as follows:




## 1.2 The Physical Realisation of CRNs

A small number of works try to work with CRNs as a high-level programming language which compiles down into physical systems. This section divides these up into non-DNA realisations, primarily inorganic reaction systems and cell-cycle systems, and DNA realisations which have traditionally been the most prominent focus for

implementation. DNA realisations will be the main target for our CRNs and implementation of them will be discussed rigorously in Chapter 5.

### 1.2.1 Non-DNA realisations

*Cardelli's* paper on cell cycles [11] shows a replication of the approximate majority algorithm, more specifically that the inactive and active forms of the mitosis promoting Cyclin Dependent Kinases is driven by a system that is related to both the structure and the dynamics of the approximate majority computation. He shows that this process can be described by the following CRN:

$$\begin{aligned} x + y &\to y + b \\ b + y &\to y + y \\ y + z &\to x + b \\ b + z &\to x + x \end{aligned} \tag{1.7}$$

where $x$ and $y$ are competing species and $b$ is some intermediary species. The approximate majority network promotes the dominant species as an output. We show verification of this in regards to Asynchronous Circuits in Section 3.1.5.

### 1.2.2 DNA realisations

Practical work on the use of DNA as a computational method started with the use of UV light projected onto a gel which contained DNA photo-reactants [15] . Other early works include a cell-free transcription-translation system that mimicked the pattern forming program observed in *Drosophila* and amorphous computation with in-vitro transcription networks [25, 52]. The problem with these methods is that they are not purely based on the use of DNA as the sole component of computation.

*Microsoft's* DNA Strand Displacement tool (DSD) provides a programming language to design and simulate devices designed solely in terms of DNA [34]. The design and verification of components made from DNA with DSD has been constructed in conjunction with the PRISM model checker [32]. *Lakin et al.* have designed a ripple-carry adder using an extension of the DSD programming semantics [33]. The theory that underpins DSD is based on strand displacement [34]. Strand displacement is a process in which strands of DNA bind and separate to other DNA molecules.

There are a few works which use restricted classes of strand displacement DNA devices to show implementations of CRNs in DNA. We examine four of them here. The first uses a specific restricted class of double-stranded structures for use in DNA strand displacement [9]. The second uses a primitive called strand-displacement cascades [54]; a modification of the standard strand-displacement reactions. The third, our main focus, generalises these two and provides insight into the compilation of CRNs described as Ordinary Differental Equations (ODEs) [14] because of high molecular count. The fourth takes into account spatial properties of DNA in order to perform computation with low molecular count [18].

*Cardelli* explores the use of a restricted class of DNA strand displacement structures: mainly those that are made of double strands with 'nicks' in the top strand [9]. This paper constructs and verifies an implementation of fork and join gates. As with this thesis, the limitation of this approach is that specific cases are verified for each component leading towards a conclusion that suggests the use of automated analysis to exhaustively prove correctness of components.

The second paper [54] explores the use of the entire compilation process from high level ODE to an implementation in the real world. They provide the reader with a method for compiling an arbitrary CRN into nucleic-acid-based chemistry. This is achieved by the use of a molecular primitive: strand displacement cascades. Through this they argue that any CRN can be implemented by reduction to reactions using this primitive. However, it is important to note that they also suggest that their techniques have not been tested on systems with larger CRNs. They conclude that, because they have provided a method to compile CRNs into DNA molecules, CRNs can be considered as an effective programming language and used prescriptively for the synthesis of unique molecular systems. They extend on this work further to show implementations of CRN oscillators [54].

Arguably the most important work to date on this subject [14] shows an equivalence between all three of our base reaction classes: non-catalytic, catalytic and auto-catalytic, and a DNA reaction mechanism [14]. They suggest a way to compile any CRN or targeted behaviour into an equivalent DNA architecture. This DNA architecture can be realised by a transformation into a mechanical strand displacement model. We have included a figure (1.1) of this process for a non-catalytic reaction. This paper

shows the validity of the work completed in this thesis. Because any chemical reaction can be compiled into an equivalent DNA reaction mechanism we can essentially realise any high-level CRN suggested in Chapter 3 and 4. The paper also shows an implementation for approximate majority which is used extensively within this thesis.

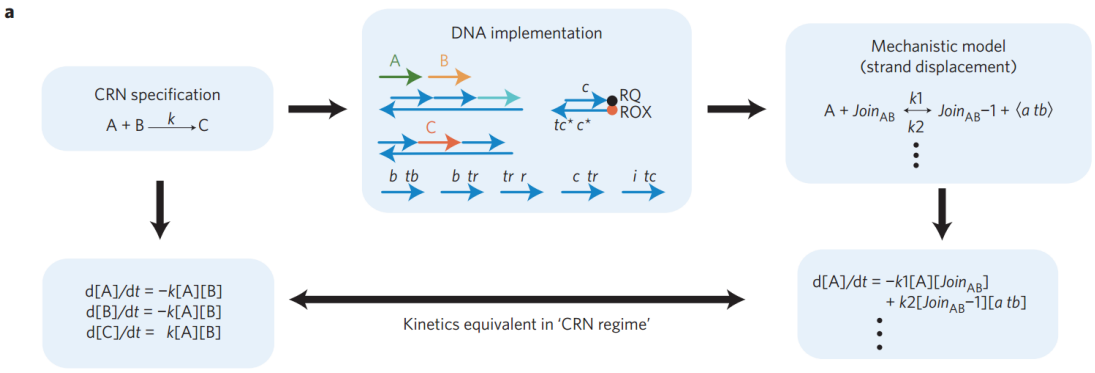



Figure 1.1: From:*Programmable Chemical Controllers Made from DNA* [14]. The formalism of CRNs is used as a programming language that specifies the desired behaviour mainly: $A + B \rightarrow C$. This is turned into a DNA architecture. This architecture is turned into a mechanistic model which is compared to the equivalent ODE behaviour given by the CRN.

The fourth work [18], the most recent, suggests the failure of the other three is that they work with high molecular count under the well-mixed assumption. This paper proposes a method for programming spatial organisation of DNA at a small scale [18]. This is achieved by means of a DNA strand displacement reaction diffusion system. They use this to demonstrate three practical components. An implementation of an auto-catalytic reaction, mainly: $A + B \rightarrow 2B$, a 'predator-prey' oscillator, and a two-species consensus network. For each of these implementations they compile a high level CRN implementation down to a specific DNA implementation which they simulate using *Microsoft's DSD* tool.

## 1.3 Probabilistic Model Checking

This thesis uses Probabilistic Model Checking extensively in verifying correctness of CRN designs in Chapters 3 and 4. This section explains the theory behind realising a CRN as a PRISM model and how we query this model. We also discuss the use of PRISM as the specific model checker we use.

Model checking is an automated formal verification technique, based on the exhaustive construction and analysis of a finite-state model of the system being verified [4]. A model is a labelled state-transition system, in which each state represents a possible configuration of the system and each transition between states represents a possible evolution from one configuration to another.

Probabilistic model checking is a generalisation of model checking for the verification of systems that exhibit stochastic behaviour [31]. In this thesis, our SCRNs have reactants or "species" with attached rates which determine the likelihood of their reaction with other species. This behaviour can be described stochastically.

To model systems of reactions at a molecular level, the appropriate model are continuous-time Markov chains (CTMCs) [19], in which transitions between states are assigned rates and molecular concentrations.

### 1.3.1 PRISM

PRISM is a probabilistic model checking tool developed by a shared grant between the Universities of Birmingham and Oxford and is maintained by David Parker. It provides support for analysis upon continuous-time Markov chains (CTMCs) which characterises the behaviour of our SCRN state space. Models are specified in a simple, state-based language based on guarded commands [30]. PRISM provides a conversion process between SBML and its own modelling language which opens a gateway to the construction models from higher level biological languages including *Microsoft's GEC* language, which we use extensively within this thesis. Other high level conversions from stochastic process algebras and stochastic $\pi$ - calculus have also been developed [30].

PRISM is used within this thesis to construct models using its in-built model checking language [30]. As an example of this language please observe Appendix A.1.3: a PRISM example describing a logical AND gate. Simulation data from PRISM is used heavily throughout this thesis and, through equivalences referenced in this chapter, the reader will be satisfied that PRISM can successfully run accurate simulations upon a CTMC equivalent to an SCRN model.

We use PRISM because we are interested in components with low molecular count. It is viable to create a discrete model from which we can query and simulate properties because realisations of CRNs with low molecular count generate a small state space.

### 1.3.2 Definition of a CTMC

Traditionally, analysis has focused on mass action kinetics, where reactions are assumed to involve sufficiently many molecules that the state of the system can be accurately represented by continuous molecular concentrations with the dynamics given by ordinary differential equations [14]. However, analyzing the kinetics of small-scale CRNs involving a finite number of molecules requires stochastic dynamics that explicitly track the exact number of each molecular species [17].

Many ways to simulate kinetics of chemical systems have been established. The most common and, at present, successful attempts involve simulating Boolean circuits [20, 48, 56, 57]. In this case, information is generally encoded in the high or low concentrations of various signaling molecules. This thesis focuses on simulating the kinetics of a CRN as a CTMC. We now define a CTMC formally.

Let $\mathbb{R}_{\geq 0}$ denote the set of non-negative reals and $AP$ denote a fixed, finite set of atomic propositions used to label states with properties of interest, a CTMC is a tuple ($S$, $\mathbf{R}$, $L$) where [31]:

- $S$ is a finite set of states;
- $\mathbf{R}$: (S $\times$ S) $\rightarrow \mathbb{R}_{\geq 0}$ is a *transition rate matrix*
- $L : S \rightarrow 2^{AP}$ is a *labelling* function which associates each state with a set of atomic propositions.

Each pair of states is assigned rates which are used as parameters of the exponential distribution [31]. A transition can only occur between states $s$ and $s'$ if $\mathbf{R}(s, s') > 0$ and the probability of the transition being triggered within $t$ time-units is $1-e^{-\mathbf{R}(s,s')\cdot t}$ [31]. Typically, in a state $s$, there is more than one state s' for which $\mathbf{R}(s, s') > 0$; this is known as a race condition and the first transition to be triggered determines the next state [31]. The time spent in state $s$ before any such transition occurs is exponentially distributed with the rate $E(s) = \sum_{s' \in S} \mathbf{R}(s, s')$ , called the exit rate. The probability of moving to state $s'$ is given by $\mathbf{R}(s, s')/E(s)$ [31].

A CTMC can be augmented with rewards, attached to states and/or transitions of the model. Formally, a reward structure for a CTMC is a pair $(c, C)$ where: [31]

- $c : S \to \mathbb{R}_{\geq 0}$ is a state reward function.
- $C : (S \times S) \to \mathbb{R}_{\geq 0}$ is a transition reward function.

State rewards can represent either a quantity at a particular time instant (e.g. the number of molecules representing the species "outHi" currently in the system) or the rate at which some measure accumulates over time (e.g. changes in the number of molecules). Transition rewards are accumulated each time a transition occurs and can be used to compute the number of molecules of a species over a particular time period.

### 1.3.3 From CRN to CTMC

The stochastic behaviour of both processes and reactions can be modelled as CTMCs [7, 27]. For much greater detail on this subject see the papers by *Anderson et al.* [1] and *Cardelli* [8]. In both of these papers they provide an in depth mathematical formalism. In *Cardelli*'s paper he proves the equivalence of both by first converting an SCRN into a Continuous Chemical System which is a specific semantic described within the paper. From this he provides an algorithm to convert a Continous Chemical System into a Labeled Transition Graph (LTG). He then uses an equivalence between an LTG and CTMC to complete the conversion. In this thesis we build on this theory by modelling our CRN as a PRISM model. The construction of which is explained in greater detail in the next chapter.

### 1.3.4 CSL

Once we have constructed our PRISM model we need to query properties to determine its correctness. The desired correctness properties of CTMCs are typically expressed in temporal logics, such as CTL (Computation Tree Logic) or LTL (Linear-time Temporal Logic) [4]. We extend this for reward-based properties. CSL is a branching-time, CTL-like temporal logic where the state formulas are interpreted over states of a CTMC. It adopts operators of PCTL, like a time-bounded until operator and a probabilistic operator asserting that the probability for a certain event meets given bounds [31].

CSL gives us greater power over CTMCs than LTL or CTL. For example, rather than verifying that "the species outLo always eventually reaches a state where it has 10 molecules", using CSL allows us to ask "with what probability do we enter a state where the species outLo has 10 molecules?" or "what is the probability that the species changes within time $t$?". Reward-based properties include "what is the expected time for a species to reach 10 molecules?". For further details on probabilistic model checking of CTMCs, see for example. For a description of the application of these techniques to the study of biological systems, see [19].

We express some examples of CSL which we use within our project:

1. $\forall[\Box!(outHi \wedge outLo)]$ - species representing high and low output signals are never present together.
2. $\forall[\Diamond END]$ - All states eventually reach a termination state labelled END.
3. R{"rewardoutLo"} $=?[I = t]$ - What is the value of the reward structure set up for the species outLo at time $t$.

With our properties we perform an exhaustive analysis of our model. For each property either concluding that it is satisfied or, if not, providing a counterexample illustrating why it is violated.

## 1.4 Asynchronous Systems

This thesis primarily focuses on the implementation and verification of asynchronous components as CRNs. The reader may be unfamiliar with asynchronous computers purely because of their lack of use in general purpose computers, however they are used frequently in mobile devices [45]. The primary reason for using asynchronous components rather than their synchronous counterparts is the lack of clock. While a lot of work has gone into the construction of biological oscillators, these are unreliable and unstable [24, 59]. In [40] they show the construction of an oscillator using DNA-based computation but under very restricted circumstances. We instead wish to focus on what can be achieved without the use of a clock, given that asynchronous machines are also Turing complete [38]. This section provides a short introduction into common elements used within asynchronous systems.

### 1.4.1 Asynchronous Architecture

Asynchronous digital circuits provide a systems engineer with a low-power, without-clock alternative to their synchronous counterpart [55]. This comes with the downside that circuit design can, in some cases, be more complex. Asynchronous components rely on 'local cooperation' rather than a governing clock. This is usually in the form of handshaking protocols. These protocols exchange completion signals in order to establish when an action has finished.

In this thesis we work with a dual-rail implementation of asynchronous circuits. What this means is that there is a separate rail or 'signal' for high and low (or 1 and 0). This is because we cannot detect when there are no molecules in a species so an individual species can only represent one value. Circuit design follows the normal rules in which components are connected by rails which transport data around the system. In our case this transportation is of molecules. We will assume that the reader has familiarity with logic gates.

### 1.4.2 The C-Element

Asynchronous circuits rely heavily on latches and rendez-vous elements. A rendez-vous element is a component which 'waits' on two or more actions to complete before a system continues. One form of rendez-vous element is the Muller C-element named after David Muller [55, p.5]. Essentially, like a latch, it is a gate which retains a state. A C-element has two inputs and one output. When both inputs are low the output is low. Similarly, when both inputs are high the output is high. The variation from a normal gate however, is, if the inputs are high, or low, and one of them changes, it 'remembers' the last high, or low, state. Essentially, it remembers the last pure 0 or 1 state [55, p.8].

C-elements allow a circuit to be speed independent by a series of local handshakes. This means that we can wait for longer computational paths to complete before advancing without additional computation occurring, negating the use of a system clock. This leads to the creation of what is called Muller pipelining, explained within the next section. In Chapter 3 we introduce other CRN rendez-vous elements along with a CRN for the Muller-C element. C-elements have been conceptualised in DNA [41].

Another key asynchronous component is an arbiter [58]. An arbiter chooses which signal to allow, breaking any ties that competing signals may have. We show the role of the arbiter in Chapter 3, the designs for which in CRNs are relatively simple. An arbiter is sometimes refereed to as a Buridan Arbiter [58], after the 14th century French philosopher who suggested that if an ass was placed between two equal piles of hay it may starve to death as it will not know which one to pick. We discuss how to avoid such issues.

### 1.4.3 Pipelining

Asynchronous systems have no universal clock. Instead, they rely on a system of localised handshaking. In hardware this is in the form of handshaking between neighbouring registers. Some of the biggest problems with the correct operation of asynchronous circuits are: data doesn't disappear, data needs to preserve ordering along lines and data doesn't randomly create itself [55, p.18]. We will assume these problems have solutions for our domain and discuss the 2-phase and 4-phase transfer protocols.

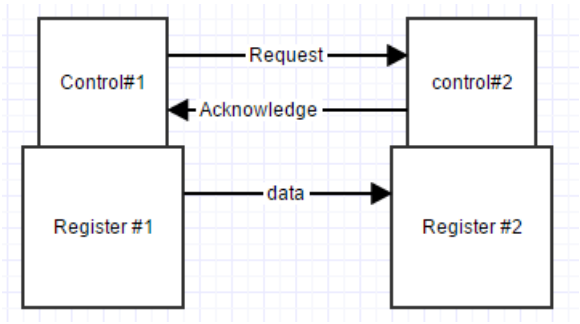



Figure 1.2: A diagram showing hand shaking between two control components, allowing data to be transferred from register 1 to register 2.

We consider the diagram in Figure 1.2. In this simple example, data, request and acknowledge rails can be set to high or low. The four phase protocol is as follows: firstly the sender sends data and sets request to high. The receiver then writes the data to the register and sets acknowledge to high. Then the sender responds by setting request to low and finally the receiver acknowledges this by setting acknowledge to low. Because there are four points to note here, this protocol is known as the 4-phase transfer protocol. We then repeat this ad infinitum. Some text books also refer to this protocol as the RTZ protocol (return to zero).

In a 2-phase handshaking protocol we use the same idea [55, p.18] however there is now no difference between transitions on the request and acknowledge wires. i.e. the transition from high to low is the same as the transition from low to high on each of these wires. We can see how this now leads to two events: (1) the sender sets data and request, (2) the receiver stores the data and sets acknowledge to high. Interestingly, although the second protocol seems more streamlined it is generally disputed as to which is more efficient due to the increased complexity of parts for the 2-phased protocol [55, p.18].

In our example, data is sent from a left sender to a right receiver but we can reverse the direction of data flow. This is called a push channel and the opposite is pull. Also we are able to remove the data wire all together to have a pure handshaking protocol circuit [55, p.18].

We do not need to restrict ourselves to three lines for one single bit. We can merge the request and data lines in what is known as a dual rail protocol. In the 4-phase dual rail protocol, we have 2 wires per bit of information which gives us 4 logical possibilities as follows:

| Data #1 | Data #2 | Meaning |
|---|---|---|
| 0 | 0 | No data |
| 1 | 0 | logical 0 |
| 0 | 1 | logical 1 |
| 1 | 1 | Not defined |

In having an additional codeword for no data we essentially bypass the need for a request line. This leads to an updated 4-phase protocol: the sender issues a valid codeword, the receiver absorbs the codeword and sets acknowledge high, the sender responds by issuing the empty codeword and the receiver acknowledges by setting acknowledge to low. Note a valid codeword is when ALL data lines have sent something. This protocol has the advantage of being delay insensitive.

The 2-phase dual rail is similar but the information is again encoded as transitions from $1 \rightarrow 0$ and $0 \rightarrow 1$. The main difference is that we acknowledge after one data codeword. For both protocols we can extend our examples to n-rails simply by taking our wire pairs and placing them in parallel n times. For n rails we get $2^n$ valid

codewords.

The implementation of these protocols can be implemented using the Muller C-Element described in the previous section. The Muller pipeline propagates handshakes. We can view this as a propagation in serial or along the x-axis. Each element propagates a 1 only if its predecessor is 1 and its successor is 0, hence creating a ripple or an oscillation. For a clearer explanation of this please see Figure 1.3.

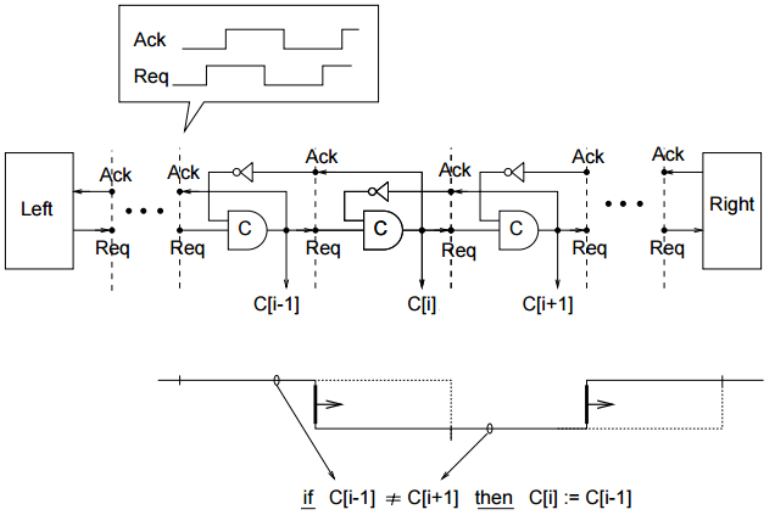



Figure 1.3: Signals are propagated from left to right using a Muller Pipeline. The pipeline effectively queues data, only allowing a transition to occur when a further signal has been acknowledged. This image was taken from Furber's book: *Principles of Asynchronous Circuit Design*, page 17. [55].

Using this pipeline and other control pipelines can allow for construction of most systems that previously would rely on a system clock. The construction and verification of the Muller Pipeline, Queues and Ripple-Carry adders as CRNs is discussed further in Chapter 4.

### 1.4.4 Classification of Asynchronous Circuits

There are three classifications of asynchronous circuits in terms of delays: speed-independent, delay-insensitive and quasi-delay-insensitive [55]. Speed independent systems are circuits where there are no delays within and between components, virtually impossible to consider within this thesis. A circuit that operates correctly but with unknown delays is known as delay-insensitive [55]. We achieve this classification of circuits by simulating forks as components rather than natural wire divisions,

demonstrated in our fork implementation in Section 3.2.1. We can achieve quasi-delay-insensitive systems through the CRN well-mixed assumption. The well-mixed assumption assumes that molecules have equal probability of interacting with other molecules within a system providing that the other molecules have equal molecular count. However the well mixed assumption does not hold for systems of low molecular count and so is not used within this thesis. If this assumption did hold we could use a natural CRN fork such as:

$$\begin{aligned} \mathrm{x} &\rightarrow_{k\alpha} \mathrm{y} \\ \mathrm{x} &\rightarrow_{k\alpha} \mathrm{z} \end{aligned} \tag{1.8}$$

# Chapter 2

# Methodology

The overall aim of the project is to explore the feasibility of constructing asynchronous machines from Stochastic Chemical Reaction Networks (SCRNs). In order for a component to be considered feasible each CRN was converted to a PRISM model and then queried using PRISMs query-language closely resembling CSL. This chapter explores this process from the choice of CRN through to the verification of it as an isolated component and its use in larger control flow structures.

This chapter, as well as giving the reasoning behind our decisions, works alongside an example, in this case an AND-gate. For readers unfamiliar with such a logic gate an AND-gate is a construct with two binary inputs $x, y$ and a binary output $z$. When $x$ and $y$ have the value 1, $z$ has the value 1, else $z$ has the value 0. Please refer to the Appendix A.1 - AND-gate example to follow the code as well as the diagrammatic language used.

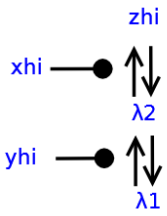
## 2.1 CRN Design

The first stage of our process is to design a component. This stage is divided into three parts. The first is to establish a simple mechanism that underpins such a component.

Secondly, with this simple mechanism in mind we attempt to construct a simple CRN that emulates this mechanism. This construction has to obey reactions rules defined in the previous chapter, most importantly, that the CRN is built from the three basic reaction types, the construction has to conserve overall molecular count and the construction is dual-rail. Lastly, we add extra reactions to account for other inputs

or to stabilise the process.

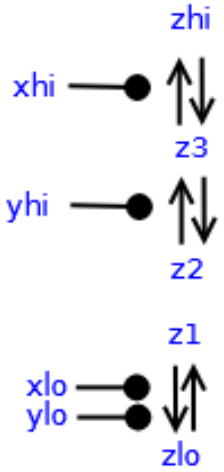
In the example of the AND-gate we first wish to construct a mechanism that will output a high signal if both of our input signals are high. A mechanism that captures such a process might look like the following:



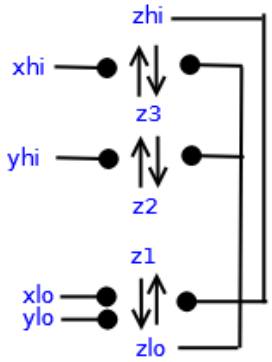

For readers still unfamiliar with our diagrammatic language this equates to the following CRN:

$$\begin{aligned} \text{yhi} + \lambda_1 &\rightarrow_{k\alpha} \text{yhi} + \lambda_2 \\ \text{xhi} + \lambda_2 &\rightarrow_{k\alpha} \text{xhi} + \text{zhi} \end{aligned} \tag{2.1}$$

Next we add extra inputs to complete the gate design for all logical inputs:




Note we use $\lambda$ to represent species or states that are not important to us. They are simply intermediary to the output species. Finally, we add further reactions which will change the output of the gate if one of the inputs changes, essentially creating an adaptive gate. The completed design for our AND-gate looks as follows:




### 2.1.1 The use of Microsoft's GEC Tool

Once we construct our design we wish to perform some preliminary tests as the conversion process to a PRISM model takes considerable time. For this we use the *Microsoft GEC* tool. This tool allows us to simulate CRNs in a primitive, isolated fashion to see if the stochastic behaviour of a CRN approximately behaves as desired. Appendix A.1.1 shows the AND-gate example written in Microsoft's GEC-LBS language.

The second function that GEC provides is a conversion to Systems Biology Markup Language (SBML), which in turn has a conversion to PRISM's Modelling Language. While this conversion process often failed it, in theory it allowed us insight into how the PRISM model should be structured. Appendix A.1.2 shows the SBML file for the AND-gate example and similarly Appendix A.1.3 shows the PRISM modelling language conversion.

### 2.1.2 The PRISM Model

As discussed in the literature review, we can construct a CTMC model from a CRN. We specify this CTMC in PRISM's modelling language. This specification is almost entirely automated, except for the occasional error, converted from the CRNs designed in *Microsoft's GEC-LBS* language. We use PRISM as we are specifically testing components that interact with low molecular count.

