Modelling, Realisations and Limitations of Concurrent Delay-Insensitive Networks

Thesis submitted in accordance with the requirements of the Department of Informatics, University of Leicester for the degree of Doctor in Philosophy

by

Daniel Morrison

February 2017

Modelling, Realisations and Limitations of Concurrent Delay-Insensitive Networks

Daniel Morrison

Abstract

Concurrent and distributed behaviour encompasses a wide range of ever evolving phenomena and features of computation such as communication, mobility, causality, failure recovery and reversibility. In order to understand better and make precise the properties of such behaviour, concurrent and distributed behaviour needs to be modelled abstractly and with formal rigour.

Delay-insensitive networks are a class of asynchronous systems which makes no assumptions about the timing of signals or components. This makes them suitable for the implementation of highly concurrent systems. Unfortunately, concurrency within delay-insensitive networks is an underdeveloped concept which lacks formal rigour. Reversibility of such systems is also typically only studied in the context of serial systems without concurrency.

In this thesis, a new model for describing the behaviour of delay-insensitive components is introduced which more naturally permits the modelling and study of concurrency. Reversibility of such components is discussed. The concept of an environment for a component is formalised, and its limitations in terms of interactivity with such a component is also studied. Algorithms for generating the environments for delay-insensitive components, such that desirable properties always hold are given. Universality results and properties of networks of such components are examined in-depth.

A new process algebra which allows the encoding of these networks is introduced. This permits rigorously defined notions of implementation and other desirable runtime properties of these networks.

A family of new novel cellular automata is defined which allows the encoding of delay-insensitive networks. These cellular automata are competitive with existing CA regarding universality, and number of rules and states. They have a feature we call direction-reversibility, which allows the inversion of behaviour simply by reversing the direction of signals.

Finally, two pieces of software called *Delay-Insensitive Network Tool Suite* and *STCA Simulator*, developed to aid in this research, are also detailed.

Acknowledgements

I would like to thank my supervisor Irek Ulidowski, who has guided me through both my MComp degree and my PhD studies.

My PhD studies were funded by a PhD studentship from the College of Science and Engineering PhD Studentship Scheme and by the ESPRC via the ESPRC Doctoral Training Award (DTA). I thank both the College of Science and Engineering at the University of Leicester and the ESPRC.

Much of the material in this thesis was published in the RC 2013 [52], RC 2014 [53] and ACRI 2014 [54] conference proceedings, and the Journal of Cellular Automata [55]. I thank the reviewers for providing invaluable feedback.

I would like to thank my parents for providing for me. This has given me the opportunity to pursue a career in Computer Science.

Finally, I thank my partner Alexandra. She has supported me throughout all of my studies over the last 6 years, and has given me a reason to work hard and achieve all that I can.

Contents

List of Figures

Chapter 1

Introduction

We begin with a brief motivation for the work in this thesis, as well as an overview of its structure and results.

1.1 Motivation

Modern computers are reaching limits in terms of the speed which can be achieved using a single thread of execution. This has forced CPU designers to adopt a design strategy that involves concurrency, with general purpose CPUs now possessing multiple cores, allowing them to execute many threads in parallel. Similarly, graphical processing units (GPUs) are inherently highly parallel by nature, often executing many thousands of threads concurrently. As a result, it is important to study general theoretical models of concurrency.

Asynchrony is similarly important for study. It is believed by some [13, 39] that a move to an asynchronous paradigm for general-purpose computing is inevitable. Nevertheless, asynchrony is already present in some existing technologies [20, 78]. It has numerous advantages over synchronous forms of computing, such as reduced energy consumption due to both the absence of a global clock and inactive components not consuming power, and a removal of "worst-case" timing assumptions due to each component being self-timed [75]. Reversible circuit elements are also energy efficient for different reasons [3, 12] as operations result in no loss of information, and it has been shown in [26] that irreversible operations necessarily result in the dissipation of energy. Combining concurrency, asynchrony, and reversibility is therefore desirable.

Studying these types of models allows us to understand the limitations, practicality, and complexity of concurrent, reversible or asynchronous systems. Methods for verification of these types of systems can be studied and developed, ensuring correctness of implementations. Inferring the resulting behaviour arising from some starting specification is also possible. A general enough model can be independent

of any particular technology or implementation, and can therefore be useful for any potential future technology which employs concurrency, reversibility or asynchrony. An example of a category of models for studying concurrent behaviour is *Process Calculi* [17, 42], of which reversible versions exist [4, 7].

Delay-insensitive (*DI*) networks are a category of asynchronous networks which makes no assumption about delays within elements or wires. As a type of asynchronous circuit, they have no global clock. It is shown in [40] that it is not possible to realise the full Turing-complete class of digital circuits with typical logic gates such as NAND and XOR while requiring delay-insensitivity. Hence the typical DI model of study, introduced by Keller [23], is more abstract and uses different types of modules. These operate based upon the presence or absence of signals rather than the values of wires like in the typical boolean circuit model. As the need for a replacement technology for CMOS arises, DI networks are seen as a possible future direction for the industry. Their implementation in several alternative technologies such as cellular automata [29] and RSFQ circuits [63] is considered a potential option. Implementing these modules in CMOS is possible, but requires internal timing constraints [23] due to the aforementioned impossibility of logic gates to provide sufficiently expressive delay-insensitivity at all levels of abstraction. Moreover, as DI networks represent the complete absence of any timing conditions, correctness of an implementation when assuming a DI operating environment guarantees correctness under any model with stricter timing conditions.

Ensuring a rigorous well-developed model for DI systems is therefore critical. However the exact set of conditions which are considered essential for correct operation of DI modules and networks varies between publications. The result is that there are subtle differences between the models used and results which hold in one model do not necessarily hold in another. A lack of consistent notation for describing module behaviour (examples found across [23, 30, 60]) further compounds this issue and makes the identification of behavioural properties difficult. Research into combining reversibility and delay-insensitive networks [36, 45, 49] has been carried out, in the context of single-signalled networks. However, a comparatively small amount of research has been carried out into reversibility together with concurrency in the setting of DI networks.

Cellular automata (*CA*) [6] are a category of computational models which naturally implement concurrency. Variations which implement notions of reversibility [22] and asynchrony [11], as well as combinations of the two [37] also exist. They are considered an effective computational model for potential future nano-technologies [66]. The difficulty of performing useful computation using a cellular automata-based model has led to a concentrated effort towards encoding DI networks in a subcategory of cellular automata, known as *Self-Timed Cellular Automata* (*STCA*)

[65]. Research has focused on minimising the number of transition rules in cellular automata, as well as ensuring that various notions of reversibility hold. Combining concurrency, reversibility and DI networks in the setting of STCA is a relatively unexplored concept.

1.2 Contributions

When conducting the research for this thesis, we set out to define a new, more robust model for abstract delay-insensitive networks, which exhibits truly concurrent behaviour at the fundamental level. This model would allow us to define reversibility of concurrent modules, as well as a clear distinction between deterministic and non-deterministic behaviour. We wished to also formalise the notion of an environment for a given module or network. This would allow us to study and define clear notions of universality and implementation.

We set out to introduce a new process algebra for the modelling and formal representation of systems defined using such a newly developed model, and further formalise behaviour and important properties (such as implementation) in the context of this algebra. Such systems would then be able to have their behaviours inferred based on the rules of this algebra. Formalising properties and behaviour in this algebra would also help prevent the ambiguity of properties and conditions which is commonly found in existing DI literature.