In our model transition rates mimic those of CRN kinetics, as in rates are proportional to the number of molecules present in that species. For example, if a species had a low reaction rate it may interact quite slowly with other species in the model. Each reactant or "species" has an initial molecular count and an initialised range of possible

values. The range of possible values ranges between whatever is defined as a high and low signal. Please refer to Appendix A.1.3 for the example CRN AND-gate in PRISM's modelling language.

## 2.2 The Verification Process

The majority of time spent on this project was verifying whether components had the expected behaviour. This section informs the reader on the steps undertaken to simulate and verify these components. An important extension, which was infeasible for a Master's Thesis, would be to automate the entire verification process and prove that these components behave as desired by some exhaustive means. This is discussed further in Chapter 5: *Evaluation*.

We use simulation-based verification in which we attach rewards to our PRISM model. These rewards track molecular count of a species over time. Each simulation plot that appears in this thesis is actually several simulations of rewards of different species on the same plot. For instance the plot in Figure 2.1 shows the output of the reward structures at some time $t$ tracking the species xhi, xlo, yhi.. etc. Because the rates are stochastic these plots could be viewed as the expectation of a species at $t$. Included in Appendix A.1.4 is a typical properties file so that these experiments can be replicated. We also verify properties of our model such as "does the species xhi always exhibit a high signal after time $t$"?. These were verified using the CSL query language demonstrated in the previous chapter.

### 2.2.1 Verification of an Isolated Component

The first stage of the verification process is to examine our component as an isolated model in PRISM. We iterate through all logical input values for a component by adjusting the starting concentrations of each molecular species to exhibit various logical 'high' or 'low' signals. Once we have produced models representing all of the possible configurations of these input signals we run simulations of each model to test that they indeed have the intended target behaviour on each input. We also examine the probabilities that they will have reached a certain state within some time $t$.

In the example of our AND-gate we would have four models for the inputs: 00, 01, 10, 11. We discuss how many molecules constitutes a low or high signal at the end of this chapter but for now we assume a high signal, or species, is 10 molecules and

a low signal is 0. We then run simulations on all of these inputs. In Figure 2.1 we see one of these simulations which shows the expected concentration of that species at time $t$. We have 7 species: the 4 input signals xhi,xlo,yhi and ylo, the 2 output signals zlo and zhi, and the internal species z1, z2 and z3. Initially yhi and xhi are set to a value of high and eventually zhi changes to high so this simulation produces the target behaviour we expected. The next case in Figure 2.2 shows the behaviour of the system when the input is x:0,y:1 and im the corresponding model xlo and yhi are set to high. We see that indeed the expected output is 0 or zlo. The case of y:0, x:1 is symmetric. In Figure 2.3 we conclude by showing the output zlo is high when both inputs are 0. We have now exhaustively simulated our gate design on all inputs.

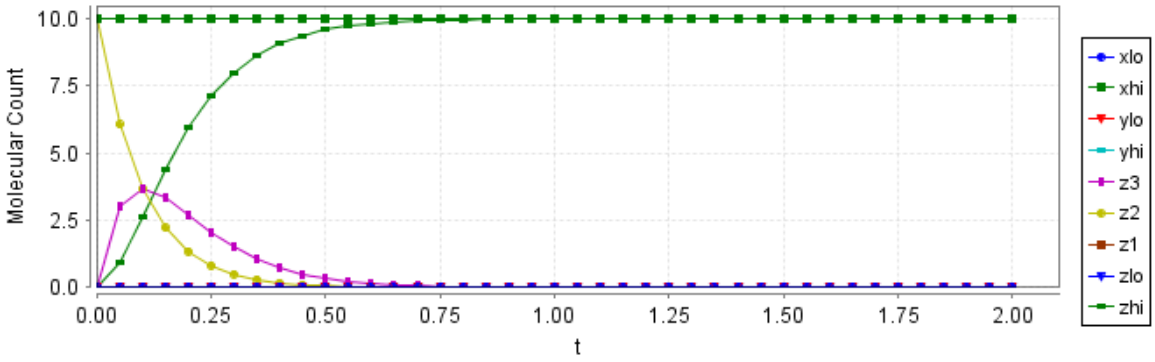



Figure 2.1: A simulation performed on a model equivalent to an AND-gate with both inputs set to high (xhi, yhi). Notice how the output signal zhi changes to high in response to this.

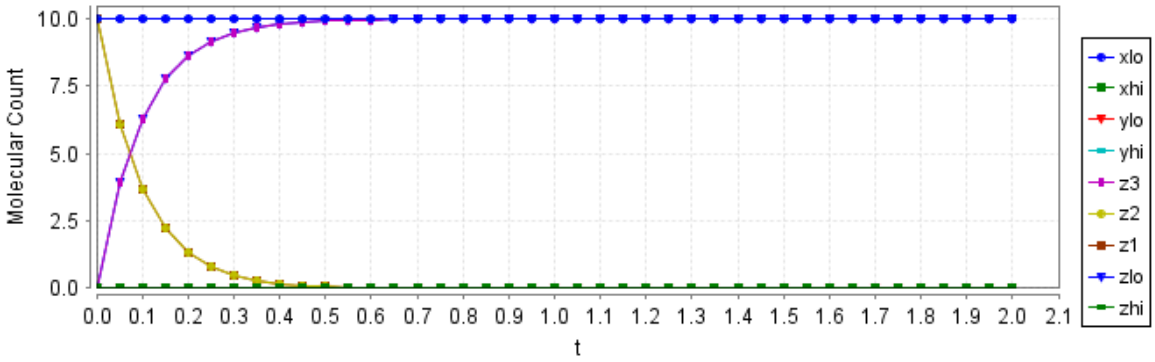



Figure 2.2: Again we look at a simulation performed on our AND gate but with different starting concentrations. xlo has now replaced xhi making the input x:0, y:1. zlo is now high and zhi is now low which is the expected behaviour of an AND-gate given these inputs.

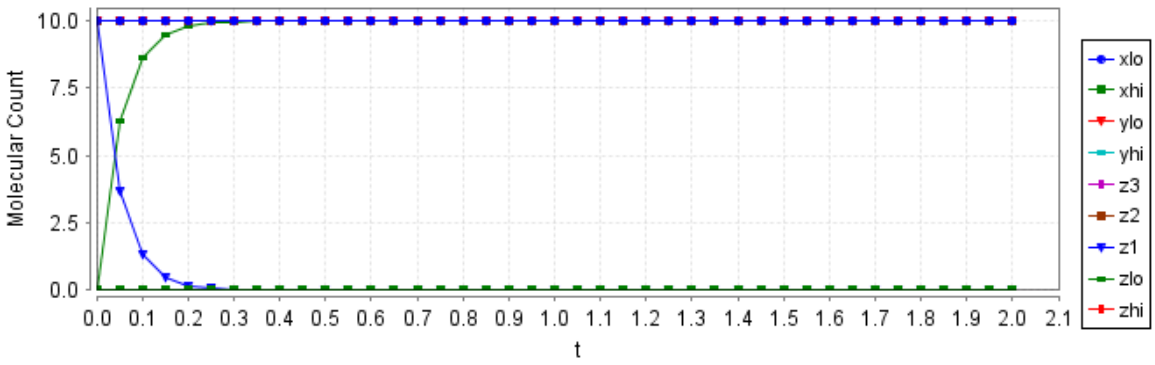



Figure 2.3: We then show the same and gate with inputs x:0,y:0 to show that, exhaustively, our AND-gate has the correct behaviour on all inputs.

### 2.2.2 Non-Binary and Changing Values

Once the component has been simulated in isolation we examine how a component might behave in a larger system, mainly through a change in input. This is done in two ways. First we examine how a component behaves with 'half-values', or signals that are neither purely high or low. In any CRN simulation if the input changes there will be at least some point where the value of that input is half the value that it will eventually become. For instance consider two inputs which at low are 0 molecules and at high are 10 molecules. Initially we set one to low and the other to high, and then reverse the input. This will mean at some point the configurations of the inputs will be x:0,y:10 ; then x:3,y:7; then x:5, y:5 and so on. We take into account some of these intermediary stages to see if the internal and output values of the gate behave in the way we would expect. In Figure 2.4 we see one of these simulations for our AND-gate where xhi, yhi are set to 7 and xlo, ylo are set to 3. As we can see the signal zhi has started to reach its peak and so the gate appears to operate correctly under these conditions. We would expect if the signals xhi and yhi were on the rise then the output zhi would also follow.

The second, more rigorous test, which eventually replaced the above, is to introduce new reactions to emulate a signal change. For instance, if we wished to change a carrier signal from high to low we would introduce a reaction:

$$\text{xhi} \rightarrow_{k\alpha} \text{xlo} \tag{2.2}$$

where we convert all of the signal xhi into a signal xlo. We also varied the rate $k\alpha$ considerably to allow the component to stabilise first before changing the input. This variable input technique is superior because the component is allowed to stabilise;

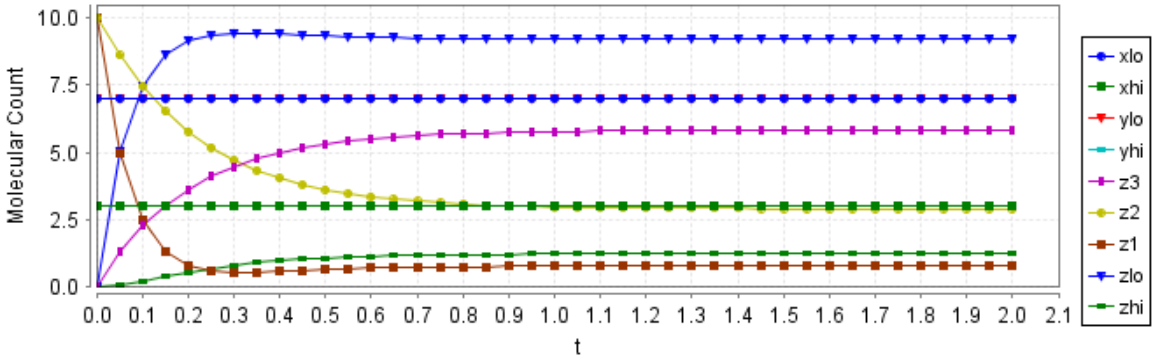



Figure 2.4: In our AND-gate we test for semi-high and semi-low signals to mimic external changes within our system. In this example we see both xhi and yhi at 7/10 high and xlo and ylo at 3/10 low. We observe that the output species zhi is tending towards a 'high' signal.

however, constructing and verifying a model in this way takes considerably more time. We show an example, using the above reaction, on our AND-gate, in Figure 2.5. Notice how the output responds to this change and zhi is converted to zlo.

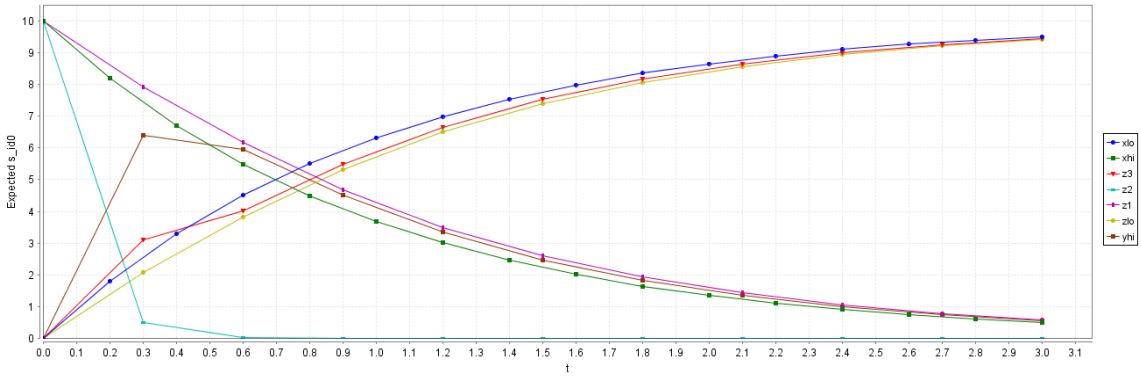



Figure 2.5: In this more complicated test we change the value of xhi to xlo during testing. This allows us to simulate a change in external input during testing.

### 2.2.3 The Verification of a Larger System

The verification of larger systems was completed in much the same way. First we would consider a system as an isolated entity before adjusting and changing inputs to measure a response. However, there were several major limiting factors.

As the number of reactions increases, the state space increases exponentially. Because of this we had to consistently scale the number of molecules in the larger systems to

avoid this. All tests were run on a standard desktop computer and some of the more complex queries took upwards of two days. A reasonable upper bound would be 100 molecules per system; after this testing becomes impractical with limited resources both in time and in computing power.

Another major problem was the implementation itself. When dealing with components in conjunction it was hard to keep track of the reactions governing each component. We therefore wrote a python script that would rename the species according to which component they belonged to, essentially creating a higher-level language on top of the existing CRN language. This had the advantage that we could focus on how the components connected and interacted rather than the components themselves. We discuss further development of this within the evaluation.

Another problem was with reaction rates. We often had to slow down the rates considerably in order to accurately analyse key moments in the development of a system. This was a trial an error process and so could take a very long time. Our reaction rates were usually proportional to the number of molecules within the system. The precise rates were irrelevant as, they only needed to be uniform.

# Chapter 3

# Component Design and Verification

*Note: For each component as many as 20 different experiments were run in order to analyse that a component behaves as expected. Therefore most experiments run are simply omitted from the following two chapters. We provide insight into the most interesting experiment run for each component as this will hopefully be enough to convince the reader that the components exhibit their expected behaviour. The verification process used was much the same as has been described in detail for the AND-gate in the previous chapter.*

This chapter offers a foundation for asynchronous circuit design which the reader can use to create more complex systems built from CRNs. The components to be discussed are:

- rendez-vous elements: latches, memory, c-elements and merges
- arbiter elements: forks, amps and arbiters
- conditional flow elements: FOR, IF and WHILE
- logical elements: logic gates and number representation.

The design of the above asynchronous components as CRNs is novel. In order for the reader to understand the notation used within this chapter it is first advised that the reader reads Section 1.1.5 first as it may be hard to understand the key concepts behind the design of these components.

Unless stated otherwise we use low molecular counts to test our components. Usually the maximum, representing an output signal of 1, is between 5 to 10 molecules. The words species and signal are interchangeable in this chapter as the presence of one

molecule or species usually is a catalyst or 'signal' to some other reaction. Because of low molecular count we do not assume an isochronous-fork as our well-mixed assumption no longer holds. An isochronous-fork is described in 1.4.4. We provide a new design for a fork later in the chapter.

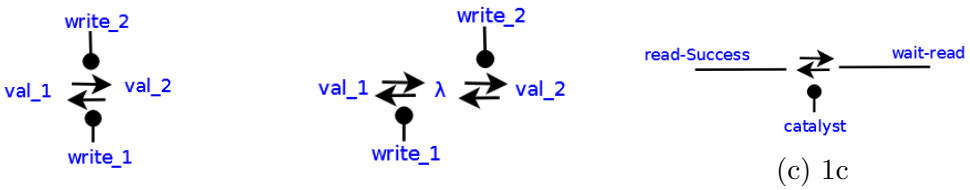
For each component an example experiment coupled with a simulation is given to convince the reader that each component has been thoroughly tested.

## 3.1 Rendez-Vous Elements

Because of a lack of clock in asynchronous systems, information flow relies on a series of 'handshaking' elements to make sure signals are synchronised. A rendez-vous element allows flow to synchronise by a series of 'wait' operations [58]. These elements are usually in the form of three components: latches, Muller C-elements and Controlled Merges [58]. Latches can control when data is cascaded or retained and play a crucial role in the construction of memory units. C-elements are used to create a ripple-carry affect which is used in the queueing of operations. Merges allow for a 'wait' operation in order for computation to resume. These components are particularly important in CRNs with low molecular account as 'obvious' forks and merges break our well-mixed assumption as described in 1.4.4.

### 3.1.1 Atomic Operations

At its most fundamental, in order to perform any computation we must define the foundational operations read and write. Figure 3.1 reveals the CRNs analogies to the read, write and controlled-write operations.




(a) 1a (b) 1b (c) 1c

Figure 3.1: The Write, Controlled Write and Read CRNs from Left to Right

A read operation requires 3 species: the species that is to be read, the wait-read species and the read-success species. When the catalyst is present it can be read. We

increase the population of the wait-read species if we wish to read. If the catalyst is present we can convert the wait-read species to the read-success species, thus performing a successful read. A read operation is needed when information needs to be extracted from a system.

A write operation is similar but needs to be thought about in a different way. We take two species linked by some reversible reaction. If species $\text{write}_1$ is present it catalyses with our reversible reaction and converts $\text{val}_2$ species into $\text{val}_1$. If species $\text{write}_2$ was present it would reverse the reaction. Thus, if the signals $\text{write}_1$ or $\text{write}_2$ are present they are recorded. A write operation is needed when we wish to record information about our system.

A controlled write has an intermediary species which starts with a high molecular count. In this way we avoid the problem that, in the case we wish to read the value recorded before any value is written, one species of the values that are to be read, is not initially present.

In Figure 3.2 we see a simulation of a process which first uses our write operation and then our read operation. In our CRN we have the species: xhi, xlo, mhi, mlo, creadmhi, creadmlo, readmhi and readmlo. xhi and xlo represent the signal carriers, mhi and mlo are the values which xhi and xlo write to, creadmhi is the control for the reading of the value mhi, which if present activates the readmhi wire. The value xhi is written to m1. After this, we read this value. So in the initial configuration we set xhi, mlo and readmone to 5 molecules and the rest to 0. As we observe the expectation of the readhi species is at 5 molecules, therefore the initial xhi species was recorded and read successfully.
This is fine when examining the case where high and low signals are strictly separated, but the reader might be asking the question, as was asked in Chapter 2, of what happens when both write signals are present at the same time. The outcome of this is shown in Figure 3.3.
We have probabilistic convergence on a centre point. We therefore have to be sure to safeguard and test against this in future simulations as there is a likelihood of this occurring in systems without correct queue structures in place . We explore this problem in much greater detail in the next chapter.

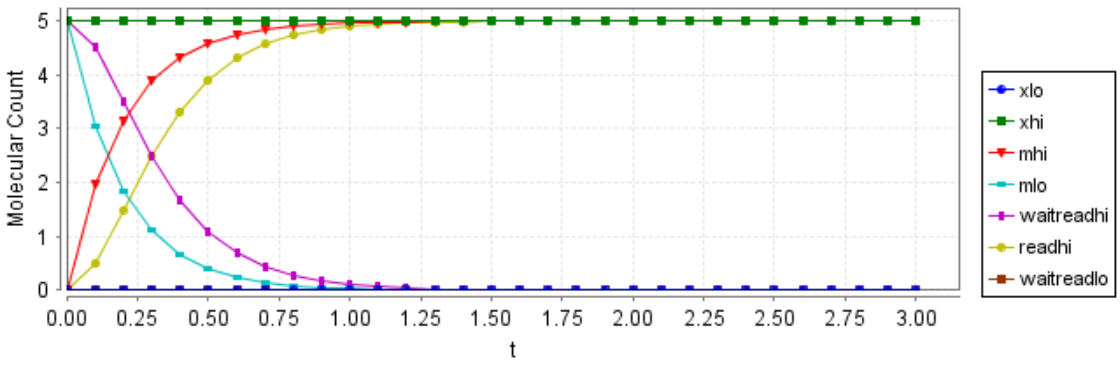



Figure 3.2: An experiment showing the recording of the species xhi to mhi and then the reading of mhi expressed by the conversion of waitreadhi to readhi

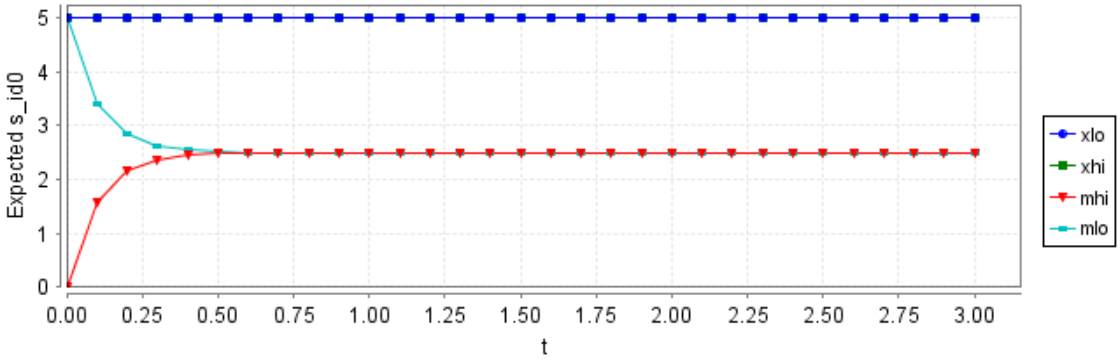
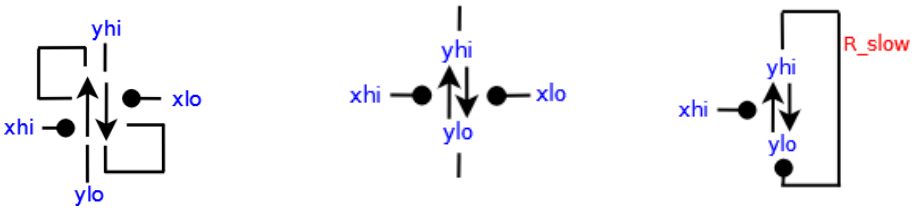



Figure 3.3: A similar experiment where xhi and xlo try to both register their presence to the reaction yhi $\leftrightarrows$ ylo which results in a convergence in values of yhi, and ylo

### 3.1.2 The Retention of Information

From this point on it becomes apparent as to why we use a diagrammatic language. We are now considering complicated CRNs with between 6 to 38 species and so to write-out every CRN would be confusing and impractical.

The ability to retain information is a crucial step in the construction of latches, c-elements and controlled merges. After all, a latch or flip-flop is a circuit that has two stable states and can be used to store one bit of information. It is a fundamental component of any computation that requires memory. We wish to create a CRN that can store a value despite changing variables. The most obvious way to implement a storage of memory is to create a feedback loop, which is demonstrated in Figure 3.4a. The problem with an arbitrary feedback loop is that it violates one of our main restrictions outlined in Chapter 1.




(a) 1a (b) 1b (c) 1c

Figure 3.4: Plots of prototype latches. Unfortunately, due to restrictions placed upon our CRN model these cannot be realised.

Mainly, that the reaction

$$\text{zhi} \rightarrow \text{zhi} + \text{yhi} \tag{3.1}$$

defies conservation laws. This is shown clearly in the simulation plot in Figure 3.5. In this plot ylo replicates itself producing double its original amount. So instead we have to find some way of storing information without violating conservation laws.

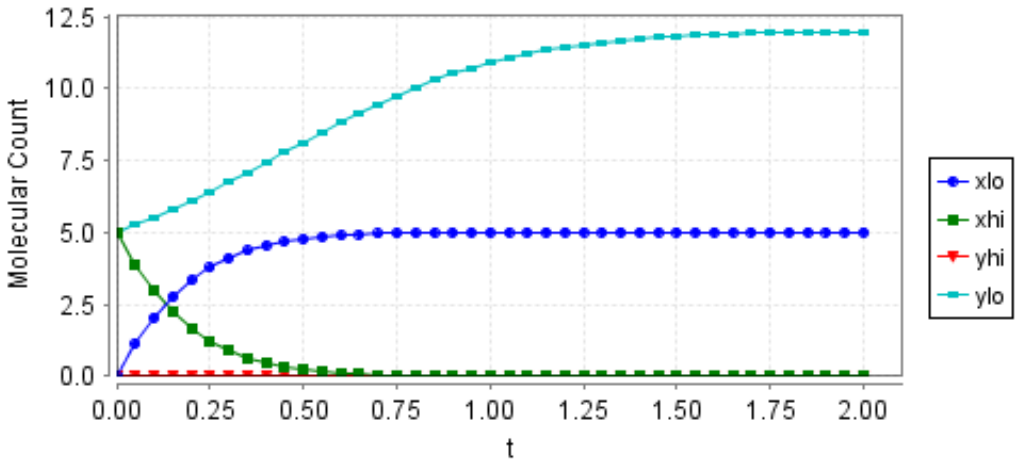



Figure 3.5: An experiment showing the violation of conservation laws from a memory component. ylo and yhi replicate a potentially infinite number of themselves

If we consider the diagram in Figure 3.4b then we have another problem. Assuming yhi and ylo are used as our carry signals then they may change, or be destroyed when the component is connected in parallel, so in affect they are only temporarily useful. In our simulation in Figure 3.6 we see the value of ylo trickles away to zero, in that we use the reaction ylo$\rightarrow$ {} to simulate ylo being used up in a reaction elsewhere.

So we need to find some way to be able to reuse our result.

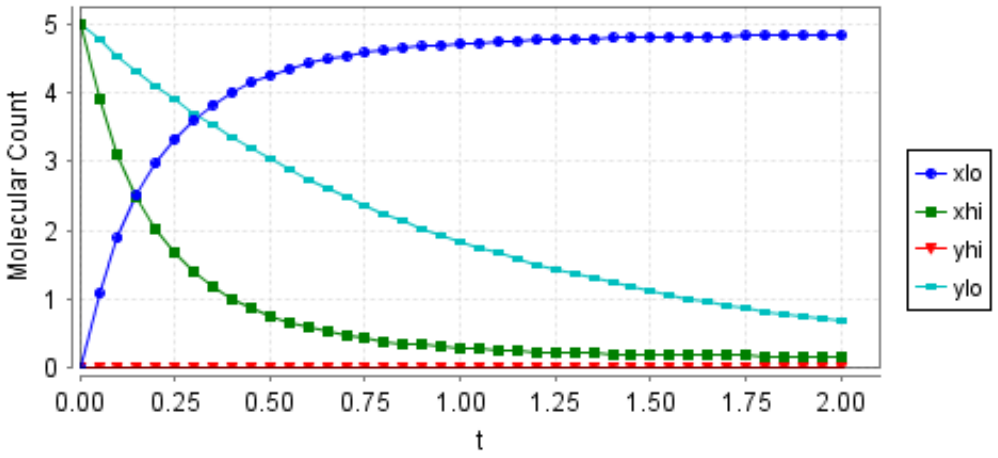



Figure 3.6: An experiment showing that the species ylo, when used up elsewhere, cannot replicate itself and therefore is useless if shared with another component. We simulate this 'other' component with the destructive reaction ylo $\rightarrow \{\}$

We could create a slow moving feedback loop as seen in Figure 3.4c, which gradually resets the latch when a signal is no longer present. However this violates our rate assumption that rates remain uniform across a CRN. We show a simulation of this exact latch in Figure 3.7; it works in isolation but is essentially useless when used in conjunction with other control flow elements.