We also wanted to investigate how to implement concurrent delay-insensitive networks in cellular automata, and see if any newly developed notions of reversibility could be exploited in order to minimise the complexity of a cellular automaton's transition rules, or constructions within such cellular automata.

The above concepts were intended to be further explored and studied via the development of dedicated software tools.

Many results in this thesis appeared in publications co-authored by Morrison during the period of registration. This includes the works [52, 53, 54, 55]. Irek Ulidowski is recognised as a co-author for these publications, and has provided valuable feedback for the results which are present in each of these publications in addition to this thesis. More generally, the reviewers of the RC 2013, RC 2014 and ACRI 2014 conferences, and the Journal of Cellular Automata are thanked for their comments.

1.3 Overview of thesis

We now detail the content and main results found in each chapter:

2: Background to DI networks and the sequential machine model: We give an overview of the existing model for DI networks formulated by Keller, which we refer to as the *sequential machine* model. We include all operating conditions and relevant definitions. The general construction method for any module in the sequential machine model is detailed in depth. Further research in the field of DI networks, including work related to reversibility, is also discussed. Several existing universality results are detailed. We introduce our own reversible memory modules in the sequential machine model, and we use these modules to infer some simple universality results. We finish by discussing what we see as the main shortcomings of the sequential machine model for DI networks. Motivation is given for the development of a new DI model which implements concurrency more directly.

3: The Set Notation model: We introduce a new model for describing the behaviour of delay-insensitive modules, called *Set Notation* which more naturally models concurrency and notions of reversibility. We define important classes of modules in the Set Notation model, such as the non-arb, eq-arb and arb classes. We define networks of modules in the Set Notation model, along with the execution behaviour of such networks. We define properties of such networks, such as the non-clashing and safety properties. We investigate several further properties of modules in the Set Notation model which limit the behaviour of the environment in unexpected ways. Examples include the auto-firing or 1-step consistency properties of modules.

4: Environments and Implementation: We formalise the notion of an environment for a module in the Set Notation model. We give an algorithm for calculating what is referred to as a *maximal environment* of any non-arb module. An algorithm for calculating a maximal environment of any module is then given. Finally, we use this notion to define implementation of a module using a network of modules.

5: Correspondences between models: We compare the sequential machine model for DI networks with the new Set Notation model. We introduce an extension to the sequential machine model called the *ND sequential machine* model. We establish limited correspondences between three models; the sequential machine model, the ND sequential machine model, and the new Set Notation model. We prove universality results for the ND sequential machine model. We give algorithms for converting modules which satisfy certain conditions in the sequential machine model or the ND sequential machine model, to corresponding definitions in the Set Notation model, and vice versa.

6: Universality and implementing modules using concurrency: We inves-

tigate inversion of networks of modules. We give some universality results for serial modules in the Set Notation model. We then give a series of detailed construction methods and universal sets for the non-arb and eq-arb classes of modules. We also prove a universal set for all Set Notation modules, by utilising a correspondence between the ND sequential machine and Set Notation models defined in the previous chapter. We demonstrate an interesting property inherent to networks of concurrent DI modules, which is that bijectivity of all modules' transition maps can still result in useful irreversible behaviour at the global level. We compare this with a similar but less general result in the literature.

7: DI-Set algebra for DI networks: We introduce a new process algebra, called *DI-Set algebra*, which is intended to model the behaviour of networks from the Set Notation model. We give examples of encoding modules (such as *Merge*) and networks (such as a network that implements *Merge*) in DI-Set algebra, and define properties of networks discussed in previous chapters (such as safety and non-clashing) more formally in the context of DI-Set algebra. We investigate the use of bisimulation and simulation, and use these together with the aforementioned properties to define more formally the concept of implementation of a module using a network of modules.

8: Background to STCA: We give an introduction to the concepts related to *Self-Timed Cellular Automata*, including the definitions of important properties such as *local reversibility* and *local determinism*. We define new versions of *global reversibility* and *global determinism*, which are related to similar properties in the literature. However these new versions are more flexible properties which are based on notions of convergence, and are appropriate for STCAs which simulate concurrent DI networks. We also give an example of an existing STCA from the literature.

9: Implementation in STCA and direction-reversibility: We introduce four novel STCAs for implementing DI networks, including two STCAs for reversible serial and non-arb non-b-arb networks. The two main STCAs have several very useful properties, they are locally deterministic, locally reversible and support what we call direction-reversibility. This allows us to operate a network in reverse by changing the direction of signals and utilising its output lines as input lines (and vice versa). This removes the need for separate constructions to implement the inverse of a network. The new notions of global determinism and global reversibility are proven to hold for these two STCA. We also introduce two further extensions to the STCAs which simulate irreversible serial and non-arb b-arb networks. These two additional STCAs are shown to be locally deterministic and globally deterministic. Finally, we

prove that the third and fourth STCAs can be used to implement any module in either Set Notation or the ND sequential machine model.

10: Software tools: We detail the two pieces of software developed in support of this thesis. The first, called *Delay-Insensitive Network Tool Suite* contains implementations of the maximal environment generation algorithms in Chapter 4, conversion algorithms in Chapter 5 and algorithms which follow the non-arb and eq-arb construction methods in Chapter 6. It also implements a version of the DI-Set algebra in Chapter 7 with interactive execution and LTS generation with property and bisimulation/simulation checking. The second piece of software, called *STCA Simulator*, implements the four direction-reversible STCA in Chapter 9 with an interactive graphical interface. It also contains the constructions from that chapter. A brief description of how important features are implemented is given for each piece of software.

11: Conclusion and future research: We summarise the work achieved in this thesis and outline possible future directions of research.

Chapter 2

Background to DI networks and the sequential machine model

In this chapter we give an overview of the existing model for DI networks formulated by Keller in [23], which we refer to as the *sequential machine* model. We include all operating conditions and relevant definitions. The general construction method for any module in the sequential machine model is detailed in-depth. Further research in the field of DI networks, including work related to reversibility, is also discussed. Several existing universality results are detailed. We introduce our own reversible memory modules in the sequential machine model, and we use these modules to infer some simple universality results. We finish by discussing what we see as the main shortcomings of the sequential machine model for DI networks. Motivation is given for the development of a new DI model which implements concurrency more directly.

The contents of Section 2.3 with the exception of Corollary 2.41 were published in [52].

2.1 Sequential machine model by Keller

We begin by outlining the original model by Keller given in [23], which we will refer to as the *sequential machine* model. As in [23], let $P[X]$ represent the powerset (the set of all subsets) of the set X.

Definition 2.1. ([23], Definition 1.1) A sequential machine is a 6-tuple $N = (Q, q_o, \Sigma, \Delta, f, g)$, where:

1. Q is a finite set of states,
2. $q_o \in Q$ is the initial state,
3. Σ is the input alphabet,

4. Δ is the output alphabet,
5. $f : Q \times \Sigma \to Q$ is a partial function, the state-transition function,
6. $g : Q \times \Sigma \to \Delta$ is a partial function, the output function.