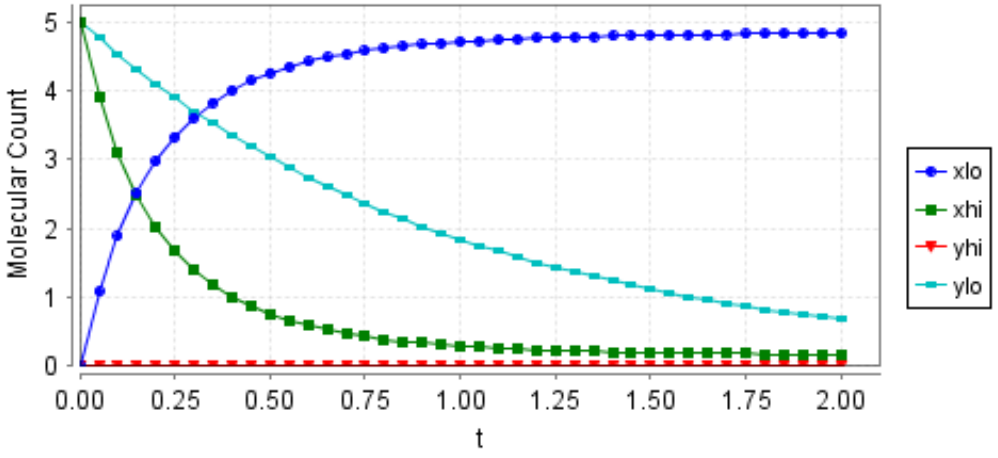



Figure 3.7: An experiment showing that the species yhi is slowly reset back to ylo. This particular component violates conservation laws.

### 3.1.3 Latches

Instead, for the design of a latch, we should consider an element which does not break any of the restrictions imposed on our CRN and instead focuses on the retention of information through the use of catalytic reactions. We consider two 'concepts' and build gates around them. In the simple latch design seen on the left of Figure 3.8 we have two inputs 1 and 0. These are stored in the reaction $1 \leftrightarrows 0$. These catalyse the read line which is the same as was explained in the read operation earlier in the chapter. The more complex gate design is Figure 3.8 (right) comprises two further reactions. These reactions have the effect of speeding up the reaction if the latch flips from one state to the other. This is done by two additional catalytic reactions relying on the presence of one input signal already being present.

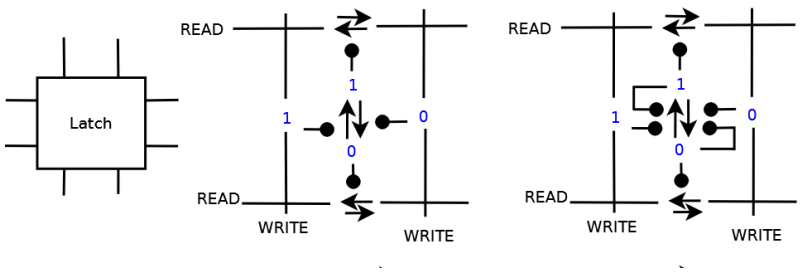



Figure 3.8: Two latch designs. The first being simpler but slower, the second faster but requires two more reactions.

We tested this in a similar experiment to our original read and write operations. First the value '1' is committed to the latch and then read. The simulation for this is provided in Figure 3.9. In this simulation the species writeOne is catalyst to the reaction mZero $\leftrightarrows$ mOne. The species waitReadOne, waitReadZero are signals to test if the species mOne, mZero are present. If they are, then these signals are converted into readOne and readZero. The experiment shows that the value is successfully committed and read. The more complex design produces similar, but faster results.

We can also build a slightly more advanced latch with a reset wire, based on our controlled write mechanism. Note that between any given reaction we can have some intermediary $\lambda$ in a reaction $m_1 \leftrightarrows \lambda \leftrightarrows m_2$. This intermediary can be a reset or dormant state. Our two latch designs are given in Figure 3.10 . When the reset species is introduced it effectively pulls the species $m_1, m_2$ towards a central $\lambda$.

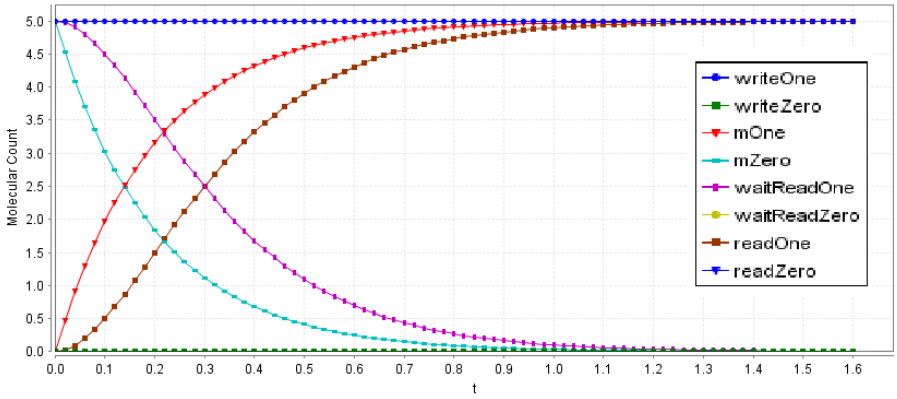



Figure 3.9: A simulation of the more complex gate design. In this simulation a value is first committed to the latch before being read. Once read the value is preserved.

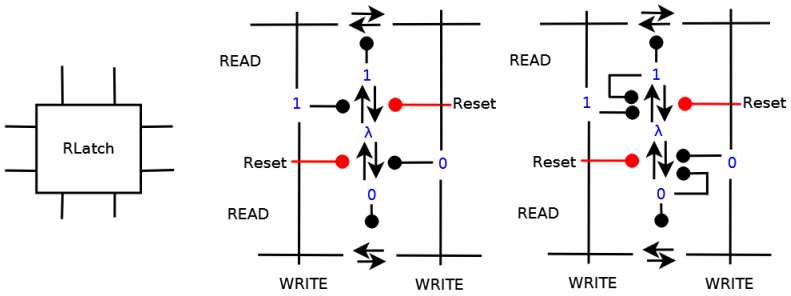



Figure 3.10: A reset latch. Notice how the reset acts as a catalyst which returns the latch to a central 'intermediary' state $\lambda$

The experiments designed to test these latches are more complicated. A new reaction writeOne $\rightarrow$ rOne is added to simulate a signal change during the experiment. rOne and rZero represent the reset catalysts. In Figure 3.11 we show a simulation which is setup as previous; however, the value of writeOne changes from high to low and the value of resetOne changes from low to high. Notice how the value of mOne and readOne respond to this. When the stimulus of writeOne is removed the latch resets, meaning that no value can now be read from it. We provide the CRN used in Appendix A.2.1; entitled 'resetLatchExperiment'.

### 3.1.4 Evaluation of Latch Designs

The simpler latch has the advantage over the complex latch that it requires two fewer catalytic reactions to function than its counterpart. However, the more complex latch is faster as it amplifies the effect of dragging to a state 0 or 1. Because speed is not something that we focus on primarily, the simpler match may have more use when it

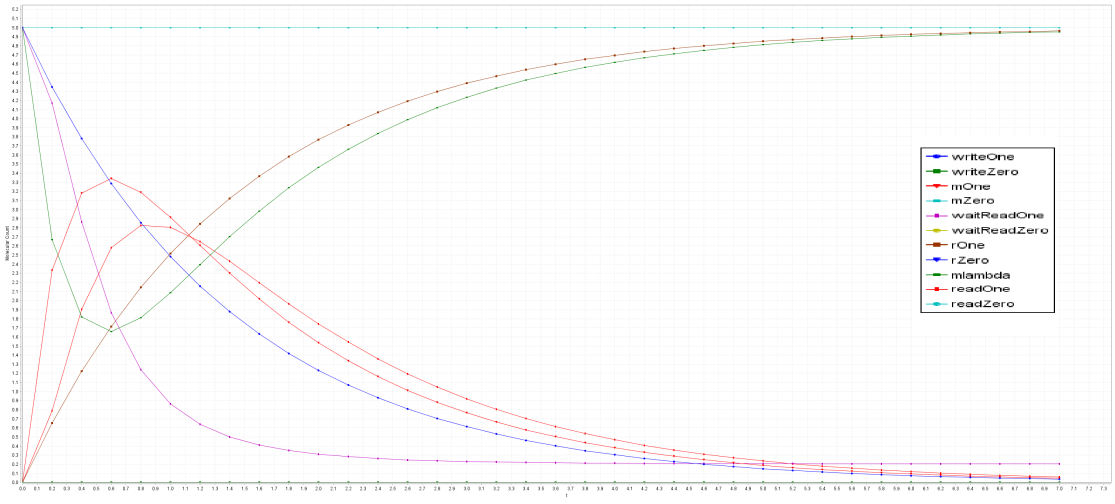



Figure 3.11: A more complex experiment for the complex reset latch. Notice how, as the input is transformed into a reset signal, the latch no longer exhibits a high or low signal and thus no value can be read.

comes to implementation of a CRN.

The tri-state reset latch has the major advantage over the bi-state latch that a 0 or 1 signal does not need to be present and therefore will not always be read. In any real-life counterpart a system usually has a rail deciding whether the component is active or inactive and so our intermediary $\lambda$ fulfils this function. This has the downside that a tri-stable state species need to be present in order for a reset-latch to become viable. Another major disadvantage of the reset latch is, when we want to consider larger blocks of latches or even a memory unit it can be considerably more expensive in terms of numbers of reactions. However, despite this disadvantage, we use the reset latch for the remainder of this thesis.

The CRN implementation of these latches can work in parallel with carrier signals such that we can have a battery of latches. For each unit we need 2 (or 3 in the case of the reset latch) species; however, these can be the same for every unit. For each unit we need two unique input signals and output signals (if every latch is to be addressed uniquely) so in a battery of latches we would need $4n + 2$ or $4n + 3$ species where $n$ is the number of uniquely addressable latches. For pipelining no such issues exist, as each latch does not need to be uniquely addressable.

### 3.1.5 Muller-C Element

As stated in Chapter 1, there exist implementations for the Approximate Majority algorithm in real world biological systems and DNA strand displacement. We represent Approximate Majority (AM) by the CRN shown in Figure 3.12. If a population has more members than the competing population then it has a positive effect, essentially dragging the already greater population to a maximum. The approximate majority is much like our tri-stable reset latch except for the fact that we can retain a state even when an input changes.

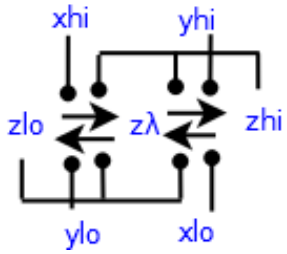



Figure 3.12: A CRN representation of the approximate majority algorithm.

This is much like a C-element. A Muller C-element is a gate with "memory" which holds a state. Although this was discussed in Chapter 1, a reminder of the truth table for this is shown below.

| Rail #1 | Rail #2 | Result |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | t - 1 |
| 0 | 1 | t - 1 |
| 1 | 1 | 1 |

When both inputs are 0 the output is set to 0, when both inputs are 1 the output is set to 1. For other combinations the output remains unchanged. This has the effect that, if an observer sees a change, this indicates that in order for this change to have happened both inputs must have changed. As we discuss in the next chapter most handshaking protocols rely on this cyclical transition.

We define the C-element as a CRN in Figure 3.13. The reason behind the additional states xlo, xhi are that it essentially drags the non-binary output of the approximate majority unit to a certain high or low signal.

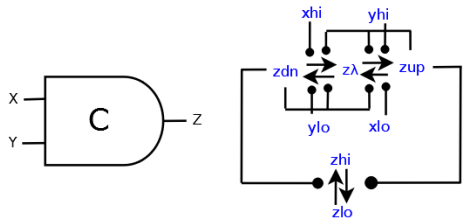



Figure 3.13: A CRN representation of a Muller-C element.

In the first experiment shown in Figure 3.14 the C-element responds to an input of two low signals. These signals are represented in the input species xlo and ylo. The central concentration of the AM element is dragged towards zdn and in turn the element outputs zlo as the dominant species.

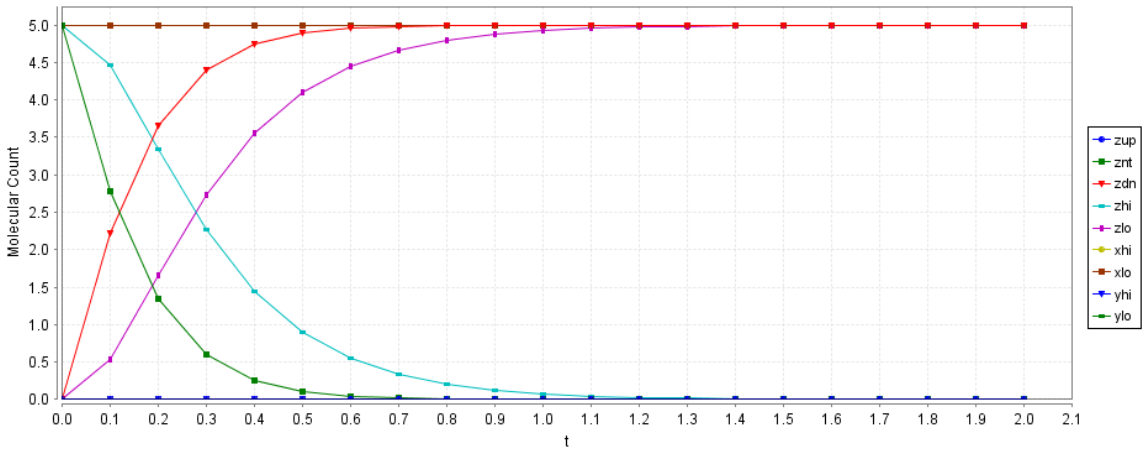



Figure 3.14: An experiment on our Muller C-Element testing the capabilities of the Approximate Majority Algorithm. The C-Element outputs a low signal with two low inputs.

In the second experiment, seen in Figure 3.15, we introduce a new reaction xhi $\rightarrow$ xlo on an initial input of xhi, yhi. In this case our element should still output a high as output despite the $x$ input change. We see that again the output signal zhi remains high.

In the third experiment, in Figure 3.16, we introduce a new reaction which changes both inputs from high to low. The C-element responds to this change and outputs a high signal.

From these three experiments we can see that the element exhibits the exact behaviour that we desire. The element remains stable in a situation where inputs change and the output is correct according to the logic table listed above.

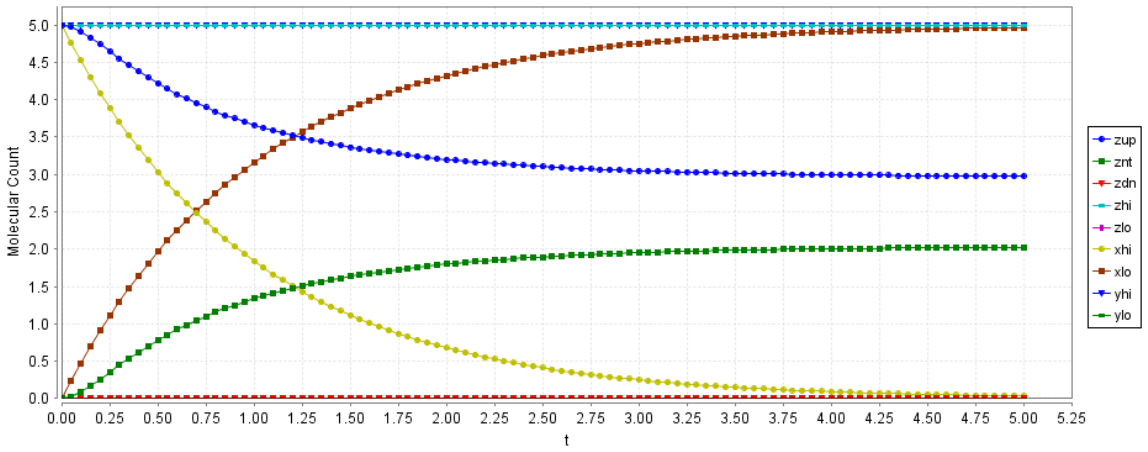



Figure 3.15: Again we test the C-element. This time we introduce a new reaction xlo $\rightarrow$ xhi which should have no overall effect on the output. This is indeed the case as our output remains zlo.

### 3.1.6 Merging

In most asynchronous circuits there is at least some form of branching and merging, be it for some condition or for parallel computation. We define a merge as a component where two or more signals need to be present in order for this component to output a signal. In essence one species must wait for another in order for both signals to have registered. The merge, as a CRN, is defined in Figure 3.17. Notice how the output signal requires two reactions to occur. If the bottom signal, $x_1$, is present this does not guarantee that the computation can continue.

In order to test the merge component we develop an experiment, seen in Figure 3.18 which combines a fork action with a merge. The fork is explained in more detail in the next section. We have two rates which govern two paths: pathOne and pathTwo. We can see from Figure 3.19 that xfinish, the output signal, does not respond until the two stimulus pathOne and pathTwo are both present. This is seen in the form of the slight delay in uptake seen between t= 0.0 and 0.5.

The reader might be unimpressed with the lack of information this merge outputs. We can use the latch designs, combined with logical elements discussed in a later section, to decide how these signals should be interpreted. The merge is purely used as a rendez-vous / control flow element in these circumstances.

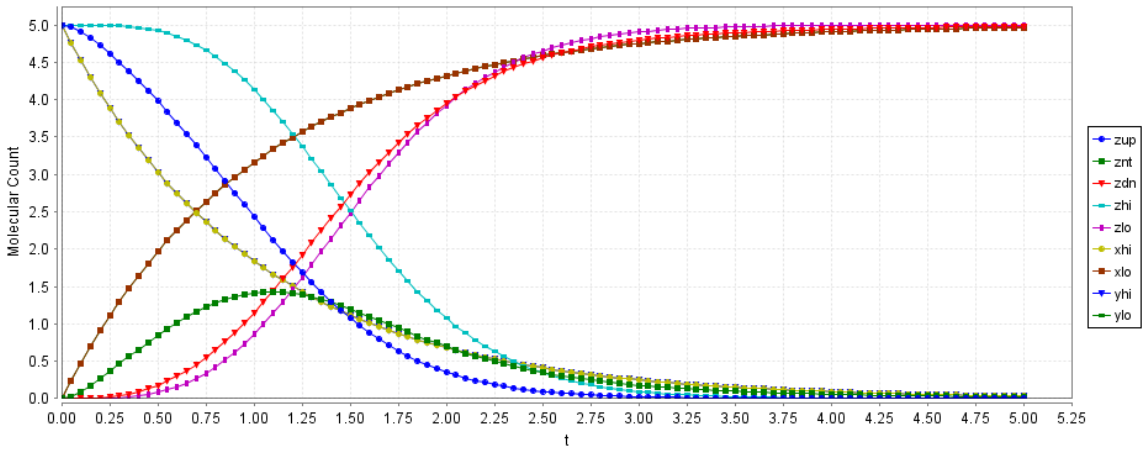



Figure 3.16: The C-element responds positively when both input signals are changed. The species zhi is converted into zlo.

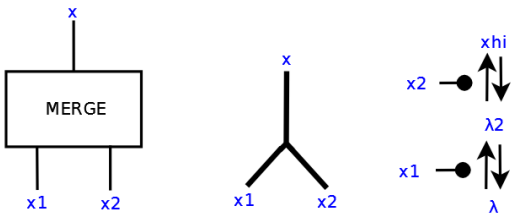



Figure 3.17: A definition of our merge element. The central representation is just a diagrammatic simplification. The chemical $x$ is only present when two input species $x_1$ and $x_2$ are present

## 3.2 Arbiter Elements

An arbiter element 'breaks ties'. We discuss three such elements within this section: the fork, the amp and biased arbiters. Arbiters are used in asynchronous systems so that a system doesn't halt and that it can eventually continue.

### 3.2.1 Forks

A fork is essentially the negation of a merge. The most obvious way to implement a fork would be to simply divide a set of molecules into two through two reactions of equal rate. mainly:

$$\begin{aligned} x &\rightarrow x_1 \\ x &\rightarrow x_2 \end{aligned} \tag{3.2}$$

However, because our model behaves stochastically it would be impossible to guarantee with acceptable probability that $x_1$ would decompose into two equal sums of

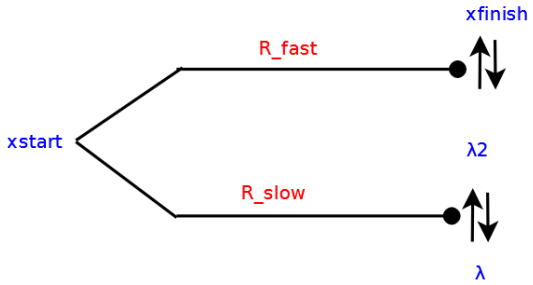



Figure 3.18: Our merge element was tested with an experiment which first forks and then merges. Two rates govern the seperate paths to mimic differences in computation. If both paths eventually terminate this activates the species xfinish.

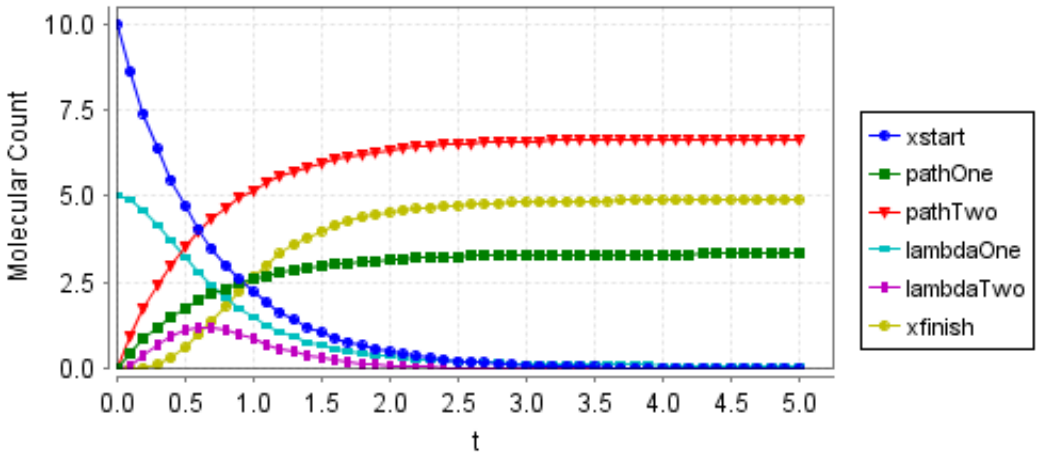



Figure 3.19: The output of the merge experiment. Notice how the output xfinish (in yellow) does not react until both signals, pathOne and pathTwo, are present.

$x_2$ and $x_3$. We therefore implement a fork using catalytic reactions which is seen in Figure 3.20. The fork itself acts as a catalyst to the two reactions, which in turn increases the speed and therefore the probability that both species will be activated at the same time.

We experimented with a fork in the last section which was actually an implementation of the newly defined fork. In Figure 3.21 we can see an implementation of this same fork, but in isolation. Unfortunately, the two signals $x_1$ and $x_2$ have exactly the same expectation and therefore rise at the same time, obscuring the reader from seeing the value for $x_2$. However, this graph does signify that $x_1$ and $x_2$ have the same expectation and therefore operate as we expected.

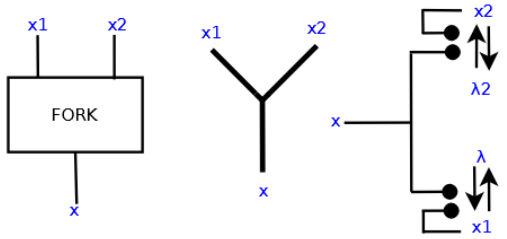



Figure 3.20: The fork splits a signal into two equal measures in approximately the same time step. We use four catalytic reactions to achieve this.

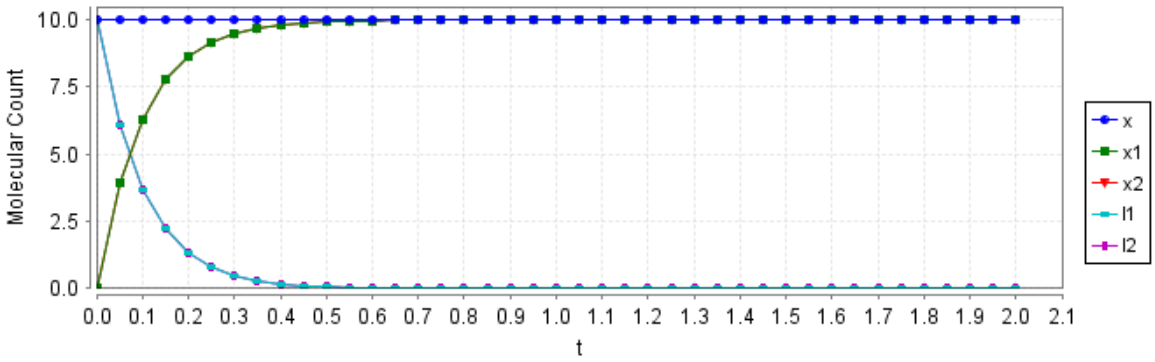



Figure 3.21: A graph showing the expectation over time for the molecular count of our fork. Fortunately, the signals $x_1$, $x_2$ have the same reaction time. Unfortunately for the reader, because of this, the signal $x_2$ is covered by the signal $x_1$ in this graph.

### 3.2.2 Amplification and Muting

The function of the amp is to amplify the dominant signal in two competing species. In our example in Figure 3.22, 0.7 is the larger number and forces the system into the state ylo. In Figure 3.23 we see a simulation of for this.

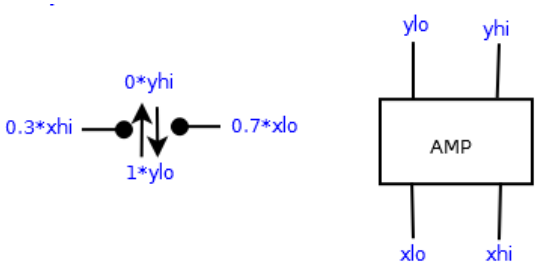



Figure 3.22: The function of the amp is to amplify the dominant signal; in this case xlo is the dominant signal.

It is important to realise that, in a catalytic reaction, the witness is not consumed and therefore can have arbitrary molecular count. In Figure 3.23 we see that a simulation

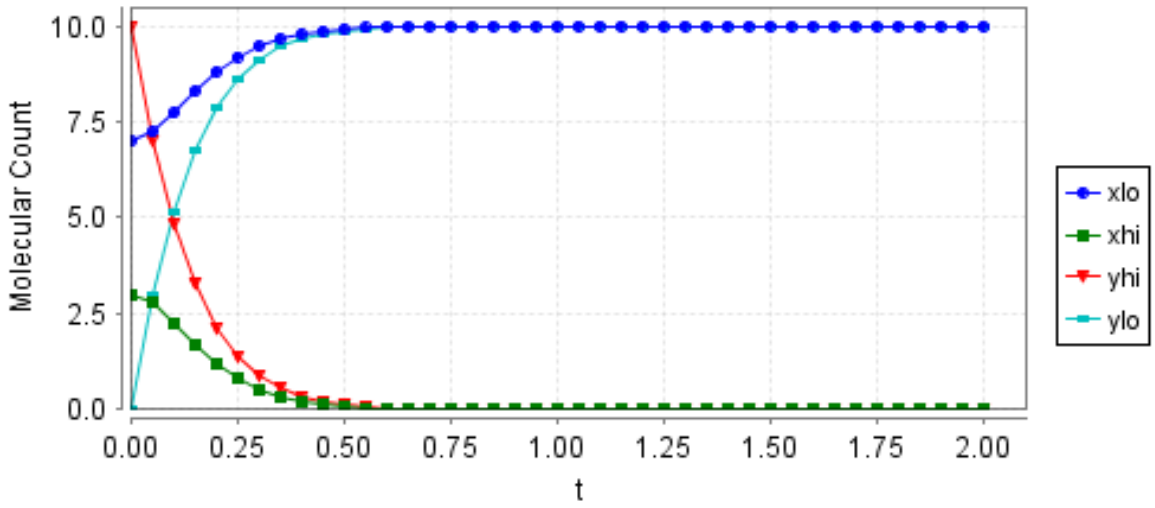



Figure 3.23: xlo is the dominant species in this experiment and so ylo becomes the only output signal that is present.

of the amp defined in Figure 3.22 shows that the smaller species, xhi and xlo, can influence the larger molecular count of the reaction yhi $\leftrightarrows$ ylo. Similarly, in a process called 'muting' larger amounts of molecules can give rise to smaller reactions. This becomes useful when we discuss control flow elements later in the chapter as we can limit the number of molecules we have to simulate and verify.

### 3.2.3 Biased Arbiters

The main problem with the amp listed above is in the case when the input signals are equal 0.5, our CRN cannot deal with this and we get probabilistic convergence similar to that displayed in Figure 3.3. However we need some method to 'break ties' as this is the function of an arbiter. We do this by introducing a biased arbiter, the CRN for which is seen in Figure 3.24 . In this arbiter yhi is always favoured over ylo even when the inputs are even, hence breaking a tie. We simulate an example of this in Figure 3.25. In this example xhi and xlo have the same concentrations. However, despite this yhi has the dominant concentration revealing the arbiter's bias towards yhi. This example experiment is far from conclusive and we also tested for when one signal was more dominant than the other and vice-versa.

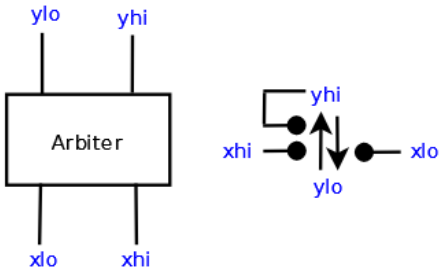



Figure 3.24: The arbiter we use here is biased towards the output signal 'high' in the form of yhi.

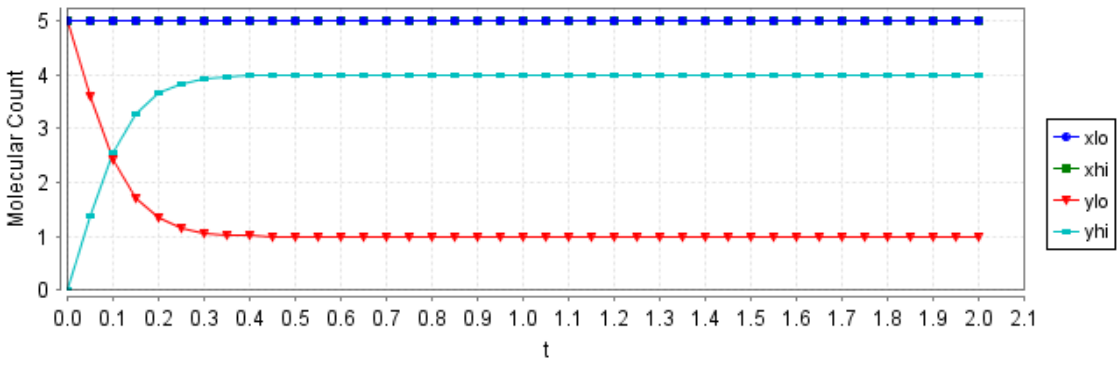



Figure 3.25: Both xhi and xlo are tied in concentration. Despite this, yhi is revealed to be the dominant species, showing the bias of the tie.

## 3.3 Control Flow Elements

As well as the introduction of *rendez-vous* and *arbiter* elements, we also discuss theoretical implementations for loops and decisions. A loop is where a computation or action is repeated over and over until some condition is met. We use a 'black box' element in this section 'LOOP LOGIC' which we deliberately obscure. This is because the loop mechanism can be constructed using pipelining ,which is discussed in detail in the next chapter. Unfortunately for now we do not yet have the required knowledge to understand the inner workings of LOOP LOGIC and so we treat it as a box which outputs a signal when an iteration is completed. The components to be discussed in this section are the IF, WHILE and FOR components.

### 3.3.1 Conditional Statements

A conditional statement is where, based on some condition a branch of computation is realised. We define an if statement in Figure 3.26. The if computation can be seen

as a reading of some condition followed by a branch. As we can see it is comparable to our read action. Our read action can be seen as the statement if: $m_1$ read $m_1$ else: wait.

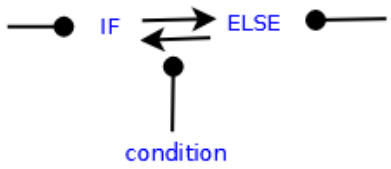



Figure 3.26: A conditional IF statement. If a condition is present we shift to a new branch of computation. It is comparable to our original read statement.

### 3.3.2 Loops

The while condition is similar. We can set up a condition gate as before, which can change over when a computation has been completed. The internals of the structure are deliberately left vague as this can vary for different while loops. The basic concept is that we iterate through a pipeline, described in detail in the next section, until some condition is met. We show our basic structure in Figure 3.27.

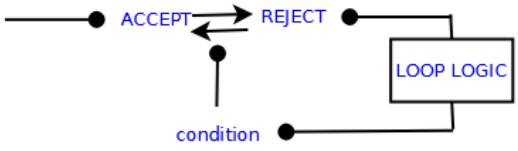



Figure 3.27: A CRN representation of a while loop. It continues to perform computation inside the loop body until the condition is accepted.

We also show an experiment taken from the next chapter, see Figure 3.28 . In this experiment a pipeline is used to convert the species testOne into testTwo. Eventually the accept condition is met and overtakes reject in molecular count. We are then free to continue on with whichever computation is being completed at the time.
The for statement is unfortunately very difficult to achieve. We need to define some concept of a number, which, as we discuss in section 3.4.3, is not easily done. We show an example of a for loop in Figure 3.29. In this example, at each iteration a species pass is converted twice until the presence of $pass_2$ can convert REJECT into ACCEPT.

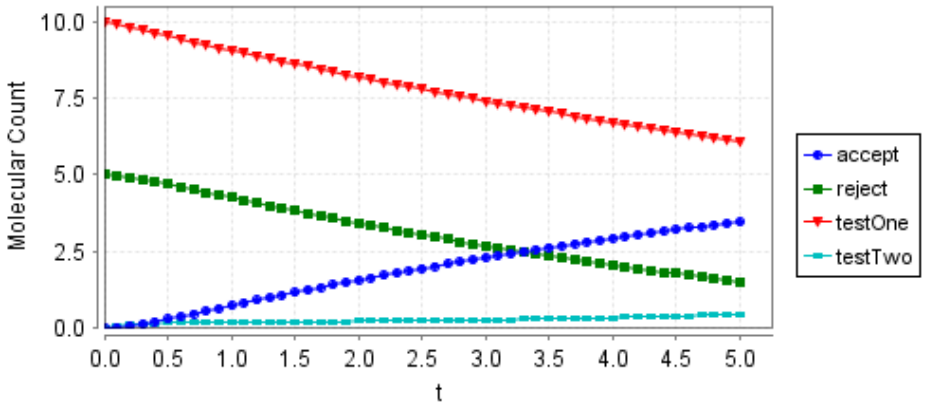



Figure 3.28: An experiment where a pipeline converts the species testOne into testTwo. Eventually enough is converted for the condition to be met and the while loop shifts to having been accepted.

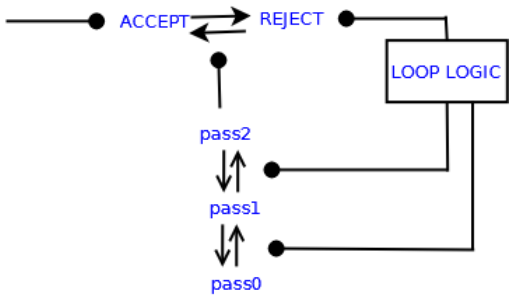



Figure 3.29: A CRN representation of a FOR loop. For each iteration we need to introduce a separate signal and a separate catalyst. This may not be viable if we are trying to limit the number of overall species needed within our system.

## 3.4 Logic and Numbers

In this section we discuss the use of logic and numbers within CRNs. Logic is used to make decisions and perform logical operations. Numbers are used to make arithmetic calculations. Unfortunately, numbers are not easily achievable in CRNs because of representation.

### 3.4.1 NOT, AND and OR

We first examine the most fundamental of logic gates. The logical NOT gate or 'inverter', as it is referred to in other papers, is used to convert a high signal into a low signal and vice-versa. As we can see in Figure 3.30 (left) the signal xlo catalyses the reaction ylo $\rightarrow$ yhi and xhi catalyses the reaction yhi $\rightarrow$ ylo. In affect we convert

a signal representing a logical 1 (xhi) into a logical 0 (ylo). This is a NOT gate as $\neg 0 \equiv 1$ and $\neg 1 \equiv 0$.

The middle diagram in Figure 3.30 should hopefully look familiar to readers, as this was the logical AND gate explained in Chapter 2. When signals xhi and yhi are high then we output the signal zhi. For any other combination of input signals we output a low signal. This is equivalent to an AND gate which will only output a logical 1 if both inputs are 1.

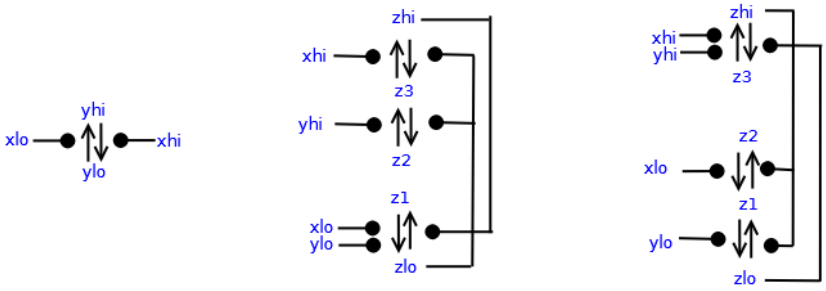



Figure 3.30: CRN representations of the logic gates: NOT, AND and OR from left to right. The logic gates are persistent, meaning that on a change of input they will reflect this change accordingly.

The final diagram in Figure 3.30 is a representation of an OR gate. In this diagram we see that only on an input of xlo,ylo would we have $z_2 \dashrightarrow zlo$. We show an experiment in Figure 3.31 which shows the signal zhi responding to an input of xlo, yhi meaning that that the input signal of logical 0,1 is still accepted.

### 3.4.2 NOR, NAND and XOR

The interesting thing about NOR and NAND is that usually they can be constructed with an AND or OR gate followed by a NOT gate. However, we can construct them by simply flipping the labels for our chemicals. In Figure 3.32 we see a NAND gate (left) and a NOR gate (right).

We show the reader an experiment on a NAND-gate where the inputs are both zero, seen in Figure 3.33. The gate responds by outputting a signal zhi.

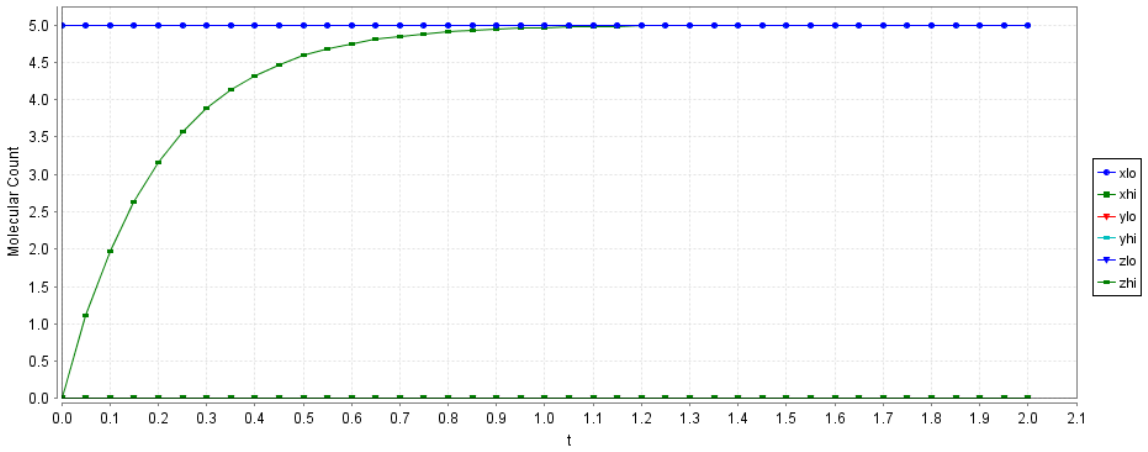



Figure 3.31: An experiment showing an OR gate on logical inputs 0,1. The signal zhi responds by a population increase resulting in a high signal.

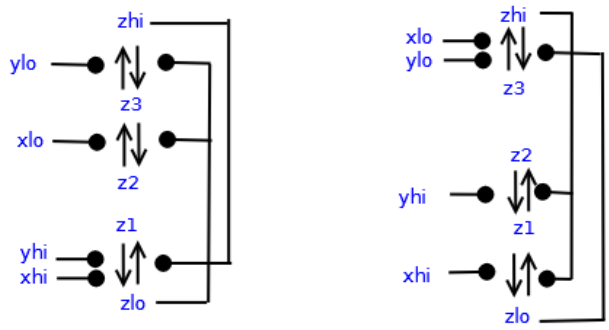



Figure 3.32: A CRN representation of the NAND and NOR gates. We achieve these by switching the labels of our AND and OR gates.

Perhaps the hardest logic gate to construct is the XOR gate. This gate needs a separate reaction for all four inputs, making it extremely inefficient to construct. A representation of this gate can be seen in Figure 3.34. The system is forced into a state zlo if the inputs are the same, and zhi if the inputs differ. We do not provide a plot for this due to the extensive use of the XOR gate in the adder, presented in the next section.

### 3.4.3 Numbers

Unfortunately numbers are very inefficient to implement. Any sort of efficient representation would have to be encoded into the label of the CRN which can, by the CRN framework, have no influence on the overall state of the system.

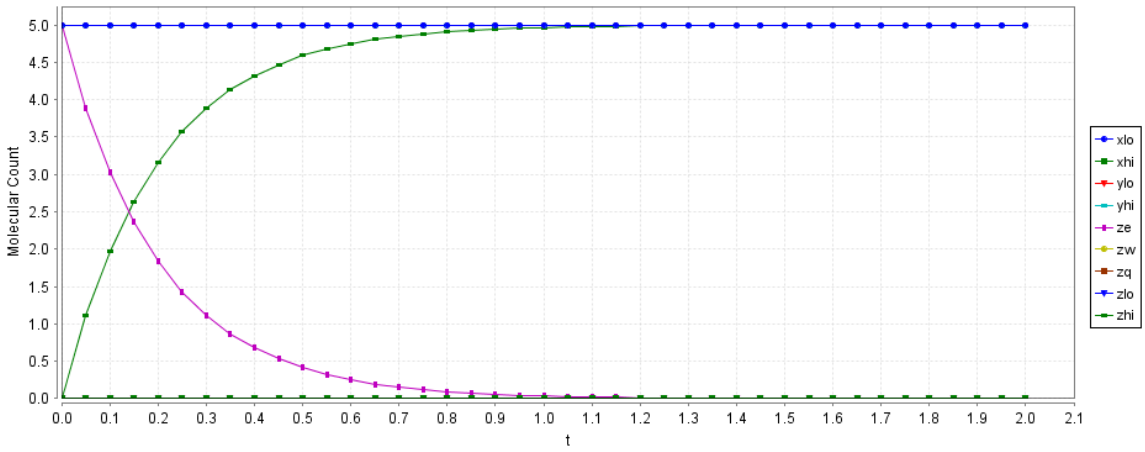



Figure 3.33: An experiment showing a NAND gate on the logical inputs 0,0. The gate outputs a signal of 1, signified by the output of the signal zhi.

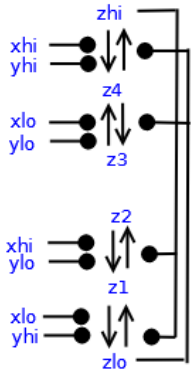



Figure 3.34: A CRN representation of a XOR gate. In this representation we need separate reactions for each input meaning the gate is considerably more complex than our other gates.

The best way to implement numbers is through a series of binary latches which can store and read numbers. In this way we can use pipelining to transfer numbers in parallel by a series of merges. This would unfortunately require an exponential number of latches in the size of the number we wish to store. Because of these conclusions we only deal with very small numbers in this thesis. We implement a 3-bit adder in the next chapter.

A circuit diagram for an adder is given in Figure 3.35. Because we have defined all of the atomic components, we now switch to a circuit based notation. This is because we only care about interactions between components from now on rather than the

components themselves. Each component is connected through a series of catalytic reactions.

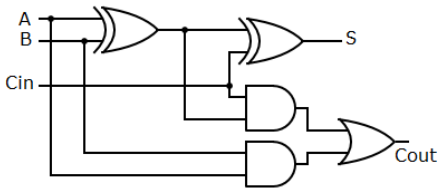



Figure 3.35: A circuit diagram for a full adder. Because we have defined all components as CRNs we resort to circuit notation. The implementation is still dual rail. This circuit takes three bits: A,B and a previous carry bit. It outputs the sum of A + B + *Cin* in S and *Cout*. *Cout* is high if there is an overflow.

The adder has 3 inputs. We show simulations on a CRN model which is given in Appendix A.2.3 in Figure 3.36 and Figure 3.37 on two standard inputs. This model is contains CRN components listed previous, connected through a series of catalytic reactions. This is logically equivalent to the circuit diagram shown in Figure 3.35.

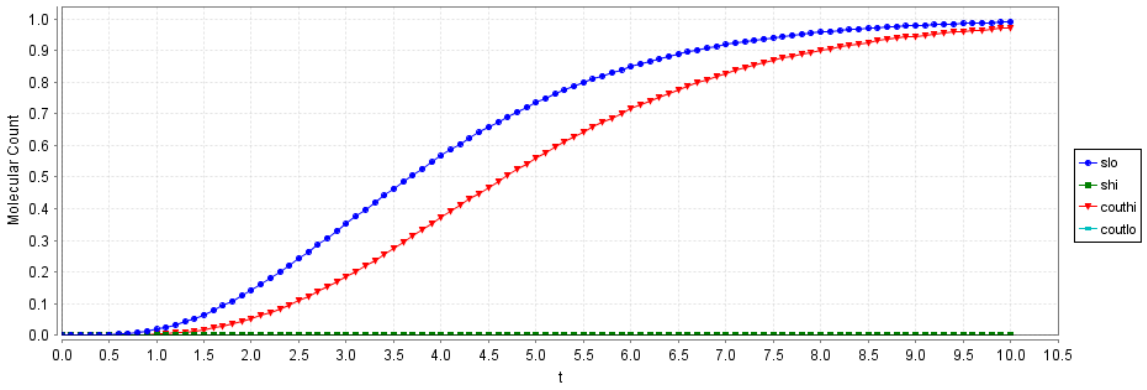



Figure 3.36: In this simulation we have inputs A = 1, B = 0, Cin = 1. Logically, we would expect the adder to output a 0 carry 1 as we wish to add two ones together (01 + 01 = 10). As we can see the signals slo and couthi are at high after 10 seconds. These are the species which represent S = 0 and Cout = 1 and so this component produces the right result given the input.

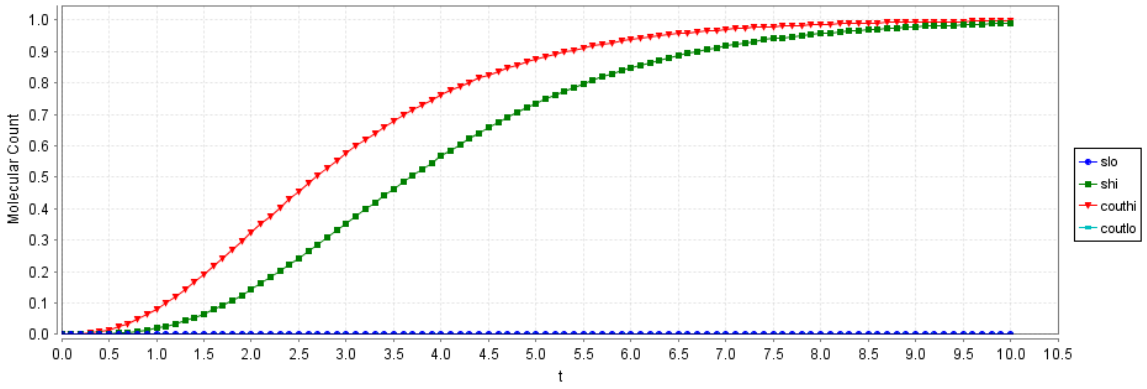



Figure 3.37: Similarly, in this simulation we start with the configuration A = 1, B = 1, Cin = 1. We would expect the output 1 carry 1 (01 + 01 + 01 = 11). Both species couthi and shi are present which is the correct response to this input. Confirming that our gates can be assembled in parralel to create more complex components.

# Chapter 4

# Systems, Pipelining and Algorithms

*As in Chapter 3, we keep results concise. We provide the most interesting experiments performed on each structure. Most of the CRNs used within this chapter can be found in the Appendix A.2. In section 1.4.3 of the literature review we cover the relevant material to understand how these pipelines operate.*

From the components we have just defined and verified we can build more complex systems. This chapter explores pipelining through a series of localised 'hand-shaking' protocols. Two protocols are rigorously tested, the 4-phase dual-rail protocol and the 2-phase dual-rail protocol, both of which are described in detail within the literature review. The use of pipelining in asynchronous systems is an alternative to a clock found in most synchronous systems.

This chapter explores the construction of the Muller pipeline and some of its applications in the form of a queue and a ripple-carry adder. The pipeline allows data to be transferred from one end of a line to another without loss or gain of information. The pipeline is the main novel result, central to the theory that underpins the thesis.

## 4.1 Muller Pipeline

This section explores the construction and verification of the Muller pipeline. The pipeline uses several components defined in the last chapter, mainly: the Muller-C element, the Fork and the inverter or logical NOT gate.

### 4.1.1 Construction

The Muller pipeline, seen in Figure 4.1 is a mechanism that relays handshakes. We first initialize all the components to their zero or neutral state. Following this we can begin cascading a 1 to the right. To understand how this works consider the middle element. It will propagate a 1 from its predecessor only if its successor is 0. In a similar way it will propagate (input and store) a 0 only if the previous input is one. It may be useful to think of this propagation sequence as a wave cycling from zero to one and back to zero. This could be compared to an asynchronous clock. In this way the C-element is used as a means to propagate waves, ensuring the integrity of data at any stage in the pipeline.

We construct our model pipeline by placing three of our C-element CRNs in parallel. At each intermediary stage between the C-elements we add a fork. One path of the fork is negated and fed back into the previous C-element. One path is fed into the new C-element. We include the full CRN for this model in Appendix A.2.4.

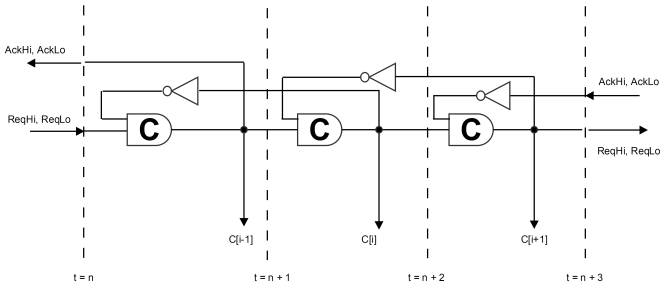



Figure 4.1: A circuit diagram for the Muller pipeline. We construct our CRN using this circuit diagram and our CRNs for each component defined in the last chapter. Note that the small black circles signify a fork.

We verify this pipeline using the same methodology as before. First we propagate a value of one along the pipeline, looking at each component individually and then we change the input values in a cyclical fashion. In Figure 4.2 we initialise the pipeline by setting the request line to high. The output species of each C-element are called ahi, alo, bhi, blo etc. Notice how each signal rises one after another, showing that indeed a value of one is being propagated along the pipeline.