Definition 2.2. ([23], Definition 1.2) A module is a 4-tuple (I, O, N, A) where:

1. I is a finite set of input lines,
2. O is a finite set of output lines,
3. $N = (Q, q_o, \Sigma, \Delta, f, g)$ is a sequential machine with Σ in one-to-one correspondence with I and Δ in one-to-one correspondence with $P[O]$,
4. $A : Q \to P[P[I]]$ is a function which specifies for each state the combination of inputs which can occur.

The execution behaviour of such modules is given in prose in [23].

Definition 2.3. ([23]) An element of Σ represents the occurrence of a (one-valued) "signal" on the corresponding element of I. Similarly, an element of Δ represents the occurrence of a signal on the corresponding elements of O. Signals are placed on the input lines of a module m by other modules external to m. In turn, m "assimilates" these signals by possibly changing state and creating signals on its output lines, according to the specification of its machine N.

We also make note of the following description, given informally in [23], of the A function.

Definition 2.4. ([23]) A **function description**: For any internal state q of a module, $A(q) \subseteq P[I]$ represents the allowable input sets; i.e. if $S \in A(q)$ then any subset of the lines corresponding to elements of S are allowed to be signalled concurrently.

There is also the following assumption, given informally in [23], which for convenience we will name and refer to as the *safety assumption*.

Definition 2.5. ([23]) **safety assumption**: When the state-transition function or output function is undefined for some particular state and input combinations, it is assumed that this combination will never occur in actual operation.

Observation 2.6. It is not made explicit in [23] that for all q, a, if $f(q, a)$ is defined, then $g(q, a)$ is defined and vice versa. However, it does not make sense for this not to be the case. Similarly, it is not made explicit that for all q, a, $f(q, a)$ is defined iff there exists some set L such that $a \in L \in A(q)$. However this is implied by the existence of such sets in all modules defined by Keller in [23]. In the rest of this thesis, we assume that these two conditions always hold (as in Definition 2.7).

In the rest of this thesis, when referring to the sequential machine model for delay-insensitive networks, we do not make a distinction between a module and its internal sequential machine: instead we combine them into a single definition. This reduces the complexity of definitions and improves readability, without affecting the model's generality. This is similar to the approach used in other publications [33, 45, 48, 60]. We therefore define a module as follows.

Definition 2.7. A module is defined by the 6-tuple (Q, I, O, f, g, A) where:

1. Q is a finite set of states,
2. I is a set of input lines,
3. O is a set of output lines,
4. $f : Q \times I \to Q$ is a partial function, the state-transition function,
5. $g : Q \times I \to P[O]$ is a partial function, the output function,
6. $A : Q \to P[P[I]]$.

We require that for all q, a, $f(q, a)$ is defined iff $g(q, a)$ defined. Finally, for all q, a, $f(q, a)$ is defined iff there exists some L such that $a \in L \in A(q)$.

Note that we do not maintain the notion of an "initial" state. This is because in practice, modules are typically initialised in various states, depending on the intended operation of the network. As a result, it is more convenient to simply indicate the current state of a module in each given network, and keep the definition of a module's operation independent of any assumptions about the state in which it is initialised. All conditions and restrictions given by Keller regarding sequential machines can be assumed to refer to our notion of a module.

We introduce some useful notation. q represents a state, and $a, b, c \ldots$ and $B, C, D \ldots$ range over input/output lines and sets of such lines respectively. We also utilise subscripts, superscripts and prime symbols to increase the set of variables at our disposal. Alphanumeric symbols may also be names of literal state names and input/output lines. It will be clear from the context whether these refer to variables or constants.

We use a CCS-like [42] notation to succinctly define a module.

Definition 2.8. If $f(q,a)$ and $g(q,a)$ are defined, then $(a,g(q,a)).f(q,a)$ is called an *action* of q, where $(a,g(q,a))$ is an input/output pair and $f(q,a)$ is the resulting state.

We specify all actions of q by writing $q = (a_1,B_1).q_1 + \ldots + (a_n,B_n).q_n$ where $B_x = g(q,a_x), q_x = f(q,a_x)$ and $f(q,a_x)$ and $g(q,a_x)$ are defined for all $1 \leq x \leq n$. Then the definition of a module N is given by a set of such equations, one for each state of N, together with a definition of the A function. Input and output lines are implicit.

We require that for all pairs q,q': if $f(q,a) = f(q',a)$ and $g(q,a) = g(q',a)$ for all a, then $q = q'$ (no two different states have the same CCS-like definitions). Sometimes we write $(a,B).q' \in q$ to mean $(a,B).q'$ is an action of q.

Networks of modules are defined as follows.

Definition 2.9. ([23], Definition 1.3) A network is a collection of modules with some of their lines interconnected. If an input line of a module is unconnected, then it is an *input line* to the network. If an output line of a module is unconnected then it is an *output line* of the network.

We outline the main operating conditions of modules and networks after Keller by listing them directly. Each Condition is taken directly from [23].

- **Condition 1**: I and O are disjoint.
- **Condition 2**: A module, once having created a signal on a line, cannot "withdraw" the signal before it is assimilated by a module on the opposite end of the line.
- **Condition 3-(Arbitration Condition)**: If two signals appear on different input lines of a module simultaneously, or very close together in time, the action of the module should be as if one signal, then the other, occurred as specified by the sequential machine. If the action depends on the order of occurrence, then the action may be chosen arbitrarily by the module. Hence a module may assimilate signals in one order, even if the actual order of occurrence is just the opposite.
- **Condition 4**: There may be an arbitrary delay between the assimilation of an input signal by a module and the production of a corresponding output signal. This delay is always finite but is not necessarily bounded.
- **Condition 5**: At most two modules in a network are ever connected by the same line, and this line must be an input to one module and an output from the other.

- **Condition 6**: If a signal is produced by one module on an input line to another module, it must be assimilated before a second signal occurs on the same line.

- **Condition 7**: A line interconnecting two modules has no intrinsic delay.

- **Condition 8**: Two successive signals can be placed on an input line of m only if they are interspersed by at least one output signal from m, which occurs in response to the input signal. (Otherwise the modules external to m would have no way of knowing when a signal had been assimilated, and would tend to violate Condition 6.)

- **Condition 9**: Two signals can be simultaneously placed on different input lines of m only if the outputs that occur in response to these signals individually are produced on disjoint sets of output lines. (This is because if the output signals were placed on the same line, Condition 6 would be violated for some module external to m.)

- **Condition 10**: Two signals which could occur successively on (the same or different) input lines and produce signals on overlapping sets of lines S must be such the latter input occurs after the occurrence of all signals on S which are due to the former.

- **Condition 11-(Finite-blocking condition)**: If a signal is present on an input line to a module, this signal must eventually be assimilated.

According to Keller in [23] the above conditions are motivated by a desire to develop a model which is flexible enough to be independent of the details of any particular hardware implementation. If we were to relax Condition 2, for example, this would result in a model which suggests a sophisticated mechanism by which modules have a great deal of control over their environment. This could potentially reduce the applicability of the model to any "simple" or low-level technologies, as well as any as-yet developed future technologies. We note informally that while the model intentionally has no explicit notion of time, there is the informal assumption (expressed directly by Conditions 4 and 11) that events (e.g. travelling of signals along wires, absorption of input signals) may not be indefinitely delayed, as this would also reduce the usefulness of the model.

Keller informally states the following restriction on the sequential machine.

Definition 2.10. ([23]) **sequential machine restriction**: Condition 3 implies the following restriction on the sequential machine: If $f(q, \sigma)$ and $f(q, \pi)$ are both defined, where $\sigma \neq \pi$, then so are $f(f(q, \sigma), \pi)$ and $f(f(q, \pi), \sigma)$.

We note that this is only required if there exists some $L \in A(q)$ such that $\sigma \in L$ and $\pi \in L$ (i.e. the signals are allowed to arrive concurrently), but this is omitted by Keller. In the rest of this thesis, we assume that this is the case.

We give Keller's definitions of speed-independence and delay-insensitivity.

Definition 2.11. ([23], Definition 1.4) A network is called *speed-independent* if its external behaviour is independent of the delay of the constituent modules. (The possibility of arbitration, as in Condition 3, is permitted).

Definition 2.12. ([23], Definition 1.5) A *delay element* is a module with one input line, one output line, and one state. It functions only to assimilate signals on its input and reproduce them on its output.

Definition 2.13. ([23], Definition 1.6) A network is called *delay-insensitive* if its external behaviour remains unchanged, regardless of whether any number of delay elements are inserted into, or removed from any lines.

Note that trivially, if a network is delay-insensitive, then it is also speed-independent. In this thesis, we are only interested in delay-insensitive networks. Hence, we simplify the model without loss of generality by removing the need for delay elements and modify Condition 7 to the following.

- **New Condition 7**: A line has an unbounded but finite delay.

In the rest of this thesis, when referring to the sequential machine model for DI networks, we assume that Conditions 1-6, New Condition 7, and Conditions 8-11 are enforced unless otherwise stated.

Keller gives the following definitions of a realisation and universality.

Definition 2.14. ([23], Definition 1.7) A realisation of a module m is a speed-independent delay-insensitive network of modules which has the same external behaviour as m. (A realisation always implies some specific internal initial state.)

Definition 2.15. ([23], Definition 1.8) Let E be a class of modules and M a set of modules types. M is called universal for E if every module in E may be realised (in the sense of Definition 2.14) by a network consisting only of module types in M.

We note that the term *speed-independent* is redundant in Definition 2.14.

We note that the definitions of delay-insensitivity, realisation and external behaviour are given somewhat informally. One of the outcomes of this thesis is an attempt to define these concepts more formally under the new model presented in Chapters 3-7.

Modules defined using the sequential machine model can be divided into two distinct classes. The first of which is the class of *serial* modules.

Definition 2.16. ([23], Definition 1.9) A module is called *serial* if it must operate under the condition that every input signal, regardless of upon which input line it occurs, must be followed by exactly one output signal on some line before another input can be applied. Note that this is a proper strengthening of Conditions 2 and 3.

We note that Definition 2.16 implies that for all states q of a serial module, $A(q)$ contains only singletons, and for all actions $(a, B).q'$ of q, B is a singleton.

The second type is the class of *parallel* modules.

Definition 2.17. ([23], Definition 3.1) A module is called *parallel* if it allows more than one signal on different inputs which are not necessarily separated by an output signal, or if it may produce more than one output signal on different lines due to a single input. (Conditions 2 and 6 are still to be observed however. Hence it is always assumed, in the case of parallel modules, that the sequential machine is specified in a manner consistent with Conditions 8-10).

Note that Definition 2.17 implies that for all actions $(a, B).q'$ of all states q of a parallel module, B may be a set of any size, including the empty set $\emptyset$. Furthermore, this implies that a module is parallel if it is not serial.

Definition 2.18. We say that a set of modules is *universal for the class of sequential machines* if it is universal for both the class of serial modules and the class of parallel modules.

Definition 2.18 is used to distinguish universality within the sequential machine model from alternative models presented in Chapters 3 and 5.

We note a useful measure of complexity of sets of modules introduced by Keller in [23].

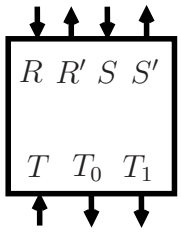
Definition 2.19. ([23], Definition 2.6) A set of modules is said to have *modularity* n if n is the maximum number of lines on any one module in the set.

In Figure 2.1 we define several useful modules from [23]. We rename the states, input and output lines of these modules for convenience. All symbols shown are constants and should not be confused with the variables used in this thesis.

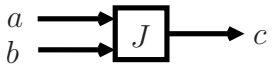
Example 2.20. *Join* is a three-state module in which signals on a and b may arrive concurrently in state S_0, as given by $A(S_0)$. After processing an input signal on both a and b, the module produces a single output signal on c.

Merge is a serial module which accepts an input signal on either a or b (but not both concurrently) and produces an output signal on c.

Fork produces a pair of output signals (one on b and c each) in response to a single input signal on a.



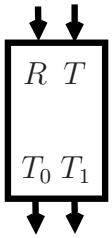
$$
\begin{aligned}
S_0 &= (T,\{T_0\}).S_0 + (S,\{S'\}).S_1 \\
&+ (R,\{R'\}).S_0 \\
S_1 &= (T,\{T_1\}).S_1 + (S,\{S'\}).S_1 \\
&+ (R,\{R'\}).S_0
\end{aligned}
$$



$$
\begin{aligned}
S_0 &= (a,\{\}).S_a + (b,\{\}).S_b \\
S_a &= (b,\{c\}).S_0 \\
S_b &= (a,\{c\}).S_0 \\
A(S_0) &= \{\{a,b\}\} \\
A(S_a) &= \{\{b\}\} \\
A(S_b) &= \{\{a\}\}
\end{aligned}
$$

$$M = (a,\{c\}).M + (b,\{c\}).M$$

$$
\begin{aligned}
F &= (a,\{b,c\}).F \\
A(F) &= \{\{a\}\}
\end{aligned}
$$



$$
\begin{aligned}
S_1 &= (T,\{T_1\}).S_1 + (R,\{\}).S_0 \\
S_0 &= (T,\{T_0\}).S_1 \\
A(S_1) &= \{\{T,R\}\} \\
A(S_0) &= \{\{T\}\}
\end{aligned}
$$

Figure 2.1: From top-left: *Select*, *Join*, *Merge*, *Fork*, *ATS* [23].

Example 2.21. *Select* is a serial module which acts as a one-bit memory. The states S_0 and S_1 correspond to each possible value. An input signal on T "queries" the memory value, and produces an output signal on T_0 or T_1 depending on whether the state is S_0 or S_1 respectively. An input signal on R sets the state of the module to S_0 regardless of the current state, and produces an output signal on R'. Similarly for S, S' and S_1.

We do not depict the function A for serial modules, as for all states q, the set $A(q)$ always contains a singleton $\{a\}$ for each input line a which is in an action of q. We also do not depict the labels of the input or output lines of *Join*, *Merge* or *Fork* in diagrams, as the choice of particular input or output lines when connecting these modules does not affect the behaviour of the network.

Example 2.22. *ATS* is a two-state module, which can hold one of two memory values (referred to as 0 and 1), and allows the arrival of inputs signals on T and R concurrently. If a signal on T is processed, the module outputs a signal on T_1 or T_0, corresponding to the held value, and resets the held value of the module to 1. If a signal on R is processed, the module sets the held value to 0 but produces no output signal, so this is not visible to the environment. Hence to avoid violating Keller's Condition 6, the environment cannot send an input signal on R twice without receiving an output signal on T_0 in between, but may send an input signal on T in between receiving any pair of output signals.