We also experimented with removing the 'dragging' reactions from the C-element described in the previous chapter. The main problem with this, seen in Figure 4.3,

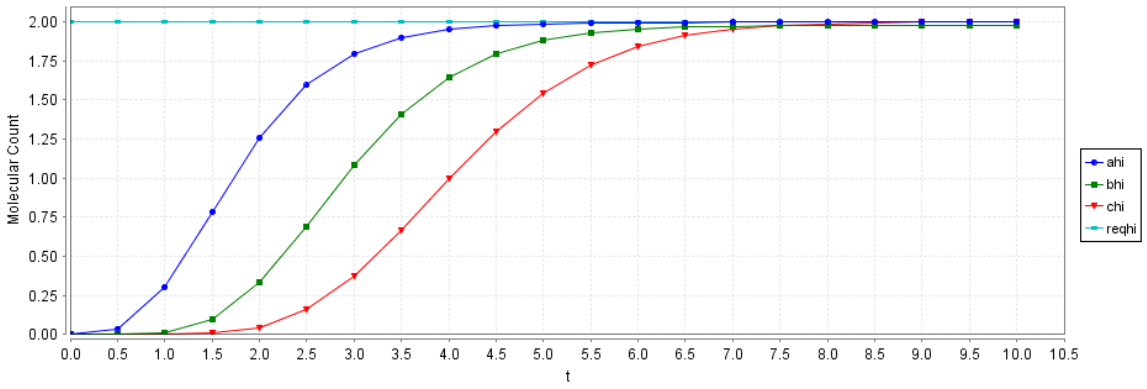



Figure 4.2: Initially we send a logical '1' down the pipeline and register the delay between each component.

was that it produced half-values along the pipeline which resulted in poor signal transfer from one C-element to the next.

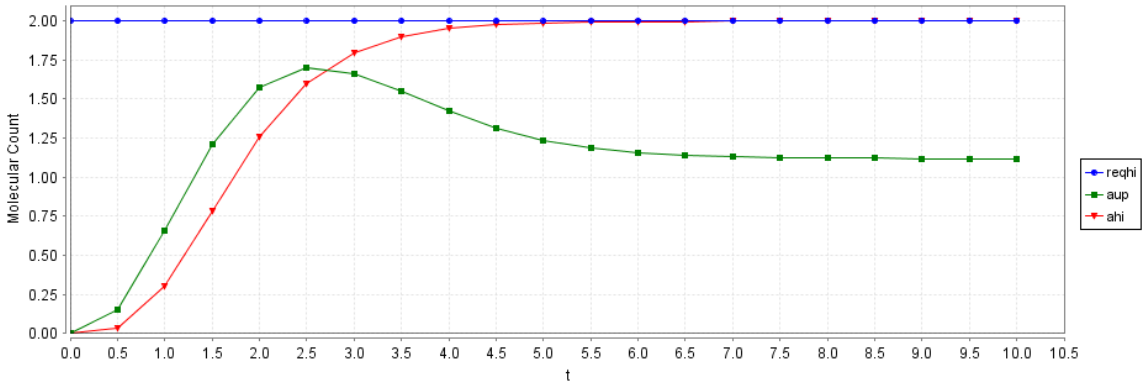



Figure 4.3: We show that when the 'dragging' effect is removed the species aup settles at some intermediary value, neither high nor low.

Once the signal has propagated from the C-element the signal forks and one path is inverted feeding back into previous C-elements. We show a simulation of this in Figure 4.4. One after another, the signal is inverted feeding back into the previous C-element.

Finally, we show a 'wave' through the pipeline propagating a high signal and then a low signal. The results of this are shown in Figure 4.5. Here the species ahi,bhi,chi exhibit a high signal before responding and diminishing back to zero. A keen reader may be wondering why the signal slightly diminishes along the pipeline, this is to do with timing issues which are more pronounced in Figure 4.5 in the section on loops.

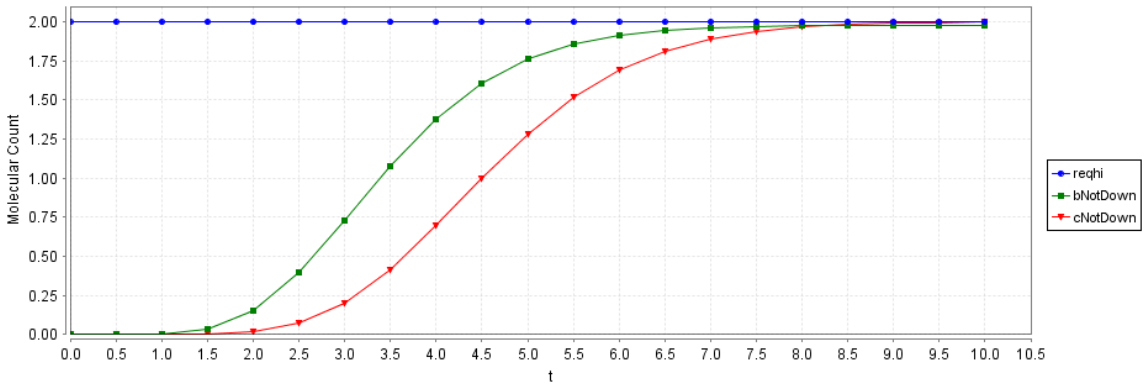



Figure 4.4: The NOT gates respond to the high signals propagated from the C-elements with the inverted signals low seen in the species bNotDown and cNotDown.

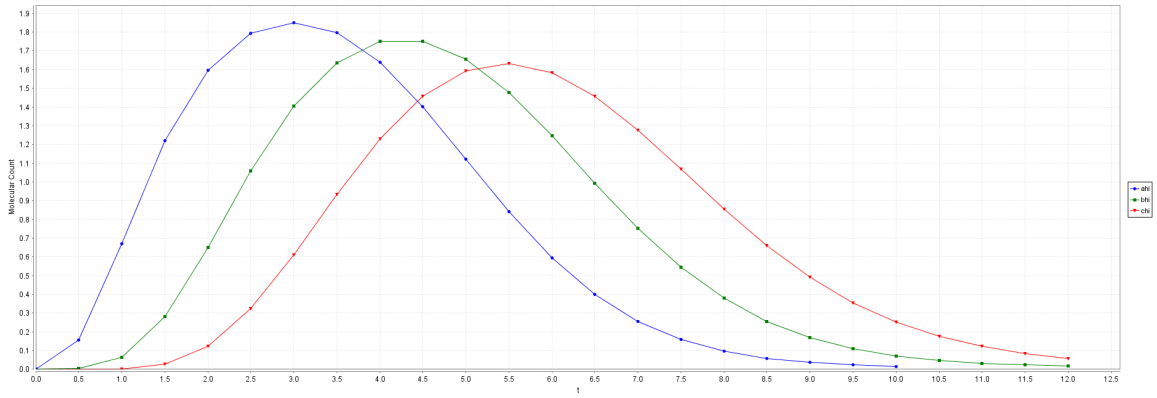


Figure 4.5: We 'pulse' a value of 1 along our Muller pipeline. Our pipeline responds with the sinusoidal pattern expected from cycling between 1 and 0.

### 4.1.2 The Queue

The queue is built as shown in Figure 4.6. We use our complex latch in order to build the queue structure. As a 1 is propagated along the pipeline, it sends a signal to the queue structure to read and store the value in the next latch along. Each latch represents some computation that could be completed at each time interval.

We experiment by sending a message of '101' propagated along the queue. We show a simulation of our queue in Figure 4.7. The signals outOne and outZero are accumulators which show indeed that two 1s and one 0 was sent along the queue during the simulation. We include the signals bZero and aOne to prove that signals are still propagating along the pipeline.

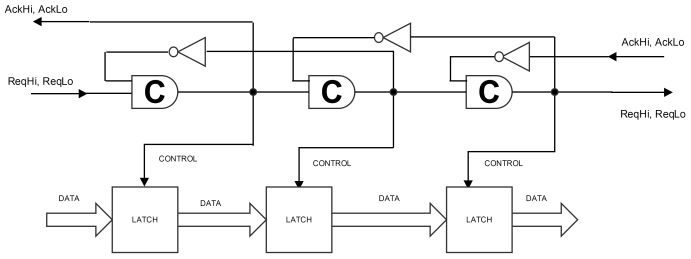



Figure 4.6: A schematic of our implemented queue.

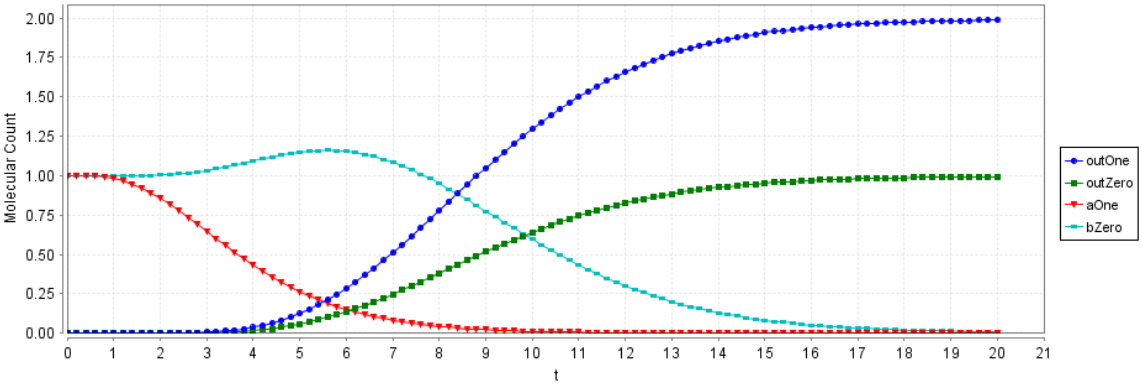



Figure 4.7: We send a signal of '101' along the queue which is accumulated in the signals outOne and outZero at the end of the simulation.

### 4.1.3 Ripple-Carry Adder

We constructed a 3 bit ripple carry adder, which works in a similar fashion to our queue. We show a circuit diagram for this in Figure 4.8. At each timestep we input two bits and a carry which outputs a the sum and a carry as described in the previous chapter. In this way we can add two three-bit numbers together.

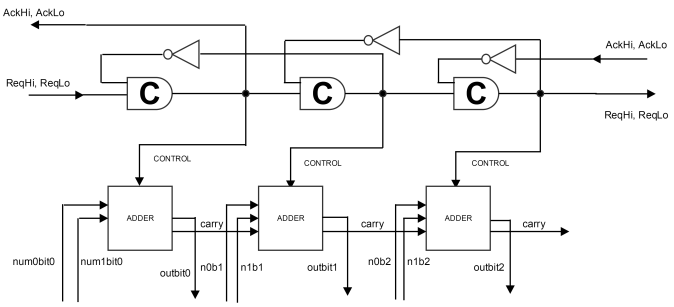



Figure 4.8: Circuit diagram for our 3-bit adder.

In the simulation shown in Figure 4.9 we add the values 010 and 111 and expect the

value 111. After the first stage we see that the first adder outputs a high signal (0 + 1 = 1 + 0c). After the second stage we see that the carry is high (1 + 1 + 1 = 1 + 1c). After the third stage we can see that the sum signal is low as ( 0 + 1 + 1c = 0 + 1c). The adder exhibits correct behaviour and each sum is not calculated until the next stage in the pipeline.

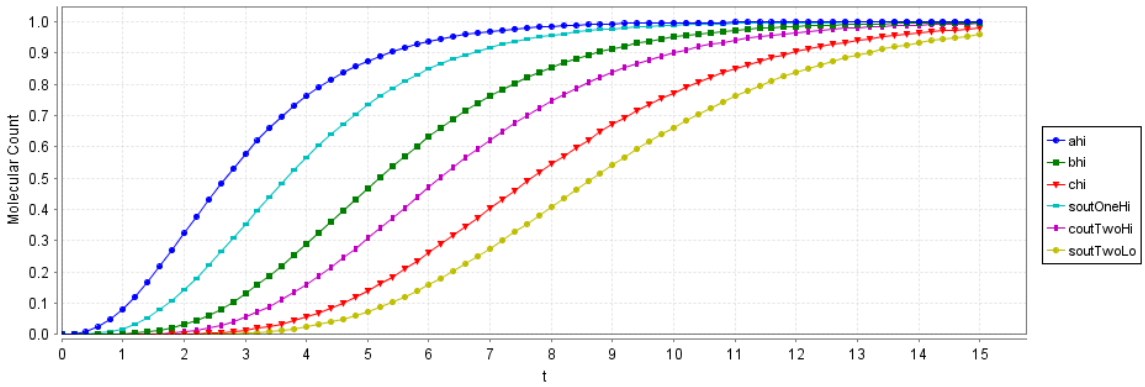



Figure 4.9: We add the values 110 and 111 together which produces the correct output. The outputs are demonstrated inbetween the signals of the pipeline showcasing the 'ripple' effect.

### 4.1.4 Loops

As discussed in the last chapter loops are a common part of asynchronous systems. Fortunately, the simulation of a loop is easily built upon our Muller pipeline. In Figure 4.10 we see a diagram representing a loop where the output of the pipeline is fed back into the beginning, creating a loop. We check if the condition is met before looping back.

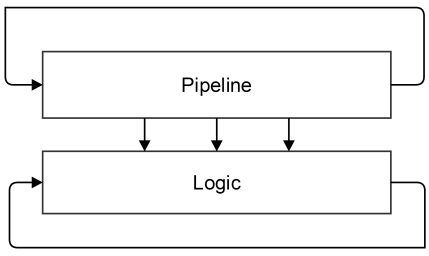



Figure 4.10: We can create loops by looping a muller pipeline back on itself creating a perpetual signal.

We show a simulation of a 'looping' pipeline in Figure 4.11. We have signal decay problems purely because of rate issues. With more time we could adjust the pipeline to slow down such that each signal would be allowed to fully rise to a maximum or minimum.

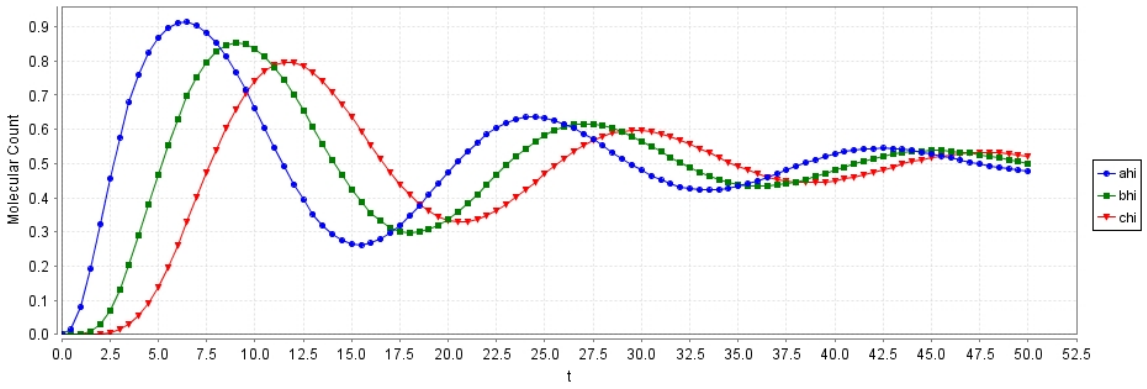



Figure 4.11: We show the pipeline oscillating by tracing the signals ahi, bhi and chi. We experience signal decay purely because the rate of chage is too fast to allow each signal to settle at its maximum or minimum.

## 4.2 Protocols

This section provides a discussion on two protocols, the 4-phase dual rail protocol and the 2-phase dual rail protocol, both of which are discussed in the literature review.

### 4.2.1 4-phase Dual Rail Protocol

A 4-phase protocol generally has 3-rails: request,acknowledge and data . These can be set to high or low. In this specific example of our 4-phase handshaking protocol we take into account two people: Alice and Bob. The four phase protocol is as follows: firstly Alice sends data and sets request to high. Bob then writes the data to the register and sets acknowledge to high. Alice responds by setting request to low and finally Bob acknowledges this by setting acknowledge to low.

The 4-phase dual rail protocol, like before, can be implemented via a Muller pipeline. In Figure 4.12 we see a diagram describing its construction. In our construction we developed a 3-latch fourphase dual rail protocol. In Figure 4.13 we see the output of this first-latch. The signals, *adfout* and *adtout* represent the output of the C-element. The *aorout* signal represents the signal after the OR-gate. We see *adfout* is low and

*adtout* is high representing a logical 1. The OR-gate responds by propagating a logical 1.

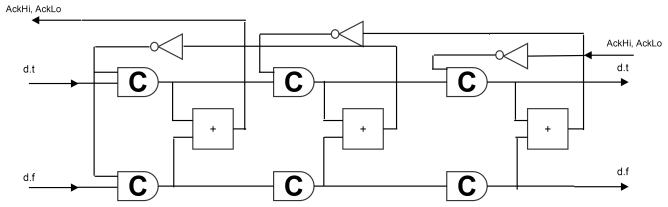



Figure 4.12: A circuit diagram describing the construction of a Muller pipeline.

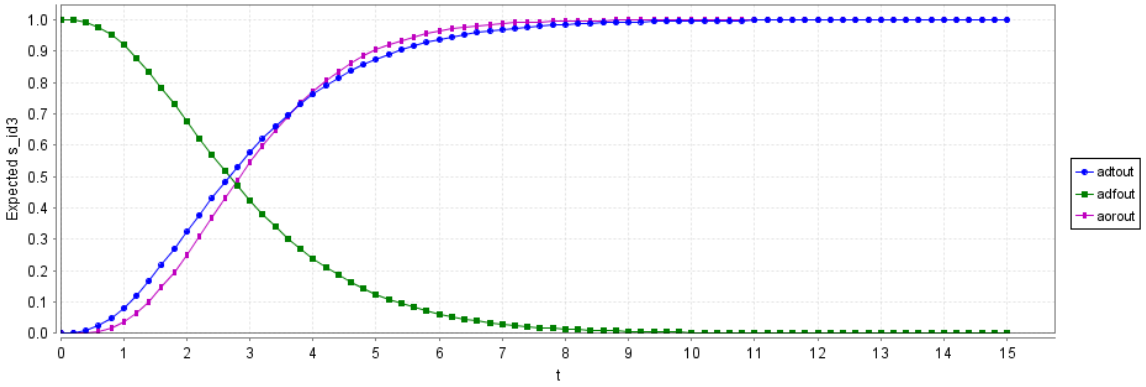



Figure 4.13: The first OR-gate in the construction of the 4-phase dual rail pipeline. The signal aifouthi is present suggesting the transmission of a codeword {0,1} or {1,0} has occurred.

As seen in the Muller pipeline we send a 'pulse' signal down our four-phase dual rail pipeline which can be seen in Figure 4.14. This 'pulse' represents the transfer of a logical one followed by a 'non-signal' to the end of the pipeline.

### 4.2.2 2-phase dual-rail Protocol

The transition from high to low is the same as the transition from low to high on each of these wires. We can see how this now leads to two events: (1) the sender sets data and request (2) the receiver stores the data and sets acknowledge to high.

The 2-phase dual rail is similar but the information is again encoded as transitions from $1 \rightarrow 0$ and $0 \rightarrow 1$. The main difference is that we acknowledge after one data codeword. The circuit diagram for this has been replaced with a single-rail pipeline

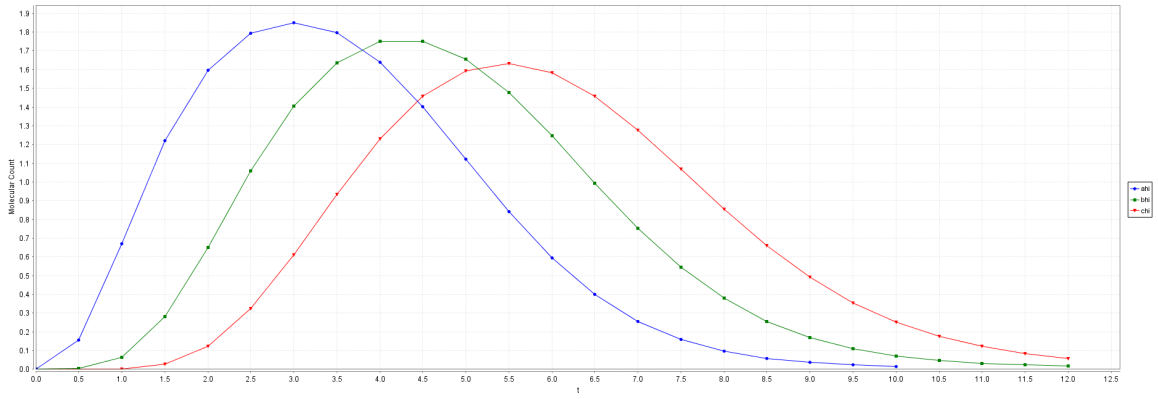



Figure 4.14: A 'pulse' of a logical 1 through our pipeline construction. The signals ahi, bhi, chi are all present suggesting that the protocol has successfully negotiated the transfer of a 1 followed by a 'no signal' to the end of the pipeline.

to describe to the reader how the implementation of two-phase differs from its four-phase counterpart, seen in Figure 4.15.

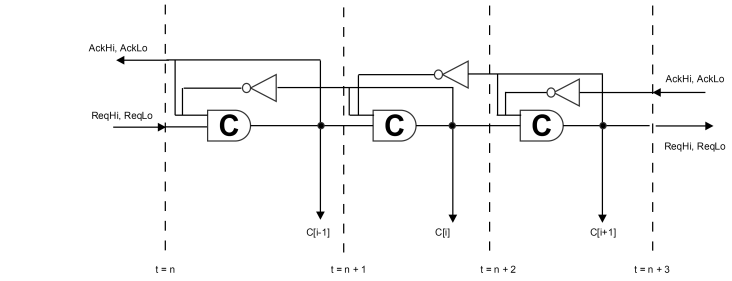



Figure 4.15: A two phase pipeline. Notice the inverted signal is looped back into the **same** C-element as well as the previous one.

Again we show an oscillation using this pipeline seen in Figure 4.16. The pipeline responds with a similar signal propagation of a logical 1 followed by a 'no signal' keyword.

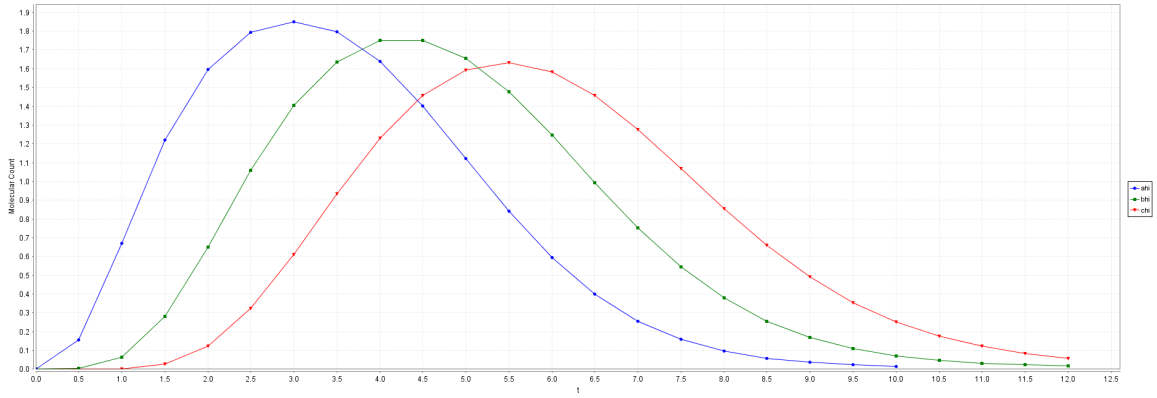



Figure 4.16: A ‘pulse’ of a logical 1 through our 2-phase pipeline construction. Like before the signals ahi, bhi, chi are all present suggesting that the protocol has successfully negotiated the transfer of a 1 followed by a ‘no signal’ to the end of the pipeline.

# Chapter 5

# Evaluation

Our thesis was largely successful in showing the construction of asynchronous components from Chemical Reaction Networks (CRNs). However, the major limitations of this work stem from the methodology. Testing was an extremely time-intensive process, with several experiments taking several days to run. Worse, some experiments were impossible because of memory limitations. This chapter discusses the shortcomings and limitations of this project. The conclusions from this produce ideas for future work which could lead to a faster and better results than those given in the chapters previous. We conclude by discussing how the results from this thesis could be used in physical implementations.

## 5.1 Limitations

### 5.1.1 State Space Explosion

Our models contained many species and reactions. These resulted in many equivalent states and transitions. A major problem we faced was that a slight increase in the molecular count for a species could result in exponentially more states and transitions. For example, the adder was a relatively simple component and yet as we increase the number of molecules the state space increases exponentially.

Larger systems, such as applications of the pipeline, were almost infeasible even with low molecular count in each species. Running any simulation or verification of some property could take a large amount of time. Designs were verified only for small molecular counts and uniform rates.

### 5.1.2 Automated Verification

Adjusting rewards structures and testing for specific times and properties was a manually intensive process. The obvious solution to this problem would have been to implement some script that would automatically query and generate plots for us. However, this was beyond the scope of this project.

A problem with automated verification is that we would need to know the expected result before we received it. This is fine for logical values but, given that molecular count is not binary it would be very hard to guess what bounds we would expect molecular count to fall within.

### 5.1.3 Multi-purpose circuit design

We have defined components which can be used in general computing: latches, logic gates etc. However when it comes to larger systems we have not generalised them in such a way that someone could take the implementation and use it directly within their system. A further stage to our work would be the construction of generic computational components that were multi-purpose for different inputs and tasks. These are referred to as programmable logic devices and are used in memory devices [61].

### 5.1.4 Implementation Issues

Constructing PRISM models from CRNs was eventually done manually as the conversion process from Microsoft's GEC was incomplete, obscure in variable names, and tended to produce errors. Each configuration of input values created a different model. This led to problems with models that took a long time to code, as the values had to be reconfigured and adjusted before testing could resume.

Construction of larger CRN systems was complex. It is fair to compare CRNs, as a language, to that of assembly code, in that coding anything beyond one component took a considerable amount of time. CRNs were very hard to 'debug' as we could only rely on graphical output in the form of species count.