We note that it is possible to create arbitrary n-way *Join*, *Merge* and *Fork* trees by connecting multiple instances of the same module together in a tree formation. We will depict such trees using the same symbols as the standard modules shown in Figure 2.1, with only the number of input lines differing in the case of *Join* and *Merge* trees, and the number of output lines differing in the case of *Fork* trees.

Example 2.23. Figure 2.2 shows how a 4-way *Join* tree and a 4-way *Fork* tree can be realised using *Join*s and *Fork*s respectively, along with the symbols we use to depict them.

We note two important universality results proven by Keller in [23].

Theorem 2.24. ([23], Theorem 2.1) The set {*Merge, Select*} is universal for the class of serial modules.

Proof. The proof, given in [23], is achieved by showing how to realise modules *Call* and *D-call* (which we do not define here) using only *Merge* and *Select*. A general construction method for realising any serial module, using *Call* and *D-call* modules, is then shown. □

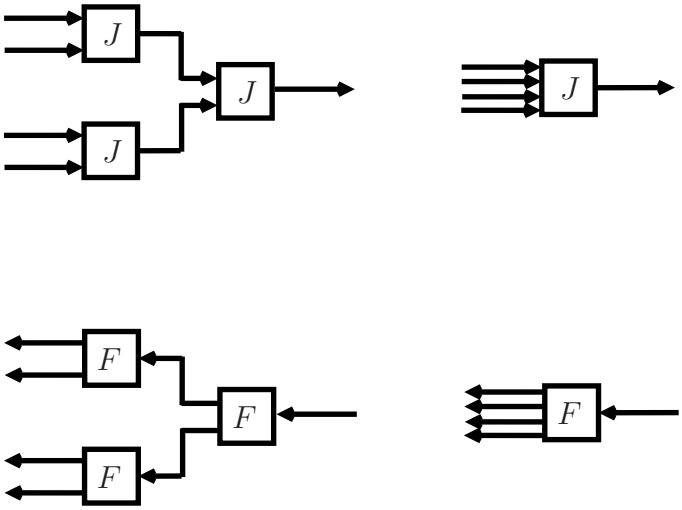


Figure 2.2: (Top) 4-way *Join* tree and the symbol used to commonly depict it. A single signal on each of the four input lines to the network will eventually produce a signal on the output line from the network. (Bottom) 4-way *Fork* tree and the symbol used to commonly depict it. A single signal on the input line to the network will eventually produce signals on each of the four output lines from the network.

Theorem 2.25. ([23], Theorem 3.2) The set {*Merge, Select, Fork, ATS*} is universal for the class of sequential machines.

Proof. Universality of serial modules is implied by Theorem 2.24. Universality of parallel modules is proven by showing a general construction method for any parallel module. □

This construction method is directly relevant to material in this thesis, so we describe it here. We give the general construction method for any parallel module in Figure 2.3. The image in Figure 2.3 is adopted from a combination of Figures 15 and 16 in [23] and the description of the construction in [23]. We give our own explanation of the construction.

Definition 2.26. Given some parallel module $N = (Q, I, O, f, g, A)$ where $I = \{I_1 \dots I_m\}$ and $O = \{O_1 \dots O_n\}$, we define $N' = (Q', I', O', f', g', A')$ to be any serial module (with A' defined appropriately) where if $OSets_N = \{C : ((q, a), C) \in g \text{ for any } q, a\}$:

1. $Q' = Q$ and $I' = \{I'_1 \dots I'_m\}$,
2. $f' = \{((q, I'_i), q') : ((q, I_i), q') \in f\}$,
3. $O' = \{O'_1 \dots O'_k\}$ where $k = |OSets_N|$,
4. $g' = \{((q, I'_i), mapO_N(C)) : ((q, I_i), C) \in g\}$ where $mapO_N$ is any bijection that maps $OSets_N$ to O'.

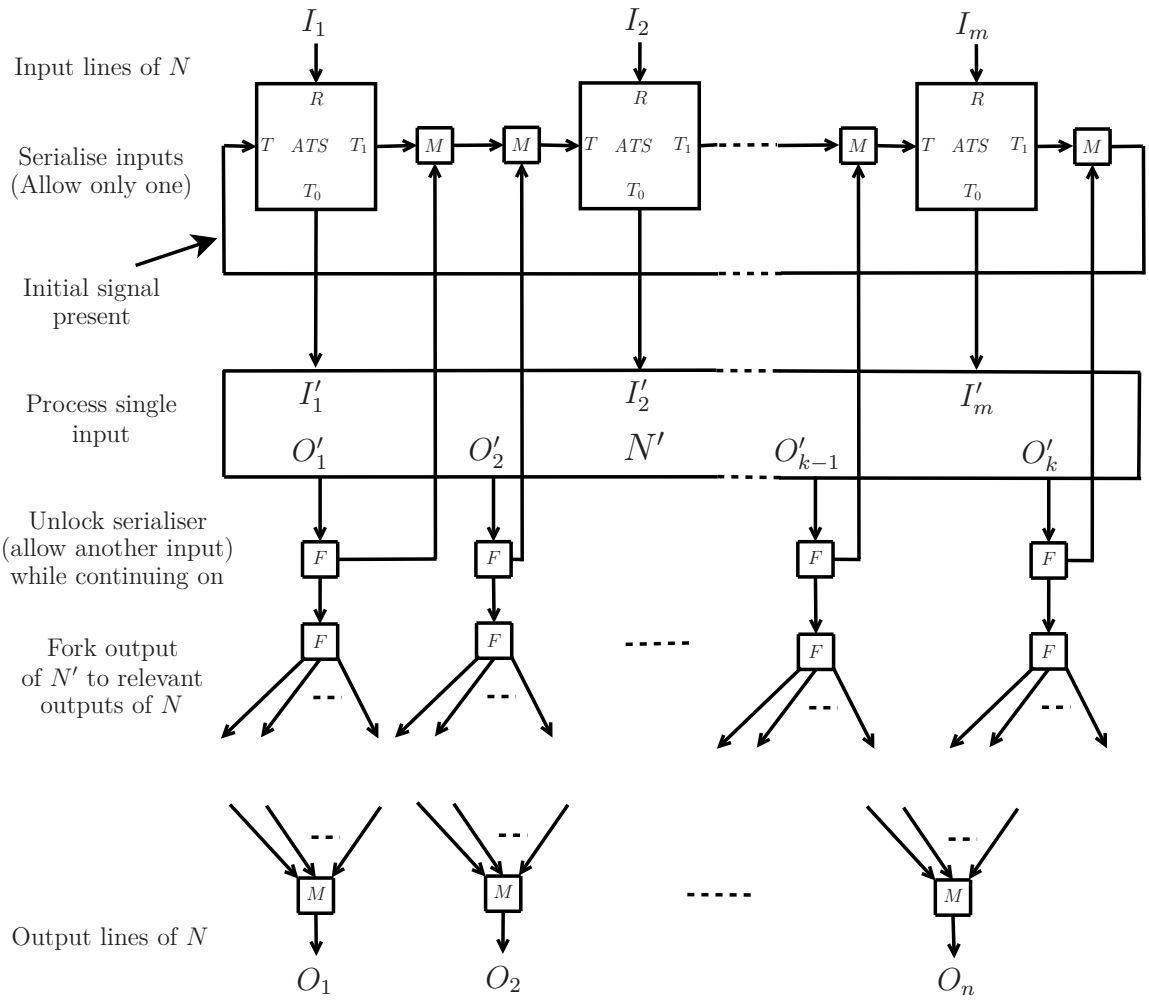


Figure 2.3: Construction method for an arbitrary parallel module N as described by Keller in [23], using {*Merge, Select, Fork, ATS*}, where N' is the serial module with its output lines mapped to output sets of N.

Informally, N' is a serial module which represents the behaviour of N but with output sets replaced with singleton outputs (which are in one-to-one correspondence according to $mapO_N$). The set of states is equivalent between N and N', and the set of input lines and state transition functions are in one-to-one correspondence between N and N'. The only differences lie in the sets of output lines (O and O') and output functions (g and g'). Note that $OSets_N$ may include the empty set.

Assume that initially N' is in the desired state of N, and the *ATS* modules are all in S_1. The construction forces concurrent inputs to N to be processed one at a time by the serial module N'. The top ring of *Merge* and *ATS* modules (with an initial signal circulating around the ring of modules) in Figure 2.3 achieves this by causing only one input signal of N to reach N' at a time. The serial module N' (constructable using the set {*Merge, Select*} as given by Theorem 2.24) processes each input signal on each a of N, and produces the corresponding output signal on b and change of state according to its own definition. This output signal then forks in two directions, with one signal returning to and entering the ring of *ATS* modules via the various *Merge*s; allowing a new input signal to reach N'. Simultaneously, the other output signal connects to a *Fork* tree which corresponds to the output set C where $mapO_N(C) = b$. This sends signals to various *Merge* trees, with each *Merge* tree corresponding to an output line c of the original module N, such that $c \in C$. This has the effect of converting the output signal of N' into the corresponding output set of N, while depositing a signal on each of the appropriate output lines of N.

Observation 2.27. We address a minor oversight of the description given by Keller in [23]. It is possible that some output line b of N' corresponds to the empty output set (if there is some action $(a, \{\}).q'$ in some state q of N, such that $mapO_N(\emptyset) = b$). An output signal on b would therefore not be intended to produce output signals on any $O_1, \ldots, O_n$ of N. Hence the output line b would not be connected to a *Fork* module (unlike all $O'_i \neq b$), and would instead connect directly to the input line of some *Merge* module within the top ring of *Merge* and *ATS* modules.

We note that the parallel module construction method from [23] (and shown in Figure 2.3) can be used to realise any module, including the class of serial modules. However given a serial module N, the definition of N' is exactly equivalent to N. Hence the remainder of the construction in Figure 2.3 is redundant, as it suffices to simply realise N' using {*Merge, Select*}. However, in the rest of this thesis we consider that it may be used to realise any module, and not just the class of parallel modules.

Remark 2.28. We note that there exists numerous other proven universal sets for the serial and parallel classes of modules. These can be found in [23], as well as

other publications such as [59, 60].

Finally, we define notions of a *transition* and a module *processing* an action.

Definition 2.29. Consider some module (Q, I, O, f, g, A) with some action $(a, B).q'$ of some state $q \in Q$. Assume that the module is in state q, and the environment has sent a signal on the input line a. We refer to the acceptance of a signal on a, the production of signals on all $b \in B$, and a move to state q' as a *transition*. We also say that the module has *processed* the action $(a, B).q'$.

2.2 Reversibility

Subsequent work by Patra and Fussell [59, 60] went into finding more efficient universal sets of modules, where efficiency is measured as low modularity and low cardinality.

Reversible modules in general were originally studied by Fredkin and Toffoli [12] who proposed a number of synchronous universal boolean logic gates. In the context of DI modules however, more recently, Morita, Lee, Peper and Adachi carried out research into finding efficient universal sets of reversible serial modules with memory [31, 32, 33, 36, 45, 46, 47, 48, 49, 56]. Reversibility is simple to define in the context of serial modules, as this is simply when no two actions share both the same singleton output set and resulting state.

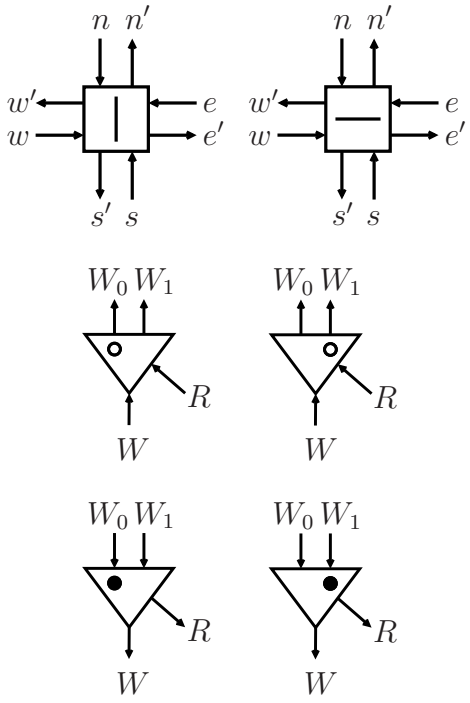
We define reversibility in the context of our CCS-like notation.

Definition 2.30. A serial module N is *reversible* if there are no two states q_1 and q_2 of N such that $(a, \{b\}).q_1'$ and $(a', \{b'\}).q_2'$ are actions of q_1 and q_2, respectively, and $q_1' = q_2'$, $b = b'$ and $q_1 \neq q_2$. A serial module is *irreversible* if it is not reversible.

Rotary Element (RE) [45], *Reading Toggle* (RT) and *Inverse Reading Toggle* (IRT) [32, 36] are examples of reversible modules. These can be found in Figure 2.4. The definitions of RT and IRT are taken from [32].

Example 2.31. For RE, the inputs n, s, w and e represent, informally, the "north", "south", "west" and "east" directions of input respectively. The outputs are analogous. V and H represent "vertical" and "horizontal" respectively, and refer to the depiction of the state as a rotating bar.

Example 2.32. The two modules RT and IRT are each others' mutual inverse. Informally, RT behaves as follows. A signal on R in state 1 will query the state, and cause an output signal on W_1. An input signal on R in state 0 is undefined and assumed to never occur. Signalling W will query the state by producing an output signal on W_1 or W_0, depending on whether the module is in state 1 or 0 respectively,



$$
\begin{aligned}
V &= (n,\{s'\}).V + (s,\{n'\}).V \\
&+ (w,\{s'\}).H + (e,\{n'\}).H \\
H &= (n,\{w'\}).V + (s,\{e'\}).V \\
&+ (w,\{e'\}).H + (e,\{w'\}).H
\end{aligned}
$$

$$
\begin{aligned}
0 &= (W,\{W_0\}).1 \\
1 &= (W,\{W_1\}).0 + (R,\{W_1\}).1
\end{aligned}
$$

$$
\begin{aligned}
0 &= (W_1,\{W\}).1 \\
1 &= (W_1,\{R\}).1 + (W_0,\{W\}).0
\end{aligned}
$$

Figure 2.4: *Rotary Element*, *Reading Toggle* and *Inverse Reading Toggle*. The left column of images depicts the modules in states V, 0 and 0 respectively. The right column of images depicts the modules in states H, 1 and 1 respectively.

and will also cause the module to change to the other state. Note that there is no action which produces an output signal on W_0 and then moves to state 0. *IRT* is the inverse of this module's behaviour, and hence signalling W_0 in state 0 is undefined and assumed to never occur.