## 5.2 Further Work

### 5.2.1 High-Level Programming Language

As discussed previously, a major problem with the implementation of our designs was the fact that they were written in a language that would closely match assembly code in terms of expressive power. This meant that large systems were taxing to construct and even more taxing to debug. Combined with the fact that testing of larger systems could take several days, the GEC-LBS language is not scalable for multi-component systems. Mentioned briefly in Chapter 2 was the fact that we wrote a python script that would help group and rename components in order to construct larger systems. However, rather than just a python script, there is the possibility of implementing a high-level programming language that would directly compile into CRNs. Therefore we propose that a natural extension of this work would be to implement any of the following:

- A high-level programming language which would compile into the GEC-LBS or PRISM modelling language, alleviating the burden of component design.
- An interpreter for an existing 4th generation programming language, such as *Matlab*, which would take instructions and convert them into chemical reactions or PRISM models.
- A hybrid where a user could override existing component implementations with their own designs, but still write in a high-level language.

A compiler for a high level language would take high level instructions and, using pre-existing component design, would compile into the GEC-LBS language or a similar CRN language. For an example take the following instruction:

$$\text{if ( x and y ) do: z} \tag{5.1}$$

Using our existing AND implementation in the GEC-LBS language, given in Appendix A, we could substitute the inputs for this component with tokens for the inputs x, y. We could then wrap this inside the implementation of an if-statement making sure that z is the conditional result. In this way a user is abstracted from the actual implementation and can focus on verification and simulation. PRISM's modelling language would also be a valid target for compilation, the reason we focus on GEC-LBS language is because of parallels that can be drawn to assembly language.

In a similar way we do not have to produce our own high-level language. Given that we have designed components that would resemble most assembly code instructions we could argue that a high-level language like c or even matlab could be compiled down into the CRNs given. Of course the major set back in this is the efficient representation of numbers as CRNs. A high-level functional language may even be a solution as was done for *Microsoft's DSD* [34]. In this paper the authors examine how low level DSD instructions can be a compiler target for a functional language.

Perhaps the best solution would be a high-level language with some way of overriding compile instructions. For instance, in the above the user may redefine the instructions for the AND operator as:

$$\begin{aligned} &\text{redefine and}\{ \\ &\text{inputA} + \text{inputB} \rightarrow \text{outputA} \\ &\ldots \\ &\} \end{aligned} \tag{5.2}$$

Then the user might execute the original high level statement (5.1). In this way users could write, targeting larger systems, whilst still having flexibility over how the compiler operates.

The use of some higher-level language is a logical extension from the results produced here just as C was the natural progression from assembly code instructions. The problem we face here is that, because we have not formally proved that our components exhibit their expected behaviour, people may be mistrusting of a higher level language. We therefore need to discuss how we could go about formally proving component behaviour.

### 5.2.2 Proofs and Exhaustive Testing

A major problem with our testing was that it was not exhaustive and therefore we did not take into account all unexpected behaviour. Although we verified behaviour for all inputs that we deemed feasible we cannot conclude statements such as "For all inputs in the range $[x..y]$ the component had behaviour which replicates that of the theoretical component in the range $[x..y]$". We could only conclude this if there was either a concrete formalism for interactions of CRNs, and therefore some way to prove equivalence between expected behaviour and actual behaviour produced by

our CRNs, or an exhaustive testing method that also showed equivalences between behaviours. However exhaustive testing might not be practical as it would a long time and many inputs would be pointless to test or covered by similar inputs.

The behaviour of components with high molecular count can be described using Ordinary Differential Equations (ODEs) [14]. If we could describe our target behaviour as an ODE then we could show direct equivalence between a CRNs behaviour and our target component. Unfortunately, because we are dealing with components with low molecular count we cannot show such an equivalence.

Instead, it makes more sense to test through a series of exhaustive simulations, in the hope that the behaviour exhibited is the one we expect. In order to produce exhaustive simulations it would make sense to implement an automated tester. An automated tester would be given a CRN and a range of inputs for which it would output probabilistic behaviour. A major issue with this is that the user would still have to verify behaviour by eye. A possible extension then is for the user to input the expected behaviour in a given range and the automated checker verifies that the CRNs do indeed exhibit this correct behaviour.

### 5.2.3 Physical Implementation

In order for the CRNs constructed within this thesis to be of use, there should ideally exist some physical application which could use these CRN designs as target behaviour. Fortunately in they discuss a method for constructing the base reaction classes: normal, catalytic and auto-catalytic using DNA strand displacement such that any CRN could potentially be encoded as a DNA program [14]. Further to our work here, it would be useful to compile our CRNs into DSD language or some other concrete formalism in order that we could implement our designs. DSD has been used in real-world lab experiments [63].

# Chapter 6

# Conclusion

This thesis attempted to demonstrate the feasibility of asynchronous components and systems built from Chemical Reaction Networks (CRNs) with low molecular count. This combined theory of CRNs and biological diagrammatic notation, with implementations in PRISM's modelling language. Given that the CRN framework has been formally established [28, 49, 50], this thesis focused on applying this to build complex asynchronous networks, which, to our knowledge, has not been done before on this scale. The cornerstone of this thesis was the C-element built upon the circuit of approximate majority. We introduced verified CRN designs for most components, described as essential, in Furber's book on Asynchronous Systems [55]. This culminated in the verification of the Muller-C pipeline and applications of this pipeline in queues, adders, loops and protocols.

Models were constructed using the PRISM model checker, where Continous-Time Markov Chains (CTMCs) modelled an equivalent CRN. We then verified and simulated the behaviour of the CRN using these models. In this way we were able to establish whether or not a CRN exhibited input-output behaviour that closely resembled that of the targeted logical or control flow behaviour.

This thesis successfully demonstrates asynchronous components built from CRNs. This has acute relevance to the field of DNA computing as multiple works have shown a compilation of chemical reactions into physical DNA implementations. With this work, researchers could, in theory, take the CRNs given in this paper and compile these into lab experiments, demonstrating incredible potential for DNA circuit design. CRNs allow a level of abstraction which provides power to implement much larger systems than could normally be implemented using languages such as DSD [49].

However, despite the relative success in implementation of these components, this thesis has several shortcomings. While the feasibility of our CRNs was broadly explored, there was no exhaustive verification process. This issue could be solved with the implementation of an automated verifier.

We encountered state space problems, mainly that an increase of a few molecules and reactions to our model dramatically increased the number of states and transitions. There are many relevant techniques to reduce the state space such as aggregation, bisimulation and lumping [12]. Recently it has been shown that CRNs can be simulated through Linear Noise Approximations [35]. Using this process we could scale the number of molecules to deal with the larger systems developed within this thesis.

In our evaluation we suggest that future work could include the implementation of a programming language which could compile into CRNs. This would abstract the user further from implementation, allowing the production of even larger systems. We also suggest the implementation of an automated verifier, allowing for faster production of components.

We hope to explore the implementation of these CRNs in DNA-based computation in further work. Implementation of these designs in some medium, such as DNA, will prove the success of this thesis.

# Appendix A

# Appendix - Example code

## A.1 AND-Gate Example

### A.1.1 LBS CRN code for an AND gate

```
directive sample 10.0 100

rate rt = 1.0;

//inputs
  init xlo 1.0
| init xhi 0.0
| init ylo 1.0
| init yhi 0.0

//pulleys
| init ze 0.0
| init zw 1.0
| init zq 1.0

//outputs
| init zlo 0.0
| init zhi 0.0

//reactions

|xhi + ze ->{rt} xhi + zhi
|yhi + zw ->{rt} yhi + ze

|xlo + zq ->{rt} xlo + zlo
|ylo + zq ->{rt} ylo + zlo

//for persistance

|zhi + xlo ->{rt} ze + xlo
```

```
|zhi + ylo ->{rt} zw + ylo

|zlo + zhi ->{rt} zhi + zq
```

### A.1.2 AND-gate SBML File

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level2/version4" level="2" version="4">
  <model>
    <listOfUnitDefinitions>
      <unitDefinition id="time">
        <listOfUnits>
          <unit kind="second" scale="0" multiplier="1"/>
        </listOfUnits>
      </unitDefinition>
      <unitDefinition id="substance">
        <listOfUnits>
          <unit kind="mole" scale="-9"/>
        </listOfUnits>
      </unitDefinition>
      <unitDefinition id="volume">
        <listOfUnits>
          <unit kind="litre"/>
        </listOfUnits>
      </unitDefinition>
      <unitDefinition id="dimless">
        <listOfUnits>
          <unit kind="dimensionless"/>
        </listOfUnits>
      </unitDefinition>
      <unitDefinition id="concentration">
        <listOfUnits>
          <unit kind="mole" scale="-9"/>
          <unit kind="litre" exponent="-1"/>
        </listOfUnits>
      </unitDefinition>
      <unitDefinition id="k_unit_2">
        <listOfUnits>
          <unit kind="second" exponent="-1" scale="0" multiplier="1"/>
          <unit kind="mole" exponent="-1" scale="-9"/>
          <unit kind="litre" exponent="1"/>
        </listOfUnits>
      </unitDefinition>
    </listOfUnitDefinitions>
    <listOfCompartments>
      <compartment id="c" size="1.66053886312701E-15"/>
    </listOfCompartments>
```

```
<listOfSpecies>
<species id="s_id0" name="xlo" compartment="c" initialConcentration="1">
  <species id="s_id1" name="xhi" compartment="c" initialConcentration="0">
  <species id="s_id2" name="ylo" compartment="c" initialConcentration="1">
  <species id="s_id3" name="yhi" compartment="c" initialConcentration="0">
  <species id="s_id4" name="ze" compartment="c" initialConcentration="0">
  <species id="s_id5" name="zw" compartment="c" initialConcentration="1">
  <species id="s_id6" name="zq" compartment="c" initialConcentration="1">
  <species id="s_id7" name="zlo" compartment="c" initialConcentration="0">
  <species id="s_id8" name="zhi" compartment="c" initialConcentration="0">
</listOfSpecies>
<listOfReactions>
  <reaction id="r_id9" reversible="false">
    <listOfReactants>
      <speciesReference species="s_id1" stoichiometry="1"/>
      <speciesReference species="s_id4" stoichiometry="1"/>
    </listOfReactants>
    <listOfProducts>
      <speciesReference species="s_id1" stoichiometry="1"/>
      <speciesReference species="s_id8" stoichiometry="1"/>
    </listOfProducts>
    <kineticLaw>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <times/>
          <ci>c_size</ci>
          <ci>k</ci>
          <ci>s_id1</ci>
          <ci>s_id4</ci>
        </apply>
      </math>
      <listOfParameters>
        <parameter id="c_size" value="1.66053886312701E-15" units="volume"/>
        <parameter id="k" value="1" units="k_unit_2"/>
      </listOfParameters>
    </kineticLaw>
  </reaction>
  <reaction id="r_id10" reversible="false">
    <listOfReactants>
      <speciesReference species="s_id3" stoichiometry="1"/>
      <speciesReference species="s_id5" stoichiometry="1"/>
    </listOfReactants>
    <listOfProducts>
      <speciesReference species="s_id3" stoichiometry="1"/>
      <speciesReference species="s_id4" stoichiometry="1"/>
    </listOfProducts>
    <kineticLaw>
```

```
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <times/>
        <ci>c_size</ci>
        <ci>k</ci>
        <ci>s_id3</ci>
        <ci>s_id5</ci>
      </apply>
    </math>
    <listOfParameters>
      <parameter id="c_size" value="1.66053886312701E-15" units="volume"/>
      <parameter id="k" value="1" units="k_unit_2"/>
    </listOfParameters>
  </kineticLaw>
</reaction>
<reaction id="r_id11" reversible="false">
  <listOfReactants>
    <speciesReference species="s_id0" stoichiometry="1"/>
    <speciesReference species="s_id6" stoichiometry="1"/>
  </listOfReactants>
  <listOfProducts>
    <speciesReference species="s_id0" stoichiometry="1"/>
    <speciesReference species="s_id7" stoichiometry="1"/>
  </listOfProducts>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <times/>
        <ci>c_size</ci>
        <ci>k</ci>
        <ci>s_id0</ci>
        <ci>s_id6</ci>
      </apply>
    </math>
    <listOfParameters>
      <parameter id="c_size" value="1.66053886312701E-15" units="volume"/>
      <parameter id="k" value="1" units="k_unit_2"/>
    </listOfParameters>
  </kineticLaw>
</reaction>
<reaction id="r_id12" reversible="false">
  <listOfReactants>
    <speciesReference species="s_id2" stoichiometry="1"/>
    <speciesReference species="s_id6" stoichiometry="1"/>
  </listOfReactants>
  <listOfProducts>
    <speciesReference species="s_id2" stoichiometry="1"/>
```

```
    <speciesReference species="s_id7" stoichiometry="1"/>
  </listOfProducts>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <times/>
        <ci>c_size</ci>
        <ci>k</ci>
        <ci>s_id2</ci>
        <ci>s_id6</ci>
      </apply>
    </math>
    <listOfParameters>
      <parameter id="c_size" value="1.66053886312701E-15" units="volume"/>
      <parameter id="k" value="1" units="k_unit_2"/>
    </listOfParameters>
  </kineticLaw>
</reaction>
<reaction id="r_id13" reversible="false">
  <listOfReactants>
    <speciesReference species="s_id8" stoichiometry="1"/>
    <speciesReference species="s_id0" stoichiometry="1"/>
  </listOfReactants>
  <listOfProducts>
    <speciesReference species="s_id4" stoichiometry="1"/>
    <speciesReference species="s_id0" stoichiometry="1"/>
  </listOfProducts>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <times/>
        <ci>c_size</ci>
        <ci>k</ci>
        <ci>s_id8</ci>
        <ci>s_id0</ci>
      </apply>
    </math>
    <listOfParameters>
      <parameter id="c_size" value="1.66053886312701E-15" units="volume"/>
      <parameter id="k" value="1" units="k_unit_2"/>
    </listOfParameters>
  </kineticLaw>
</reaction>
<reaction id="r_id14" reversible="false">
  <listOfReactants>
    <speciesReference species="s_id8" stoichiometry="1"/>
    <speciesReference species="s_id2" stoichiometry="1"/>
```

```
    </listOfReactants>
    <listOfProducts>
      <speciesReference species="s_id5" stoichiometry="1"/>
      <speciesReference species="s_id2" stoichiometry="1"/>
    </listOfProducts>
    <kineticLaw>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <times/>
          <ci>c_size</ci>
          <ci>k</ci>
          <ci>s_id8</ci>
          <ci>s_id2</ci>
        </apply>
      </math>
      <listOfParameters>
        <parameter id="c_size" value="1.66053886312701E-15" units="volume"/>
        <parameter id="k" value="1" units="k_unit_2"/>
      </listOfParameters>
    </kineticLaw>
  </reaction>
  <reaction id="r_id15" reversible="false">
    <listOfReactants>
      <speciesReference species="s_id7" stoichiometry="1"/>
      <speciesReference species="s_id8" stoichiometry="1"/>
    </listOfReactants>
    <listOfProducts>
      <speciesReference species="s_id8" stoichiometry="1"/>
      <speciesReference species="s_id6" stoichiometry="1"/>
    </listOfProducts>
    <kineticLaw>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <times/>
          <ci>c_size</ci>
          <ci>k</ci>
          <ci>s_id7</ci>
          <ci>s_id8</ci>
        </apply>
      </math>
      <listOfParameters>
        <parameter id="c_size" value="1.66053886312701E-15" units="volume"/>
        <parameter id="k" value="1" units="k_unit_2"/>
      </listOfParameters>
    </kineticLaw>
  </reaction>
</listOfReactions>
```

```
  </model>
</sbml>
```

### A.1.3 AND-gate PRISM File

```
ctmc

const int MAX_AMOUNT = 100;

// Parameters for reaction r_id9
const double k_r_id9 = 1.0;

// Parameters for reaction r_id10
const double k_r_id10 = 1.0;

// Parameters for reaction r_id11
const double k_r_id11 = 1.0;

// Parameters for reaction r_id12
const double k_r_id12 = 1.0;

// Parameters for reaction r_id13
const double k_r_id13 = 1.0;

// Parameters for reaction r_id14
const double k_r_id14 = 1.0;

// Parameters for reaction r_id15
const double k_r_id15 = 1.0;

// Species s_id0 (xlo)
const int s_id0_MIN = 0;
const int s_id0_MAX = MAX_AMOUNT;
module s_id0

    s_id0 : [s_id0_MIN..s_id0_MAX] init 1; // Initial amount 1

    // r_id11
    [r_id11] s_id0 > 0 -> (s_id0'=s_id0-0);
    // r_id13
    [r_id13] s_id0 > 0 -> (s_id0'=s_id0-0);

endmodule

// Species s_id1 (xhi)
const int s_id1_MIN = 0;
const int s_id1_MAX = MAX_AMOUNT;
module s_id1
```

```
    s_id1 : [s_id1_MIN..s_id1_MAX] init 0; // Initial amount 0

    // r_id9
    [r_id9] s_id1 > 0 -> (s_id1'=s_id1-0);

endmodule

// Species s_id2 (ylo)
const int s_id2_MIN = 0;
const int s_id2_MAX = MAX_AMOUNT;
module s_id2

    s_id2 : [s_id2_MIN..s_id2_MAX] init 1; // Initial amount 1

    // r_id12
    [r_id12] s_id2 > 0 -> (s_id2'=s_id2-0);
    // r_id14
    [r_id14] s_id2 > 0 -> (s_id2'=s_id2-0);

endmodule

// Species s_id3 (yhi)
const int s_id3_MIN = 0;
const int s_id3_MAX = MAX_AMOUNT;
module s_id3

    s_id3 : [s_id3_MIN..s_id3_MAX] init 0; // Initial amount 0

    // r_id10
    [r_id10] s_id3 > 0 -> (s_id3'=s_id3-0);

endmodule

// Species s_id4 (ze)
const int s_id4_MIN = 0;
const int s_id4_MAX = MAX_AMOUNT;
module s_id4

    s_id4 : [s_id4_MIN..s_id4_MAX] init 0; // Initial amount 0

    // r_id9
    [r_id9] s_id4 > 0 -> (s_id4'=s_id4-1);
    // r_id10
    [r_id10] s_id4 <= s_id4_MAX-1 -> (s_id4'=s_id4+1);
    // r_id13
    [r_id13] s_id4 <= s_id4_MAX-1 -> (s_id4'=s_id4+1);
```

```
endmodule

// Species s_id5 (zw)
const int s_id5_MIN = 0;
const int s_id5_MAX = MAX_AMOUNT;
module s_id5

    s_id5 : [s_id5_MIN..s_id5_MAX] init 1; // Initial amount 1

    // r_id10
    [r_id10] s_id5 > 0 -> (s_id5'=s_id5-1);
    // r_id14
    [r_id14] s_id5 <= s_id5_MAX-1 -> (s_id5'=s_id5+1);

endmodule

// Species s_id6 (zq)
const int s_id6_MIN = 0;
const int s_id6_MAX = MAX_AMOUNT;
module s_id6

    s_id6 : [s_id6_MIN..s_id6_MAX] init 1; // Initial amount 1

    // r_id11
    [r_id11] s_id6 > 0 -> (s_id6'=s_id6-1);
    // r_id12
    [r_id12] s_id6 > 0 -> (s_id6'=s_id6-1);
    // r_id15
    [r_id15] s_id6 <= s_id6_MAX-1 -> (s_id6'=s_id6+1);

endmodule

// Species s_id7 (zlo)
const int s_id7_MIN = 0;
const int s_id7_MAX = MAX_AMOUNT;
module s_id7

    s_id7 : [s_id7_MIN..s_id7_MAX] init 0; // Initial amount 0

    // r_id11
    [r_id11] s_id7 <= s_id7_MAX-1 -> (s_id7'=s_id7+1);
    // r_id12
    [r_id12] s_id7 <= s_id7_MAX-1 -> (s_id7'=s_id7+1);
    // r_id15
    [r_id15] s_id7 > 0 -> (s_id7'=s_id7-1);
```

```
endmodule

// Species s_id8 (zhi)
const int s_id8_MIN = 0;
const int s_id8_MAX = MAX_AMOUNT;
module s_id8

    s_id8 : [s_id8_MIN..s_id8_MAX] init 0; // Initial amount 0

    // r_id9
    [r_id9] s_id8 <= s_id8_MAX-1 -> (s_id8'=s_id8+1);
    // r_id13
    [r_id13] s_id8 > 0 -> (s_id8'=s_id8-1);
    // r_id14
    [r_id14] s_id8 > 0 -> (s_id8'=s_id8-1);
    // r_id15
    [r_id15] s_id8 > 0 -> (s_id8'=s_id8-0);

endmodule

// Reaction rates
module reaction_rates

    // r_id9
    [r_id9] (k_r_id9*s_id1*s_id4) > 0 -> (k_r_id9*s_id1*s_id4) : true;
    // r_id10
    [r_id10] (k_r_id10*s_id3*s_id5) > 0 -> (k_r_id10*s_id3*s_id5) : true;
    // r_id11
    [r_id11] (k_r_id11*s_id0*s_id6) > 0 -> (k_r_id11*s_id0*s_id6) : true;
    // r_id12
    [r_id12] (k_r_id12*s_id2*s_id6) > 0 -> (k_r_id12*s_id2*s_id6) : true;
    // r_id13
    [r_id13] (k_r_id13*s_id8*s_id0) > 0 -> (k_r_id13*s_id8*s_id0) : true;
    // r_id14
    [r_id14] (k_r_id14*s_id8*s_id2) > 0 -> (k_r_id14*s_id8*s_id2) : true;
    // r_id15
    [r_id15] (k_r_id15*s_id7*s_id8) > 0 -> (k_r_id15*s_id7*s_id8) : true;

endmodule

// Reward structures (one per species)

// Species s_id0 (xlo)
rewards "s_id0" true : s_id0; endrewards
// Species s_id1 (xhi)
rewards "s_id1" true : s_id1; endrewards
// Species s_id2 (ylo)
```

```
rewards "s_id2" true : s_id2; endrewards
// Species s_id3 (yhi)
rewards "s_id3" true : s_id3; endrewards
// Species s_id4 (ze)
rewards "s_id4" true : s_id4; endrewards
// Species s_id5 (zw)
rewards "s_id5" true : s_id5; endrewards
// Species s_id6 (zq)
rewards "s_id6" true : s_id6; endrewards
// Species s_id7 (zlo)
rewards "s_id7" true : s_id7; endrewards
// Species s_id8 (zhi)
rewards "s_id8" true : s_id8; endrewards

// Reward structure for calculating expected times
rewards "time" true : 1; endrewards
```

### A.1.4 AND-Gate Properties for Simulation

```
const double t;

R{"s_id0"}=? [ I = t ]
R{"s_id1"}=? [ I = t ]
R{"s_id2"}=? [ I = t ]
R{"s_id3"}=? [ I = t ]
R{"s_id4"}=? [ I = t ]
R{"s_id5"}=? [ I = t ]
R{"s_id6"}=? [ I = t ]
R{"s_id7"}=? [ I = t ]
R{"s_id8"}=? [ I = t ]
R{"s_id9"}=? [ I = t ]
R{"s_id10"}=? [ I = t ]
```

## A.2 CRN Designs

### A.2.1 Latches

```
directive sample 10.0 100

rate rt = 1.0;

//INPUTS

  init writeOne 5.0
| init writeZero 0.0
```

```
| init mOne 0.0
| init mZero 5.0

| init waitReadOne 5.0
| init waitReadZero 0.0
//OUTPUTS
| init readOne 0.0
| init readZero 0.0

//REACTIONS

| writeOne + mZero ->{rt} writeOne + mOne
| writeZero + mOne ->{rt} writeZero + mZero
| waitReadOne + mOne ->{rt} readOne + mOne
| waitReadZero + mZero ->{rt} readZero + mZero


//| yhi + ylo ->{rt} yhi + yhi
//| ylo + yhi ->{rt} ylo + ylo


--------------------

directive sample 10.0 10

rate rt = 1.0;

//INPUTS

  init writeOne 5.0
| init writeZero 0.0

| init mOne 0.0
| init mZero 5.0

| init waitReadOne 5.0
| init waitReadZero 0.0

| init rOne 0.0
| init rZero 0.0

| init mlambda 5.0
//OUTPUTS
| init readOne 0.0
| init readZero 0.0
```

```
//REACTIONS

| writeOne + mlambda ->{rt} writeOne + mOne
| writeZero + mlambda ->{rt} writeZero + mZero
| rOne + mOne ->{rt} rOne + mlambda
| rZero + mZero ->{rt} rZero + mlambda

| waitReadOne + mOne ->{rt} readOne + mOne
| waitReadZero + mZero ->{rt} readZero + mZero

| readOne ->{rt}
| readZero ->{rt}

| writeOne ->{rt} rOne
```

### A.2.2 C-element

```
directive sample 10.0 100

rate rt = 1.0;

  init zup 1.0
| init znt 0.0
| init zdn 0.0

| init zhi 1.0
| init zlo 0.0

| init xhi 0.0
| init xlo 1.0

| init yhi 0.0
| init ylo 1.0

| zdn + zup ->{rt} znt + znt
| znt + zdn ->{rt} zdn + zdn
| znt + zup ->{rt} zup + zup

| zdn + xhi ->{rt} xhi + znt
| znt+ yhi ->{rt} yhi + zup
| zup + xlo ->{rt} xlo + znt
| znt+ ylo ->{rt} ylo + zdn

| zlo + zup ->{rt} zup + zhi
| zhi + zdn ->{rt} zdn + zlo
```

### A.2.3 Adder

```
directive sample 10.0 100

rate rt = 1.0;

//inputs
  init ahi 0.0
| init alo 0.0

| init bhi 1.0
| init blo 0.0

| init cinhi 0.0
| init cinlo 0.0

| init slo 0.0
| init shi 0.0

| init couthi 0.0
| init coutlo 0.0

//XOR1
| init xorOnelOne 1.0
| init xorOnelTwo 1.0
| init xorOnelThree 0.0
| init xorOnelFour 0.0
| init xorOneHi 0.0
| init xorOneLo 0.0

//XOR2

| init slone 0.0
| init xorTwolTwo 0.0
| init xorTwolThree 0.0
| init xorTwolFour 0.0

//AND1
| init andOnelq  0.0
| init andOnelw 0.0
| init andOnele 0.0
| init andOneHi 0.0
| init andOneLo 0.0