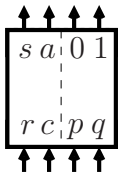
Merge is an example of an irreversible module, as both actions share the same output set ($\{c\}$) and resulting state (M).

The set of all two-state reversible modules with two, three and four pairs of input/output lines, such that all input lines are present in actions in all states, was enumerated in [49]. However, a comparatively small amount of research has been carried out into reversible parallel modules.

We note an important universality result.

Proposition 2.33. $\{RE\}$ and $\{RT,\ IRT\}$ are each universal for the class of reversible serial modules.

Proof. Universality of $\{RE\}$ is shown in [48] through a general construction method for any reversible serial module. Universality of $\{RT,\ IRT\}$ is shown in [36] through construction of *RE*. □



$$
\begin{aligned}
S_0 &= (q,\{0\}).S_0 + (p,\{1\}).S_0 + (r,\{s\}).S_1 + (c,\{s\}).S_a \\
S_a &= (r,\{a\}).S_1 \\
S_1 &= (q,\{1\}).S_1 + (p,\{0\}).S_1 + (r,\{s\}).S_0 + (c,\{s\}).S_b \\
S_b &= (r,\{a\}).S_0
\end{aligned}
$$

Figure 2.5: *Distributed Memory* module and behaviour specification

2.3 Distributed memory modules

We now introduce our own reversible serial memory modules in the sequential machine model. We use these to infer some simple universality results.

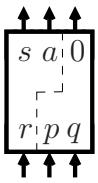
Example 2.34. A *Distributed Memory* (*DM*) module is an eight-line, four-state module with one-bit memory, given in Figure 2.5.

In the following, states S_0 and S_1 are referred to as *steady states* and S_a and S_b as *processing states*. Informally, the module is composed of two systems of functionality, separated by the dashed line. The input and output lines to the left are responsible for controlling the modification of the internal state, while the input and output lines to the right are responsible for querying the state. In S_0, the module may be said to "hold" the value 0, and similarly for S_1 and the value 1. The behaviour of the module is described as follows. In a steady state, a query may occur by signalling either the line q or p, which outputs a signal corresponding to the held value or its inverse respectively, via output lines 0 or 1. Signalling the input line q is referred to as a *query*. Similarly, signalling the input line p is referred to as an *inverse query*. An input signal on r causes an output signal on the s line, and the module moves to the other steady state. This is referred to as a *toggle*. Alternatively, an input signal on c causes an output signal on s and move to a processing state (S_a or S_b depending on whether the steady state was S_0 or S_1 respectively). It remains in this state until an input signal on r, during which it will move to a new steady state (the complement of the previous steady state) and then send an output signal on a.

A network of these modules connected in a ring, with each module's s output line connected to the next module's r input line, will cause all modules to toggle their state in sequence when an input signal is sent on any one of the modules' c lines. The module which initiated the toggle will eventually output a signal on its a line to indicate completion when the toggle signal has completed a full revolution.

Example 2.35. A *Reduced Distributed Memory* (*RDM*) is a six-line, four-state module with one-bit memory, given in Figure 2.6.

Informally, the behaviour of this module is identical to *DM* with two exceptions. Firstly, the module lacks the 1 and c lines. Secondly, if q is signalled when in



$$
\begin{aligned}
S_0 &= (q,\{0\}).S_0 + (p,\{s\}).S_a + (r,\{s\}).S_1 \\
S_a &= (r,\{a\}).S_1 \\
S_1 &= (q,\{s\}).S_b + (p,\{0\}).S_1 + (r,\{s\}).S_0 \\
S_b &= (r,\{a\}).S_0
\end{aligned}
$$

Figure 2.6: *Reduced Distributed Memory* module and behaviour specification.

S_1, or p is signalled when in S_0, the module automatically enters the corresponding processing state. We note that *RDM* can be trivially simulated by *DM* by connecting the *DM*'s 1 line to its c line.

Both modules are serial. The previous state and input line of an action is always determined by the resulting state and the output line, and hence both modules are reversible, satisfying the conditions of Definition 2.30 in Chapter 2.

Observation 2.36. *DM* (similarly to *RE*), is its own functional inverse. This can be seen by running the module backwards (i.e. if $(r,\{s\}).S_1$ is an action of S_0, then in the "inverted" module, $(s,\{r\}).S_0$ is an action of S_1), changing output lines to input lines (and vice versa) and then swapping the names of the lines as given by the pairs $(0,q),(1,p),(s,r)$ and (a,c). Informally, this is a particularly desirable property as it implies that the "reverse" behaviour of a network of *DM*s can be achieved without using different modules. However *RDM*, much like *RT* or *IRT*, does not have this property.

In the diagrams that follow, we indicate a steady state of either S_0 or S_1 with a 0 or 1 in the centre of a module respectively. The use of a *DM* or *RDM* can be distinguished via the presence or absence of the c and 1 lines. In general, we also rearrange the locations of input and output lines in order to improve readability of diagrams.

2.3.1 Reversible serial universality results

We prove the universality of *DM* and *RDM* for the class of reversible serial modules.

Example 2.37. We demonstrate the construction of *RE* using just *RDM*s in Figure 2.7. We illustrate the behaviour of the network when simulating *RE* processing the action $(w,\{s'\}).H$ in state V, and the final resulting state of the network.

The construction in Figure 2.7 uses only four modules, an alternative to the approach using six *Select* modules as demonstrated in [28]. Furthermore, unlike in [28], this decomposition does not use *Merge* and therefore contains only reversible modules. We also contrast this approach with a decomposition using *RT* and *IRT* as demonstrated in [36]. Though our modules are more complex than *RT* and *IRT*,

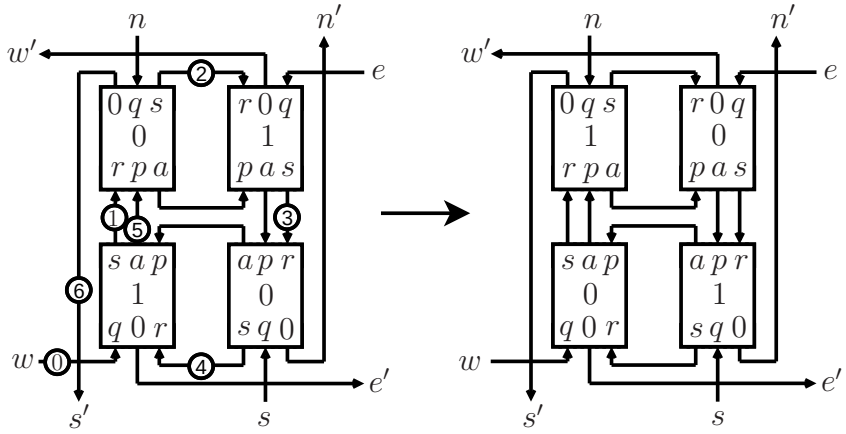


Figure 2.7: *Rotary Element* realised using *RDM*s processing the action $(w, \{s'\}).H$. The left network shows the initial state of the network corresponding to the state V, as well as the series of transitions (represented by numbered circles on the appropriate lines) following a signal on w. The final state corresponding to H is shown on the right. Intermediate states of modules are not shown.

the resulting decomposition is much simpler and more intuitive. This decomposition in particular demonstrates the advantage that our module has in networks which require multiple copies of a single memory value. The need for additional modules dedicated to controlling homogeneous updates is removed.