//AND2
| init andTwole 0.0
| init andTwolw 0.0
| init andTwolq 0.0
```

```
| init andTwoHi 0.0
| init andTwoLo 0.0

//OR1
| init orle 0.0
| init orlw 0.0
| init orlq 0.0


//XOR1
//reactions
| ahi + xorOnelFour  ->{rt} ahi +  xorOneHi
| bhi + xorOnelFour ->{rt}  bhi +  xorOneHi

| alo + xorOnelThree ->{rt} alo + xorOnelFour
| blo + xorOnelThree ->{rt} blo + xorOnelFour

| ahi + xorOnelTwo ->{rt} ahi + xorOnelOne
| blo + xorOnelTwo ->{rt} blo + xorOnelOne

| bhi + xorOnelOne ->{rt} bhi + xorOneLo
| alo + xorOnelOne ->{rt} alo + xorOneLo


//for persistance

| xorOneLo + xorOnelFour ->{rt} xorOneLo + xorOnelThree
| xorOneLo + xorOneHi ->{rt} xorOneLo + xorOnelFour

| xorOneHi + xorOnelOne->{rt} xorOneHi + xorOnelTwo
| xorOneHi + xorOneLo ->{rt} xorOneHi + xorOnelOne


//XOR2
| xorOneHi + xorTwolFour  ->{rt} xorOneHi +  shi
| cinhi + xorTwolFour ->{rt}  cinhi +  shi

| xorOneLo + xorTwolThree ->{rt} xorOneLo + xorTwolFour
| cinlo + xorTwolThree ->{rt} cinlo + xorTwolFour

| xorOneHi + xorTwolTwo ->{rt} xorOneHi + slone
| cinlo + xorTwolTwo ->{rt} cinlo + slone

| cinhi + slone ->{rt} cinhi + slo
| slo + slone ->{rt} xorOneLo + slo
```

```
//for persistance

| slo + xorTwolFour ->{rt} slo + xorTwolThree
| slo + shi ->{rt} slo + xorTwolFour

| shi + slone->{rt} shi + xorTwolTwo
| shi + slo ->{rt} shi + slone


//AND1
//reactions

|ahi + andOnele ->{rt} ahi + andOneHi
|bhi + andOnelw ->{rt} bhi + andOnele

|alo + andOnelq ->{rt} alo + andOneLo
|blo + andlq ->{rt} blo + andOneLo

//for persistance

| andOneHi + alo ->{rt} andOnele + alo
| andOneHi + blo ->{rt} andOnelw + blo

| andOneLo + andOneHi ->{rt} andOneHi + andOnelq

//AND2

//reactions

|cinhi + andTwole ->{rt} cinhi + andTwoHi
|xorOneHi + andTwolw ->{rt} xorOneHi + andTwole

|cinlo + andTwolq ->{rt} cinlo + andTwoLo
|xorOneLo + andTwolq ->{rt} xorOneLo + andTwoLo

//for persistance

|andTwoHi + cinlo ->{rt} andTwole + cinlo
|andTwoHi + xorOneLo ->{rt} andTwolw + xorOneLo

|andTwoLo + andTwoHi ->{rt} andTwoHi + andTwolq


//OR

| andOneHi + orle ->{rt} andOneHi + couthi
| andTwoHi + orle ->{rt} andTwoHi + couthi
```

```
| andOneLo + orlw ->{rt} andOneLo + orlq
| andTwoLo + orlq ->{rt} andTwoLo + coutlo


//for persistance
| coutlo + andOneHi ->{rt} orlw + andOneHi
| coutlo + andTwoHi ->{rt} orlq + andTwoHi
| coutlo + couthi ->{rt} coutlo + orle
```

### A.2.4 Muller C Pipeline CRN

```
directive sample 20.0 100

rate rt = 1.0;

 // C ELEMENTS
 //AM SWITCH 1
  init aup 0.0
| init ant 0.0
| init adn 2.0

//AM SWITCH 2
| init bup 0.0
| init bnt 0.0
| init bdn 2.0

//AM SWITCH 3
| init cup 0.0
| init cnt 0.0
| init cdn 2.0

//AMP 1
| init ahi 0.0
| init alo 5.0

//AMP 2
| init bhi 0.0
| init blo 5.0

//AMP 3
| init chi 0.0
| init clo 5.0


//INPUTS
```

```
| init acchi 5.0
| init acclo 0.0

| init reqhi 5.0
| init reqlo 0.0


//NOT

| init bNotUp 0.0
| init bNotNt 5.0
| init bNotDown 0.0

| init cNotUp 0.0
| init cNotNt 5.0
| init cNotDown 0.0


//AREA 1
| adn + aup ->{rt} ant + ant
| ant + adn ->{rt} adn + adn
| ant + aup ->{rt} aup + aup

| adn + bNotUp ->{rt} ant + bNotUp
| ant + reqhi ->{rt} aup + reqhi
| aup + bNotDown->{rt} ant + bNotDown
| ant + reqlo ->{rt} adn + reqlo

| alo + aup ->{rt} ahi + aup
| ahi + adn ->{rt} alo + adn

//AREA 2
| bdn + bup ->{rt} bnt + bnt
| bnt + bdn ->{rt} bdn + bdn
| bnt + bup ->{rt} bup + bup

| bdn + cNotUp ->{rt} bnt + cNotUp
| bnt + ahi ->{rt} bup + ahi
| bup + cNotDown ->{rt} bnt + cNotDown
| bnt + alo ->{rt} bdn + alo

| blo + bup ->{rt} bhi + bup
| bhi + bdn ->{rt} blo + bdn

| bhi + bNotNt ->{rt} bhi + bNotDown
| bhi + bNotUp ->{rt} bhi + bNotDown
```

```
| blo + bNotNt ->{rt} blo + bNotUp
| blo + bNotDown ->{rt} blo + bNotUp

//AREA 3
| cdn + cup ->{rt} cnt + cnt
| cnt + cdn ->{rt} cdn + cdn
| cnt + cup ->{rt} cup + cup

| cdn + acchi ->{rt} cnt + acchi
| cnt + bhi ->{rt} cup + bhi
| cup + acclo ->{rt} cnt + acclo
| cnt + blo ->{rt} cdn + blo

| clo + cup ->{rt} chi + cup
| chi + cdn ->{rt} clo + cdn

| chi + cNotNt ->{rt} chi + cNotDown
| chi + cNotUp ->{rt} chi + cNotDown

| clo + cNotNt ->{rt} clo + cNotUp
| clo + cNotDown ->{rt} clo + cNotUp
```

### A.2.5 Queue Structure

```
directive sample 20.0 100

rate rt = 1.0;

 // C ELEMENTS
 //AM SWITCH 1
  init aup 0.0
| init ant 0.0
| init adn 1.0

//AM SWITCH 2
| init bup 0.0
| init bnt 0.0
| init bdn 1.0

//AM SWITCH 3
| init cup 0.0
| init cnt 0.0
| init cdn 1.0

//AMP 1
| init ahi 0.0
| init alo 1.0
```

```
//AMP 2
| init bhi 0.0
| init blo 1.0

//AMP 3
| init chi 0.0
| init clo 1.0


//INPUTS
| init reqhi 1.0
| init reqlo 0.0


//NOT

| init aNotUp 1.0
| init aNotDown 0.0

| init bNotUp 1.0
| init bNotDown 0.0

| init cNotUp 1.0
| init cNotDown 0.0

//MEM 1
| init amOne  1.0
| init amZero 0.0

//MEM 2
| init bmOne  0.0
| init bmZero 1.0

//MEM 3
| init cmOne 0.0
| init cmZero 1.0

//MEM 4
| init outOne 0.0
| init outZero 1.0



//GATE 2

//AREA 1
| adn + aup ->{rt} ant + ant
```

```
| ant + adn ->{rt} adn + adn
| ant + aup ->{rt} aup + aup

| adn + bNotUp ->{rt} ant + bNotUp
| ant + reqhi ->{rt} aup + reqhi
| aup + bNotDown->{rt} ant + bNotDown
| ant + reqlo ->{rt} adn + reqlo

| alo + aup ->{rt} ahi + aup
| ahi + adn ->{rt} alo + adn

| alo + aNotDown ->{rt} alo + aNotUp
| ahi + aNotUp ->{rt} ahi + aNotDown

| ahi + amOne ->{rt} ahi + bmOne
| ahi + amZero ->{rt} ahi + bmZero




//AREA 2
| bdn + bup ->{rt} bnt + bnt
| bnt + bdn ->{rt} bdn + bdn
| bnt + bup ->{rt} bup + bup

| bdn + cNotUp ->{rt} bnt + cNotUp
| bnt + ahi ->{rt} bup + ahi
| bup + cNotDown ->{rt} bnt + cNotDown
| bnt + alo ->{rt} bdn + alo

| blo + bup ->{rt} bhi + bup
| bhi + bdn ->{rt} blo + bdn

| blo + bNotDown ->{rt} blo + bNotUp
| bhi + bNotUp ->{rt} bhi + bNotDown


| bhi + bmOne ->{rt} bhi + cmOne
| bhi + bmZero ->{rt} bhi + cmZero

//AREA 3
| cdn + cup ->{rt} cnt + cnt
| cnt + cdn ->{rt} cdn + cdn
| cnt + cup ->{rt} cup + cup

| cdn + cNotUp ->{rt} cnt + cNotUp
| cnt + bhi ->{rt} cup + bhi
```

```
| cup + cNotDown ->{rt} cnt + cNotDown
| cnt + blo ->{rt} cdn + blo

| clo + cup ->{rt} chi + cup
| chi + cdn ->{rt} clo + cdn

| clo + cNotDown ->{rt} clo + cNotUp
| chi + cNotUp ->{rt} chi + cNotDown

| chi + reqhi ->{rt} chi + reqlo
| clo + reqlo ->{rt} clo + reqhi

| chi + cmOne ->{rt} chi + outOne
| chi + cmZero ->{rt} chi + outZero
```

### A.2.6 Loop PRISM model

```
ctmc

const int MAX_AMOUNT = 1;

// Parameters for reaction r_id23
const double k_r_id23 = 1.0;

// Parameters for reaction r_id24
const double k_r_id24 = 1.0;

// Parameters for reaction r_id25
const double k_r_id25 = 1.0;

// Parameters for reaction r_id26
const double k_r_id26 = 1.0;

// Parameters for reaction r_id27
const double k_r_id27 = 1.0;

// Parameters for reaction r_id28
const double k_r_id28 = 1.0;

// Parameters for reaction r_id29
const double k_r_id29 = 1.0;

// Parameters for reaction r_id30
const double k_r_id30 = 1.0;

// Parameters for reaction r_id31
const double k_r_id31 = 1.0;
```

```
// Parameters for reaction r_id32
const double k_r_id32 = 1.0;

// Parameters for reaction r_id33
const double k_r_id33 = 1.0;

// Parameters for reaction r_id34
const double k_r_id34 = 1.0;

// Parameters for reaction r_id35
const double k_r_id35 = 1.0;

// Parameters for reaction r_id36
const double k_r_id36 = 1.0;

// Parameters for reaction r_id37
const double k_r_id37 = 1.0;

// Parameters for reaction r_id38
const double k_r_id38 = 1.0;

// Parameters for reaction r_id39
const double k_r_id39 = 1.0;

// Parameters for reaction r_id40
const double k_r_id40 = 1.0;

// Parameters for reaction r_id41
const double k_r_id41 = 1.0;

// Parameters for reaction r_id42
const double k_r_id42 = 1.0;

// Parameters for reaction r_id43
const double k_r_id43 = 1.0;

// Parameters for reaction r_id44
const double k_r_id44 = 1.0;

// Parameters for reaction r_id45
const double k_r_id45 = 1.0;

// Parameters for reaction r_id46
const double k_r_id46 = 1.0;

// Parameters for reaction r_id47
const double k_r_id47 = 1.0;
```

```
// Parameters for reaction r_id48
const double k_r_id48 = 1.0;

// Parameters for reaction r_id49
const double k_r_id49 = 1.0;

// Parameters for reaction r_id50
const double k_r_id50 = 1.0;

// Parameters for reaction r_id51
const double k_r_id51 = 1.0;

// Parameters for reaction r_id52
const double k_r_id52 = 1.0;

// Parameters for reaction r_id53
const double k_r_id53 = 1.0;

// Parameters for reaction r_id54
const double k_r_id54 = 1.0;

// Parameters for reaction r_id55
const double k_r_id55 = 1.0;

// Parameters for reaction r_id56
const double k_r_id56 = 1.0;

// Parameters for reaction r_id57
const double k_r_id57 = 1.0;

// Species s_id0 (aup)
const int s_id0_MIN = 0;
const int s_id0_MAX = MAX_AMOUNT;
module s_id0

    s_id0 : [s_id0_MIN..s_id0_MAX] init 0; // Initial amount 0

    // r_id23
    [r_id23] s_id0 > 0 -> (s_id0'=s_id0-1);
    // r_id25
    [r_id25] s_id0 <= s_id0_MAX-1 -> (s_id0'=s_id0+1);
    // r_id27
    [r_id27] s_id0 <= s_id0_MAX-1 -> (s_id0'=s_id0+1);
    // r_id28
    [r_id28] s_id0 > 0 -> (s_id0'=s_id0-1);
    // r_id30
```

```
    [r_id30] s_id0 > 0 -> (s_id0'=s_id0-0);

endmodule

// Species s_id1 (ant)
const int s_id1_MIN = 0;
const int s_id1_MAX = MAX_AMOUNT;
module s_id1

    s_id1 : [s_id1_MIN..s_id1_MAX] init 0; // Initial amount 0

    // r_id23
    [r_id23] s_id1 <= s_id1_MAX-2 -> (s_id1'=s_id1+2);
    // r_id24
    [r_id24] s_id1 > 0 -> (s_id1'=s_id1-1);
    // r_id25
    [r_id25] s_id1 > 0 -> (s_id1'=s_id1-1);
    // r_id26
    [r_id26] s_id1 <= s_id1_MAX-1 -> (s_id1'=s_id1+1);
    // r_id27
    [r_id27] s_id1 > 0 -> (s_id1'=s_id1-1);
    // r_id28
    [r_id28] s_id1 <= s_id1_MAX-1 -> (s_id1'=s_id1+1);
    // r_id29
    [r_id29] s_id1 > 0 -> (s_id1'=s_id1-1);

endmodule

// Species s_id2 (adn)
const int s_id2_MIN = 0;
const int s_id2_MAX = MAX_AMOUNT;
module s_id2

    s_id2 : [s_id2_MIN..s_id2_MAX] init 1; // Initial amount 1

    // r_id23
    [r_id23] s_id2 > 0 -> (s_id2'=s_id2-1);
    // r_id24
    [r_id24] s_id2 <= s_id2_MAX-1 -> (s_id2'=s_id2+1);
    // r_id26
    [r_id26] s_id2 > 0 -> (s_id2'=s_id2-1);
    // r_id29
    [r_id29] s_id2 <= s_id2_MAX-1 -> (s_id2'=s_id2+1);
    // r_id31
    [r_id31] s_id2 > 0 -> (s_id2'=s_id2-0);

endmodule
```

```
// Species s_id3 (bup)
const int s_id3_MIN = 0;
const int s_id3_MAX = MAX_AMOUNT;
module s_id3

    s_id3 : [s_id3_MIN..s_id3_MAX] init 0; // Initial amount 0

    // r_id34
    [r_id34] s_id3 > 0 -> (s_id3'=s_id3-1);
    // r_id36
    [r_id36] s_id3 <= s_id3_MAX-1 -> (s_id3'=s_id3+1);
    // r_id38
    [r_id38] s_id3 <= s_id3_MAX-1 -> (s_id3'=s_id3+1);
    // r_id39
    [r_id39] s_id3 > 0 -> (s_id3'=s_id3-1);
    // r_id41
    [r_id41] s_id3 > 0 -> (s_id3'=s_id3-0);

endmodule

// Species s_id4 (bnt)
const int s_id4_MIN = 0;
const int s_id4_MAX = MAX_AMOUNT;
module s_id4

    s_id4 : [s_id4_MIN..s_id4_MAX] init 0; // Initial amount 0

    // r_id34
    [r_id34] s_id4 <= s_id4_MAX-2 -> (s_id4'=s_id4+2);
    // r_id35
    [r_id35] s_id4 > 0 -> (s_id4'=s_id4-1);
    // r_id36
    [r_id36] s_id4 > 0 -> (s_id4'=s_id4-1);
    // r_id37
    [r_id37] s_id4 <= s_id4_MAX-1 -> (s_id4'=s_id4+1);
    // r_id38
    [r_id38] s_id4 > 0 -> (s_id4'=s_id4-1);
    // r_id39
    [r_id39] s_id4 <= s_id4_MAX-1 -> (s_id4'=s_id4+1);
    // r_id40
    [r_id40] s_id4 > 0 -> (s_id4'=s_id4-1);

endmodule

// Species s_id5 (bdn)
const int s_id5_MIN = 0;
```

```
const int s_id5_MAX = MAX_AMOUNT;
module s_id5

    s_id5 : [s_id5_MIN..s_id5_MAX] init 1; // Initial amount 1

    // r_id34
    [r_id34] s_id5 > 0 -> (s_id5'=s_id5-1);
    // r_id35
    [r_id35] s_id5 <= s_id5_MAX-1 -> (s_id5'=s_id5+1);
    // r_id37
    [r_id37] s_id5 > 0 -> (s_id5'=s_id5-1);
    // r_id40
    [r_id40] s_id5 <= s_id5_MAX-1 -> (s_id5'=s_id5+1);
    // r_id42
    [r_id42] s_id5 > 0 -> (s_id5'=s_id5-0);

endmodule

// Species s_id6 (cup)
const int s_id6_MIN = 0;
const int s_id6_MAX = MAX_AMOUNT;
module s_id6

    s_id6 : [s_id6_MIN..s_id6_MAX] init 0; // Initial amount 0

    // r_id45
    [r_id45] s_id6 > 0 -> (s_id6'=s_id6-1);
    // r_id47
    [r_id47] s_id6 <= s_id6_MAX-1 -> (s_id6'=s_id6+1);
    // r_id49
    [r_id49] s_id6 <= s_id6_MAX-1 -> (s_id6'=s_id6+1);
    // r_id50
    [r_id50] s_id6 > 0 -> (s_id6'=s_id6-1);
    // r_id52
    [r_id52] s_id6 > 0 -> (s_id6'=s_id6-0);

endmodule

// Species s_id7 (cnt)
const int s_id7_MIN = 0;
const int s_id7_MAX = MAX_AMOUNT;
module s_id7

    s_id7 : [s_id7_MIN..s_id7_MAX] init 0; // Initial amount 0

    // r_id45
    [r_id45] s_id7 <= s_id7_MAX-2 -> (s_id7'=s_id7+2);
```

```
    // r_id46
    [r_id46] s_id7 > 0 -> (s_id7'=s_id7-1);
    // r_id47
    [r_id47] s_id7 > 0 -> (s_id7'=s_id7-1);
    // r_id48
    [r_id48] s_id7 <= s_id7_MAX-1 -> (s_id7'=s_id7+1);
    // r_id49
    [r_id49] s_id7 > 0 -> (s_id7'=s_id7-1);
    // r_id50
    [r_id50] s_id7 <= s_id7_MAX-1 -> (s_id7'=s_id7+1);
    // r_id51
    [r_id51] s_id7 > 0 -> (s_id7'=s_id7-1);

endmodule

// Species s_id8 (cdn)
const int s_id8_MIN = 0;
const int s_id8_MAX = MAX_AMOUNT;
module s_id8

    s_id8 : [s_id8_MIN..s_id8_MAX] init 1; // Initial amount 1

    // r_id45
    [r_id45] s_id8 > 0 -> (s_id8'=s_id8-1);
    // r_id46
    [r_id46] s_id8 <= s_id8_MAX-1 -> (s_id8'=s_id8+1);
    // r_id48
    [r_id48] s_id8 > 0 -> (s_id8'=s_id8-1);
    // r_id51
    [r_id51] s_id8 <= s_id8_MAX-1 -> (s_id8'=s_id8+1);
    // r_id53
    [r_id53] s_id8 > 0 -> (s_id8'=s_id8-0);

endmodule

// Species s_id9 (ahi)
const int s_id9_MIN = 0;
const int s_id9_MAX = MAX_AMOUNT;
module s_id9

    s_id9 : [s_id9_MIN..s_id9_MAX] init 0; // Initial amount 0

    // r_id30
    [r_id30] s_id9 <= s_id9_MAX-1 -> (s_id9'=s_id9+1);
    // r_id31
    [r_id31] s_id9 > 0 -> (s_id9'=s_id9-1);
    // r_id33
```

```
    [r_id33] s_id9 > 0 -> (s_id9'=s_id9-0);
    // r_id38
    [r_id38] s_id9 > 0 -> (s_id9'=s_id9-0);

endmodule

// Species s_id10 (alo)
const int s_id10_MIN = 0;
const int s_id10_MAX = MAX_AMOUNT;
module s_id10

    s_id10 : [s_id10_MIN..s_id10_MAX] init 1; // Initial amount 1

    // r_id30
    [r_id30] s_id10 > 0 -> (s_id10'=s_id10-1);
    // r_id31
    [r_id31] s_id10 <= s_id10_MAX-1 -> (s_id10'=s_id10+1);
    // r_id32
    [r_id32] s_id10 > 0 -> (s_id10'=s_id10-0);
    // r_id40
    [r_id40] s_id10 > 0 -> (s_id10'=s_id10-0);

endmodule

// Species s_id11 (bhi)
const int s_id11_MIN = 0;
const int s_id11_MAX = MAX_AMOUNT;
module s_id11

    s_id11 : [s_id11_MIN..s_id11_MAX] init 0; // Initial amount 0

    // r_id41
    [r_id41] s_id11 <= s_id11_MAX-1 -> (s_id11'=s_id11+1);
    // r_id42
    [r_id42] s_id11 > 0 -> (s_id11'=s_id11-1);
    // r_id44
    [r_id44] s_id11 > 0 -> (s_id11'=s_id11-0);
    // r_id49
    [r_id49] s_id11 > 0 -> (s_id11'=s_id11-0);

endmodule

// Species s_id12 (blo)
const int s_id12_MIN = 0;
const int s_id12_MAX = MAX_AMOUNT;
module s_id12
```

```
    s_id12 : [s_id12_MIN..s_id12_MAX] init 1; // Initial amount 1

    // r_id41
    [r_id41] s_id12 > 0 -> (s_id12'=s_id12-1);
    // r_id42
    [r_id42] s_id12 <= s_id12_MAX-1 -> (s_id12'=s_id12+1);
    // r_id43
    [r_id43] s_id12 > 0 -> (s_id12'=s_id12-0);
    // r_id51
    [r_id51] s_id12 > 0 -> (s_id12'=s_id12-0);

endmodule

// Species s_id13 (chi)
const int s_id13_MIN = 0;
const int s_id13_MAX = MAX_AMOUNT;
module s_id13

    s_id13 : [s_id13_MIN..s_id13_MAX] init 0; // Initial amount 0

    // r_id52
    [r_id52] s_id13 <= s_id13_MAX-1 -> (s_id13'=s_id13+1);
    // r_id53
    [r_id53] s_id13 > 0 -> (s_id13'=s_id13-1);
    // r_id55
    [r_id55] s_id13 > 0 -> (s_id13'=s_id13-0);
    // r_id56
    [r_id56] s_id13 > 0 -> (s_id13'=s_id13-0);

endmodule

// Species s_id14 (clo)
const int s_id14_MIN = 0;
const int s_id14_MAX = MAX_AMOUNT;
module s_id14

    s_id14 : [s_id14_MIN..s_id14_MAX] init 1; // Initial amount 1

    // r_id52
    [r_id52] s_id14 > 0 -> (s_id14'=s_id14-1);
    // r_id53
    [r_id53] s_id14 <= s_id14_MAX-1 -> (s_id14'=s_id14+1);
    // r_id54
    [r_id54] s_id14 > 0 -> (s_id14'=s_id14-0);
    // r_id57
    [r_id57] s_id14 > 0 -> (s_id14'=s_id14-0);
```

```
endmodule

// Species s_id15 (reqhi)
const int s_id15_MIN = 0;
const int s_id15_MAX = MAX_AMOUNT;
module s_id15

    s_id15 : [s_id15_MIN..s_id15_MAX] init 1; // Initial amount 1

    // r_id27
    [r_id27] s_id15 > 0 -> (s_id15'=s_id15-0);
    // r_id56
    [r_id56] s_id15 > 0 -> (s_id15'=s_id15-1);
    // r_id57
    [r_id57] s_id15 <= s_id15_MAX-1 -> (s_id15'=s_id15+1);

endmodule

// Species s_id16 (reqlo)
const int s_id16_MIN = 0;
const int s_id16_MAX = MAX_AMOUNT;
module s_id16

    s_id16 : [s_id16_MIN..s_id16_MAX] init 0; // Initial amount 0

    // r_id29
    [r_id29] s_id16 > 0 -> (s_id16'=s_id16-0);
    // r_id56
    [r_id56] s_id16 <= s_id16_MAX-1 -> (s_id16'=s_id16+1);
    // r_id57
    [r_id57] s_id16 > 0 -> (s_id16'=s_id16-1);

endmodule

// Species s_id17 (aNotUp)
const int s_id17_MIN = 0;
const int s_id17_MAX = MAX_AMOUNT;
module s_id17

    s_id17 : [s_id17_MIN..s_id17_MAX] init 1; // Initial amount 1

    // r_id32
    [r_id32] s_id17 <= s_id17_MAX-1 -> (s_id17'=s_id17+1);
    // r_id33
    [r_id33] s_id17 > 0 -> (s_id17'=s_id17-1);

endmodule
```

```
// Species s_id18 (aNotDown)
const int s_id18_MIN = 0;
const int s_id18_MAX = MAX_AMOUNT;
module s_id18

    s_id18 : [s_id18_MIN..s_id18_MAX] init 0; // Initial amount 0

    // r_id32
    [r_id32] s_id18 > 0 -> (s_id18'=s_id18-1);
    // r_id33
    [r_id33] s_id18 <= s_id18_MAX-1 -> (s_id18'=s_id18+1);

endmodule

// Species s_id19 (bNotUp)
const int s_id19_MIN = 0;
const int s_id19_MAX = MAX_AMOUNT;
module s_id19

    s_id19 : [s_id19_MIN..s_id19_MAX] init 1; // Initial amount 1

    // r_id26
    [r_id26] s_id19 > 0 -> (s_id19'=s_id19-0);
    // r_id43
    [r_id43] s_id19 <= s_id19_MAX-1 -> (s_id19'=s_id19+1);
    // r_id44
    [r_id44] s_id19 > 0 -> (s_id19'=s_id19-1);

endmodule

// Species s_id20 (bNotDown)
const int s_id20_MIN = 0;
const int s_id20_MAX = MAX_AMOUNT;
module s_id20

    s_id20 : [s_id20_MIN..s_id20_MAX] init 0; // Initial amount 0

    // r_id28
    [r_id28] s_id20 > 0 -> (s_id20'=s_id20-0);
    // r_id43
    [r_id43] s_id20 > 0 -> (s_id20'=s_id20-1);
    // r_id44
    [r_id44] s_id20 <= s_id20_MAX-1 -> (s_id20'=s_id20+1);

endmodule
```

```
// Species s_id21 (cNotUp)
const int s_id21_MIN = 0;
const int s_id21_MAX = MAX_AMOUNT;
module s_id21

    s_id21 : [s_id21_MIN..s_id21_MAX] init 1; // Initial amount 1

    // r_id37
    [r_id37] s_id21 > 0 -> (s_id21'=s_id21-0);
    // r_id48
    [r_id48] s_id21 > 0 -> (s_id21'=s_id21-0);
    // r_id54
    [r_id54] s_id21 <= s_id21_MAX-1 -> (s_id21'=s_id21+1);
    // r_id55
    [r_id55] s_id21 > 0 -> (s_id21'=s_id21-1);

endmodule

// Species s_id22 (cNotDown)
const int s_id22_MIN = 0;
const int s_id22_MAX = MAX_AMOUNT;
module s_id22

    s_id22 : [s_id22_MIN..s_id22_MAX] init 0; // Initial amount 0

    // r_id39
    [r_id39] s_id22 > 0 -> (s_id22'=s_id22-0);
    // r_id50
    [r_id50] s_id22 > 0 -> (s_id22'=s_id22-0);
    // r_id54
    [r_id54] s_id22 > 0 -> (s_id22'=s_id22-1);
    // r_id55
    [r_id55] s_id22 <= s_id22_MAX-1 -> (s_id22'=s_id22+1);

endmodule

// Reaction rates
module reaction_rates

    // r_id23
    [r_id23]  (k_r_id23*s_id2*s_id0) > 0 -> (k_r_id23*s_id2*s_id0) : true;
    // r_id24
    [r_id24] (k_r_id24*s_id1*s_id2) > 0 -> (k_r_id24*s_id1*s_id2) : true;
    // r_id25
    [r_id25] (k_r_id25*s_id1*s_id0)  > 0 -> (k_r_id25*s_id1*s_id0) : true;
    // r_id26
    [r_id26] (k_r_id26*s_id2*s_id19) > 0 -> (k_r_id26*s_id2*s_id19) : true;
```

```
// r_id27
[r_id27] (k_r_id27*s_id1*s_id15) > 0 -> (k_r_id27*s_id1*s_id15) : true;
// r_id28
[r_id28] (k_r_id28*s_id0*s_id20) > 0 -> (k_r_id28*s_id0*s_id20) : true;
// r_id29
[r_id29] (k_r_id29*s_id1*s_id16) > 0 -> (k_r_id29*s_id1*s_id16) : true;
// r_id30
[r_id30] (k_r_id30*s_id10*s_id0) > 0 -> (k_r_id30*s_id10*s_id0) : true;
// r_id31
[r_id31] (k_r_id31*s_id9*s_id2) > 0 -> (k_r_id31*s_id9*s_id2) : true;
// r_id32
[r_id32] (k_r_id32*s_id10*s_id18) > 0 -> (k_r_id32*s_id10*s_id18) : true;
// r_id33
[r_id33] (k_r_id33*s_id9*s_id17) > 0 -> (k_r_id33*s_id9*s_id17) : true;
// r_id34
[r_id34] (k_r_id34*s_id5*s_id3) > 0 -> (k_r_id34*s_id5*s_id3) : true;
// r_id35
[r_id35] (k_r_id35*s_id4*s_id5) > 0 -> (k_r_id35*s_id4*s_id5) : true;
// r_id36
[r_id36] (k_r_id36*s_id4*s_id3) > 0 -> (k_r_id36*s_id4*s_id3) : true;
// r_id37
[r_id37] (k_r_id37*s_id5*s_id21) > 0 -> (k_r_id37*s_id5*s_id21) : true;
// r_id38
[r_id38] (k_r_id38*s_id4*s_id9) > 0 -> (k_r_id38*s_id4*s_id9) : true;
// r_id39
[r_id39] (k_r_id39*s_id3*s_id22) > 0 -> (k_r_id39*s_id3*s_id22) : true;
// r_id40
[r_id40] (k_r_id40*s_id4*s_id10) > 0 -> (k_r_id40*s_id4*s_id10) : true;
// r_id41
[r_id41] (k_r_id41*s_id12*s_id3) > 0 -> (k_r_id41*s_id12*s_id3) : true;
// r_id42
[r_id42] (k_r_id42*s_id11*s_id5) > 0 -> (k_r_id42*s_id11*s_id5) : true;
// r_id43
[r_id43] (k_r_id43*s_id12*s_id20) > 0 -> (k_r_id43*s_id12*s_id20) : true;
// r_id44
[r_id44] (k_r_id44*s_id11*s_id19) > 0 -> (k_r_id44*s_id11*s_id19) : true;
// r_id45
[r_id45] (k_r_id45*s_id8*s_id6) > 0 -> (k_r_id45*s_id8*s_id6) : true;
// r_id46
[r_id46] (k_r_id46*s_id7*s_id8) > 0 -> (k_r_id46*s_id7*s_id8) : true;
// r_id47
[r_id47] (k_r_id47*s_id7*s_id6) > 0 -> (k_r_id47*s_id7*s_id6) : true;
// r_id48
[r_id48] (k_r_id48*s_id8*s_id21) > 0 -> (k_r_id48*s_id8*s_id21) : true;
// r_id49
[r_id49] (k_r_id49*s_id7*s_id11) > 0 -> (k_r_id49*s_id7*s_id11) : true;
// r_id50
```

```
    [r_id50] (k_r_id50*s_id6*s_id22) > 0 -> (k_r_id50*s_id6*s_id22) : true;
    // r_id51
    [r_id51] (k_r_id51*s_id7*s_id12) > 0 -> (k_r_id51*s_id7*s_id12) : true;
    // r_id52
    [r_id52] (k_r_id52*s_id14*s_id6) > 0 -> (k_r_id52*s_id14*s_id6) : true;
    // r_id53
    [r_id53] (k_r_id53*s_id13*s_id8) > 0 -> (k_r_id53*s_id13*s_id8) : true;
    // r_id54
    [r_id54] (k_r_id54*s_id14*s_id22) > 0 -> (k_r_id54*s_id14*s_id22) : true;
    // r_id55
    [r_id55] (k_r_id55*s_id13*s_id21) > 0 -> (k_r_id55*s_id13*s_id21) : true;
    // r_id56
    [r_id56] (k_r_id56*s_id13*s_id15) > 0 -> (k_r_id56*s_id13*s_id15) : true;
    // r_id57
    [r_id57] (k_r_id57*s_id14*s_id16) > 0 -> (k_r_id57*s_id14*s_id16) : true;

endmodule

// Reward structures (one per species)

// Species s_id0 (aup)
rewards "s_id0" true : s_id0; endrewards
// Species s_id1 (ant)
rewards "s_id1" true : s_id1; endrewards
// Species s_id2 (adn)
rewards "s_id2" true : s_id2; endrewards
// Species s_id3 (bup)
rewards "s_id3" true : s_id3; endrewards
// Species s_id4 (bnt)
rewards "s_id4" true : s_id4; endrewards
// Species s_id5 (bdn)
rewards "s_id5" true : s_id5; endrewards
// Species s_id6 (cup)
rewards "s_id6" true : s_id6; endrewards
// Species s_id7 (cnt)
rewards "s_id7" true : s_id7; endrewards
// Species s_id8 (cdn)
rewards "s_id8" true : s_id8; endrewards
// Species s_id9 (ahi)
rewards "s_id9" true : s_id9; endrewards
// Species s_id10 (alo)
rewards "s_id10" true : s_id10; endrewards
// Species s_id11 (bhi)
rewards "s_id11" true : s_id11; endrewards
// Species s_id12 (blo)
rewards "s_id12" true : s_id12; endrewards
// Species s_id13 (chi)
```

```
rewards "s_id13" true : s_id13; endrewards
// Species s_id14 (clo)
rewards "s_id14" true : s_id14; endrewards
// Species s_id15 (reqhi)
rewards "s_id15" true : s_id15; endrewards
// Species s_id16 (reqlo)
rewards "s_id16" true : s_id16; endrewards
// Species s_id17 (aNotUp)
rewards "s_id17" true : s_id17; endrewards
// Species s_id18 (aNotDown)
rewards "s_id18" true : s_id18; endrewards
// Species s_id19 (bNotUp)
rewards "s_id19" true : s_id19; endrewards
// Species s_id20 (bNotDown)
rewards "s_id20" true : s_id20; endrewards
// Species s_id21 (cNotUp)
rewards "s_id21" true : s_id21; endrewards
// Species s_id22 (cNotDown)
rewards "s_id22" true : s_id22; endrewards

// Reward structure for calculating expected times
rewards "time" true : 1; endrewards
```

### A.2.7 Four-Phase Protocol CRN

```
directive sample 20.0 100

rate rt = 1.0;

 // C ELEMENTS
 //AM SWITCH 1
  init adtup 0.0
| init adtnt 0.0
| init adtdn 1.0
| init adthi 0.0
| init adtlo 1.0

| init adfup 0.0
| init adfnt 0.0
| init adfdn 1.0
| init adfhi 0.0
| init adflo 1.0

//pulleys
| init ae 1.0
| init aw 1.0
```

```
| init aq 0.0

//outputs
| init alo 0.0
| init ahi 0.0

| init aNotUp 1.0
| init aNotDown 0.0


//AM SWITCH 2
  init bdtup 0.0
| init bdtnt 0.0
| init bdtdn 1.0
| init bdthi 0.0
| init bdtlo 1.0

| init bdfup 0.0
| init bdfnt 0.0
| init bdfdn 1.0
| init bdfhi 0.0
| init bdflo 1.0

//pulleys
| init be 1.0
| init bw 1.0
| init bq 0.0

//outputs
| init blo 0.0
| init bhi 0.0

| init bNotUp 1.0
| init bNotDown 0.0

//AM SWITCH 3
  init cdtup 0.0
| init cdtnt 0.0
| init cdtdn 1.0
| init cdthi 0.0
| init cdtlo 1.0

| init cdfup 0.0
| init cdfnt 0.0
| init cdfdn 1.0
| init cdfhi 0.0
| init cdflo 1.0
```

```
//pulleys
| init ce 1.0
| init cw 1.0
| init cq 0.0

//outputs
| init clo 0.0
| init chi 0.0

| init cNotUp 1.0
| init cNotDown 0.0


//INPUTS
| init reqthi 1.0
| init reqtlo 0.0
| init reqfhi 1.0
| init reqflo 0.0



//AREA 1
| adfdn + adfup ->{rt} adfnt + adfnt
| adfnt + adfdn ->{rt} adfdn + adfdn
| adfnt + adfup ->{rt} adfup + adfup

| adfdn + bNotUp ->{rt} adfnt + bNotUp
| adfnt + reqfhi ->{rt} adfup + reqfhi
| adfup + bNotDown->{rt} adfnt + bNotDown
| adfnt + reqflo ->{rt} adfdn + reqflo

| adflo + adfup ->{rt} adfhi + adfup
| adfhi + adfdn ->{rt} adflo + adfdn

| adtdn + adtup ->{rt} adtnt + adtnt
| adtnt + adtdn ->{rt} adtdn + adtdn
| adtnt + adtup ->{rt} adtup + adtup

| adtdn + bNotUp ->{rt} adtnt + bNotUp
| adtnt + reqthi ->{rt} adtup + reqthi
| adtup + bNotDown->{rt} adtnt + bNotDown
| adtnt + reqtlo ->{rt} adtdn + reqtlo

| adtlo + adtup ->{rt} adthi + adtup
| adthi + adtdn ->{rt} adtlo + adtdn
```

```
//reactions
| adthi + ae ->{rt} adthi + ahi
| adfhi + ae ->{rt} adfhi + ahi

| adtlo + aw ->{rt} adtlo + aq
| adflo + aq ->{rt} adflo + alo


//for persistance

|alo + adthi ->{rt} aw + adthi
|alo + adfhi ->{rt} aq + adfhi

|alo + ahi ->{rt} alo + ae

| aiflo + aNotDown ->{rt} aiflo + aNotUp
| aifhi + aNotUp ->{rt} aifhi + aNotDown


//AREA 2
| bdfdn + bdfup ->{rt} bdfnt + bdfnt
| bdfnt + bdfdn ->{rt} bdfdn + bdfdn
| bdfnt + bdfup ->{rt} bdfup + bdfup

| bdfdn + cNotUp ->{rt} bdfnt + cNotUp
| bdfnt + adfhi ->{rt} bdfup + adfhi
| bdfup + cNotDown->{rt} bdfnt + cNotDown
| bdfnt + adflo ->{rt} bdfdn + adflo

| bdflo + bdfup ->{rt} bdfhi + bdfup
| bdfhi + bdfdn ->{rt} bdflo + bdfdn

| bdtdn + bdtup ->{rt} bdtnt + bdtnt
| bdtnt + bdtdn ->{rt} bdtdn + bdtdn
| bdtnt + bdtup ->{rt} bdtup + bdtup

| bdtdn + cNotUp ->{rt} bdtnt + cNotUp
| bdtnt + adthi ->{rt} bdtup + adthi
| bdtup + cNotDown->{rt} bdtnt + cNotDown
| bdtnt + adtlo ->{rt} bdtdn + adtlo

| bdflo + bdfup ->{rt} bdfhi + bdfup
| bdfhi + bdfdn ->{rt} bdflo + bdfdn

//reactions
| bdthi + be ->{rt} bdthi + bhi
| bdfhi + be ->{rt} bdfhi + bhi
```

```
| bdtlo + bw ->{rt} bdtlo + bq
| bdflo + bq ->{rt} bdflo + blo


//for persistance

|blo + bdthi ->{rt} bw + bdthi
|blo + bdfhi ->{rt} bq + bdfhi

|blo + bhi ->{rt} blo + be

| biflo + bNotDown ->{rt} biflo + bNotUp
| bifhi + bNotUp ->{rt} bifhi + bNotDown

//AREA 3
| cdfdn + cdfup ->{rt} cdfnt + cdfnt
| cdfnt + cdfdn ->{rt} cdfdn + cdfdn
| cdfnt + cdfup ->{rt} cdfup + cdfup

| cdfdn + cNotUp ->{rt} cdfnt + cNotUp
| cdfnt + bdfhi ->{rt} cdfup + bdfhi
| cdfup + cNotDown->{rt} cdfnt + cNotDown
| cdfnt + bdflo ->{rt} cdfdn + bdflo

| cdflo + cdfup ->{rt} cdfhi + cdfup
| cdfhi + cdfdn ->{rt} cdflo + cdfdn

| cdtdn + cdtup ->{rt} cdtnt + cdtnt
| cdtnt + cdtdn ->{rt} cdtdn + cdtdn
| cdtnt + cdtup ->{rt} cdtup + cdtup

| cdtdn + cNotUp ->{rt} cdtnt + cNotUp
| cdtnt + bdthi ->{rt} cdtup + bdthi
| cdtup + bNotDown->{rt} cdtnt + cNotDown
| cdtnt + bdtlo ->{rt} cdtdn + bdtlo

| cdflo + cdfup ->{rt} cdfhi + cdfup
| cdfhi + cdfdn ->{rt} cdflo + cdfdn

//reactions
| cdthi + ce ->{rt} cdthi + chi
| cdfhi + ce ->{rt} cdfhi + chi

| cdtlo + cw ->{rt} cdtlo + cq
| cdflo + cq ->{rt} cdflo + clo
```

```
//for persistance

|clo + cdthi ->{rt} cw + cdthi
|clo + cdfhi ->{rt} cq + cdfhi

|clo + chi ->{rt} clo + ce

| ciflo + cNotDown ->{rt} ciflo + cNotUp
| cifhi + cNotUp ->{rt} cifhi + cNotDown

| chi + reqhi ->{rt} chi + reqlo
| clo + reqlo ->{rt} clo + reqhi
```

# Bibliography


[1] David F Anderson and Thomas G Kurtz. Continuous time markov chain models for chemical reaction networks. In *Design and analysis of biomolecular circuits*, pages 3–42. Springer, 2011.

[2] Alexander Andreychenko, Thilo Krger, and David Spieler. Analyzing Oscillatory Behavior with Formal Methods. In Anne Remke and Marille Stoelinga, editors, *Stochastic Model Checking. Rigorous Dependability Analysis Using Model Checking Techniques for Stochastic Systems*, volume 8453 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 2014.

[3] Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.

[4] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[5] Charles H Bennett. The thermodynamics of computation - a review. *International Journal of Theoretical Physics*, 21(12):905–940, 1982.

[6] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94. ACM, 1989.

[7] Luca Cardelli. Invited talk: A process algebra master equation. In *null*, pages 219–226. IEEE, 2007.

[8] Luca Cardelli. On process rate semantics. *Theoretical Computer Science*, 391(3):190–215, 2008.

[9] Luca Cardelli. Two-domain dna strand displacement. *Mathematical Structures in Computer Science*, 23(02):247–271, 2013.

[10] Luca Cardelli. Morphisms of reaction networks that couple structure to function. *BMC systems biology*, 8(1):84, 2014.

[11] Luca Cardelli and Attila Csikász-Nagy. The cell cycle switch computes approximate majority. *Scientific reports*, 2, 2012.

[12] Luca Cardelli, Mirco Tribastone, Max Tschaikowski, and Andrea Vandin. Forward and backward bisimulations for chemical reaction networks.

[13] Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural computing*, 13(4):517–534, 2014.

[14] Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from dna. *Nature nanotechnology*, 8(10):755–762, 2013.

[15] Steven M Chirieleison, Peter B Allen, Zack B Simpson, Andrew D Ellington, and Xi Chen. Pattern transformation with dna circuits. *Nature chemistry*, 5(12):1000–1005, 2013.

[16] John H Conway. Fractran: A simple universal programming language for arithmetic. In *Open Problems in Communication and Computation*, pages 4–26. Springer, 1987.

[17] Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. In *Algorithmic Bioprocesses*, pages 543–584. Springer, 2009.

[18] Neil Dalchau, Georg Seelig, and Andrew Phillips. Computational design of reaction-diffusion patterns using dna-based chemical reaction networks. In *DNA Computing and Molecular Programming*, pages 84–99. Springer, 2014.

[19] Frits Dannenberg, Marta Kwiatkowska, Chris Thachuk, and Andrew J Turberfield. Dna walker circuits: Computational potential, design, and verification. In *DNA Computing and Molecular Programming*, pages 31–45. Springer, 2013.

[20] A Prasanna de Silva and Nathan D McClenaghan. Molecular-scale logic gates. *Chemistry-A European Journal*, 10(3):574–586, 2004.

[21] Péter Érdi and János Tóth. *Mathematical models of chemical reactions: theory and applications of deterministic and stochastic models*. Manchester University Press, 1989.

[22] Javier Esparza and Mogens Nielsen. *Decidability issues for Petri nets*. BRICS, Computer Science Department, University of Aarhus, 1994.

[23] Daniel T Gillespie. Exact stochastic simulation of coupled chemical reactions. *The journal of physical chemistry*, 81(25):2340–2361, 1977.

[24] Leon Glass and Michael C Mackey. A simple model for phase locking of biological oscillators. *Journal of Mathematical Biology*, 7(4):339–352, 1979.

[25] Nicholas J Guido, Xiao Wang, David Adalsteinsson, David McMillen, Jeff Hasty, Charles R Cantor, Timothy C Elston, and JJ Collins. A bottom-up approach to gene regulation. *Nature*, 439(7078):856–860, 2006.

[26] Purnananda Guptasarma. Does replication-induced transcription regulate synthesis of the myriad low copy number proteins of escherichia coli? *Bioessays*, 17(11):987–997, 1995.

[27] Holger Hermanns. *Interactive Markov Chains: And the Quest for Quantified Quality*. Springer-Verlag, Berlin, Heidelberg, 2002.

[28] Hua Jiang, Marc D Riedel, and Keshab K Parhi. Digital signal processing with molecular reactions. *IEEE Design and Test of Computers*, 29(3):21–31, 2012.

[29] Richard M Karp and Raymond E Miller. Parallel program schemata. *Journal of Computer and system Sciences*, 3(2):147–195, 1969.

[30] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In *Computer performance evaluation: modelling techniques and tools*, pages 200–204. Springer, 2002.

[31] Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In *Formal methods for performance evaluation*, pages 220–270. Springer, 2007.

[32] Matthew R Lakin, David Parker, Luca Cardelli, Marta Kwiatkowska, and Andrew Phillips. Design and analysis of dna strand displacement devices using probabilistic model checking. *Journal of the Royal Society Interface*, page rsif20110800, 2012.

[33] Matthew R Lakin and Andrew Phillips. Modelling, simulating and verifying turing-powerful strand displacement systems. In *DNA Computing and Molecular Programming*, pages 130–144. Springer, 2011.

[34] Matthew R Lakin, Simon Youssef, Filippo Polo, Stephen Emmott, and Andrew Phillips. Visual dsd: a design and analysis tool for dna strand displacement systems. *Bioinformatics*, 27(22):3211–3213, 2011.

[35] Luca Laurenti, Luca Cardelli, and Marta Z. Kwiatkowska. Stochastic analysis of chemical reaction networks using linear noise approximation. *CoRR*, abs/1506.07861, 2015.

[36] Anthony ML Liekens and Chrisantha T Fernando. Turing complete catalytic particle computers. In *Advances in Artificial Life*, pages 1202–1211. Springer, 2007.

[37] Marcelo O Magnasco. Chemical kinetics is turing universal. *Physical Review Letters*, 78(6):1190, 1997.

[38] Rajit Manohar and Alain J Martin. Quasi-delay-insensitive circuits are turing-complete. Technical report, DTIC Document, 1995.

[39] Marvin L Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967.

[40] Kevin Montagne, Raphael Plasson, Yasuyuki Sakai, Teruo Fujii, and Yannick Rondelez. Programming an in vitro dna oscillator using a molecular networking strategy. *Molecular systems biology*, 7(1):466, 2011.

[41] Nam-Phuong D Nguyen, Hiroyuki Kuwahara, Chris J Myers, and James P Keener. The design of a genetic muller c-element. In *Asynchronous Circuits and Systems, 2007. ASYNC 2007. 13th IEEE International Symposium on*, pages 95–104. IEEE, 2007.

[42] G Oster and A Perelson. Chemical reaction networks. *Circuits and Systems, IEEE Transactions on*, 21(6):709–721, 1974.

[43] Gheorghe Păun and Grzegorz Rozenberg. A guide to membrane computing. *Theoretical Computer Science*, 287(1):73–100, 2002.

[44] M Pedersen and A Phillips. Gec tool. *See http://research. microsoft. com/gec*, 2009.

[45] Marc Renaudin. Asynchronous circuits and systems: a promising design alternative. *Microelectronic engineering*, 54(1):133–149, 2000.

[46] Paul WK Rothemund. A dna and restriction enzyme implementation of turing machines. *DNA based computers*, 27:75–119, 1996.

[47] Paul WK Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of dna sierpinski triangles. *PLoS biology*, 2(12):e424, 2004.

[48] Georg Seelig, David Soloveichik, David Yu Zhang, and Erik Winfree. Enzyme-free nucleic acid logic circuits. *science*, 314(5805):1585–1588, 2006.

[49] Phillip Senum and Marc Riedel. Rate-independent constructs for chemical computation. *PloS one*, 6(6):e21414, 2011.

[50] Adam Shea, Brian Fett, Marc D Riedel, and Keshab K Parhi. Writing and compiling code into biochemistry. In *Pacific Symposium on Biocomputing*, volume 15, pages 456–464. World Scientific, 2010.

[51] Seung Woo Shin. *Compiling and verifying DNA-based chemical reaction network implementations*. PhD thesis, California Institute of Technology, 2011.

[52] Zack Booth Simpson, Timothy L Tsai, Nam Nguyen, Xi Chen, and Andrew D Ellington. Modelling amorphous computations with transcription networks. *Journal of The Royal Society Interface*, page rsif20090014, 2009.

[53] David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *natural computing*, 7(4):615–633, 2008.

[54] David Soloveichik, Georg Seelig, and Erik Winfree. Dna as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010.

[55] Jens Spars and Steve Furber. *Principles Asynchronous Circuit Design*. Springer, 2002.

[56] David Sprinzak and Michael B Elowitz. Reconstruction of genetic circuits. *Nature*, 438(7067):443–448, 2005.

[57] Milan N Stojanovic, Tiffany Elizabeth Mitchell, and Darko Stefanovic. Deoxyribozyme-based logic gates. *Journal of the American Chemical Society*, 124(14):3555–3561, 2002.

[58] Ivan E Sutherland and Jo Ebergen. Computers without clocks-asynchronous chips improve computer performance by letting each circuit run as fast as it can. *Scientific American*, 287(2):62–69, 2002.

[59] Atsuko Takamatsu, Reiko Tanaka, Hiroyasu Yamada, Toshiyuki Nakagaki, Teruo Fujii, and Isao Endo. Spatiotemporal symmetry in rings of coupled biological oscillators of physarum plasmodial slime mold. *Physical Review Letters*, 87(7):078102, 2001.

[60] Oleg N Temkin, Andrew V Zeigarnik, and DG Bonchev. *Chemical reaction networks: a graph-theoretical approach*. CRC Press, 1996.

[61] John E Turner and David L Rutledge. Programmable logic device, August 2 1988. US Patent 4,761,768.

[62] Gianluigi Zavattaro and Luca Cardelli. Termination problems in chemical kinetics. In *CONCUR 2008-Concurrency Theory*, pages 477–491. Springer, 2008.

[63] David Yu Zhang, Rizal F Hariadi, Harry MT Choi, and Erik Winfree. Integrating dna strand-displacement circuitry with dna tile self-assembly. *Nature communications*, 4, 2013.