Theorem 2.38. $\{DM\}$ and $\{RDM\}$ are each universal for the class of reversible serial modules.

Proof. Figure 2.7 demonstrates a construction of *RE* using only *RDM*s. The theorem follows from the universality of *RE* (Proposition 2.33). Furthermore, *DM*s may be used in place of *RDM*s (by connecting the 1 line to the c line for each *DM*), so *DM* is also universal. □

2.3.2 Serial universality results

Next, we illustrate the use of our modules in the domain of networks which realise irreversible serial modules.

Example 2.39. Recall *Select* (Figure 2.1). We show in Figure 2.8 how to realise *Select* using *RDM*s and *Merge*s, and furthermore we illustrate the behaviour of the network when signalling the input line corresponding to T, in order to simulate *Select* processing the action $(T, \{T_1\}).S_1$ in S_1.

Theorem 2.40. $\{$*Merge, DM*$\}$ and $\{$*Merge, RDM*$\}$ are each universal for the class of serial modules.

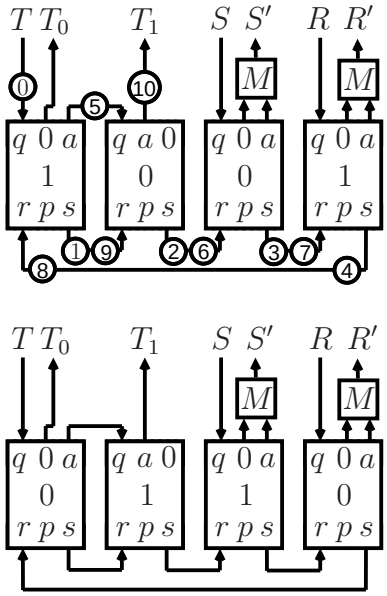


Figure 2.8: (Top) *Select* realised using *RDM*s and *Merge*s processing the action $(T, \{T_1\}).S_1$ in state S_1. The series of transitions is represented by numbered circles on the appropriate lines. The final state is the same as the one depicted in the top image. Intermediate states of modules are not shown. (Bottom) The same network in the state corresponding to S_0 of *Select*.

Proof. The universality of {*Merge, RDM*} follows from the universality of {*Merge, Select*} in [23] and the construction of *Select* using {*Merge, RDM*} in Figure 2.8. As *RDM* can be simulated using a *DM* (by connecting the 1 line to the c line), it follows that {*Merge, DM*} is universal for the class of serial modules. □

Universality of {*RT, IRT*} for the class of reversible serial modules gives us the following corollary.

Corollary 2.41. {*RT, IRT, Merge*} is universal for the class of serial modules.

Proof. Theorem 2.40 combined with Proposition 2.33. □

We are not aware of any other proof of the universality of {*RT, IRT, Merge*} for the class of serial modules.

2.4 Related work and alternative models

2.4.1 Implementation of DI modules in CMOS

The reader may be interested to know how delay-insensitive networks correspond to the prominent CMOS technology [2] used in contemporary computing devices. Recall that CMOS technology implements boolean logic gates, and utilises differing voltage levels to carry information. This seems contrary to asynchronous models

which are based around signals and events, such as the one we have described here by Keller. Asynchronous circuits in CMOS therefore use hand-shaking systems where components are connected via wires in both directions. Components then alternate logic values using some pre-determined protocol to transmit information. This allows components to detect when information has been successfully transmitted and acknowledge receipt between the sender and recipient. See [23, 75, 76] for examples of this approach. In a sense, this simulates the event-based model used by Keller. However this technique introduces significant overhead in terms of space. Furthermore, [40] shows that it is not possible to realise the full Turing-complete class of digital circuits with typical logic gates such as NAND and XOR while requiring delay-insensitivity. Therefore, at least some timing assumptions are required for correct operation of an asynchronous circuit using logic gates, and CMOS circuits which claim to be delay-insensitive are not actually such at all levels of abstraction. However, [38] shows that logic gates are Turing-complete when *quasi-delay-insensitivity (QDI)* is assumed. This is similar to delay-insensitivity but requires that certain forking wires produce signals which travel concurrently at the same speed. As a result of this, some publications [5] use the terms DI and QDI interchangeably. We note briefly that the class of circuits realisable in the CMOS QDI model is functionally identical to the class of circuits realisable in the CMOS speed-independent (SI) model, as discussed in [16].

It is shown in [40] that it is not possible to realise the full Turing-complete class of digital circuits with typical logical gates such as NAND and XOR while requiring delay-insensitivity.

Instead, implementations of DI modules and networks in alternative technologies, such as cellular automata [29] and RSFQ circuits [63], are researched actively in recent years. See Chapters 8 and 9 for further details on implementation of DI networks using Self-Timed Cellular Automata.

2.4.2 Variations on the sequential machine model

There exist several variations on Keller's sequential machine model for DI networks. Each contains additional features which are exploited in order to minimise complexity and sizes of universal sets. We note a few here.

The first is the model in [34] which supports buffering of lines. This allows multiple signals to be present on a line at a time.

The second is the bidirectional buffering model used in [35]. This permits buffering of lines similarly to the above model, but also allows input lines to act as output lines and vice versa. Note however that a signal moves in the same direction until being absorbed by a module. The direction that a signal moves is dependent upon

the end on which it is deposited.

Finally, we note the Brownian model of DI systems which is explored in [67]. This allows bidirectionality of lines, similarly to the bidirectional buffering model, but signals may move in either direction along a line at any time. This is named after the natural phenomenon of Brownian Motion which particles exhibit when suspended in fluid. [64] explores *Brownian Cellular Automata*, which can be used to model this type of network, similarly to the STCA shown in Chapters 8 and 9.

Alternative "DI" models of computation

We note Josephs' related work on DI process algebra in [19, 21], but this is not intended for the modelling of Keller's DI modules and networks. This model involves specifying desired environment behaviour in terms of a trace of events, known as a *specification*. An event represents a communication between two objects. Specifications are then decomposed into parallel compositions of smaller traces known as *components*. Various algebraic laws using a notion of delay-insensitivity are defined in order to permit notions of equivalence between traces. Hence a specification defines all possible "valid" system behaviours, and research involves finding methods of decomposing such a system into constituent components (also specified in terms of their environment behaviour) such that certain properties are guaranteed to always hold.

Contrast the above with Keller's sequential machine model where modules have states and sets of transitions (via f), and input/output behaviour of modules is considered independent of any specified behaviour of the environment. Research surrounding Keller's sequential machine model typically involves starting off with individual components, and studying the behaviour produced as a result of composition. Expressiveness (universality) of certain sets of components is also a significant focus. Environment behaviour remains abstract and is defined informally on a per-case basis, and as a consequence, in general, the sets of "valid" behaviours of systems are incapable of being enumerated or reduced to a single expression.

We note [18] which shows how to implement expressions given in Josephs' algebra using logic gates. There also exists a similar algebraic model by Ebergen shown in [9, 10]. We also note that the notation used in Ebergen's model is used by Patra [59, 60, 61] to describe the behaviour of modules within Keller's sequential machine model.

2.5 Shortcomings of Keller's sequential machine model

We now briefly discuss some issues with the sequential machine model and associated research into DI networks.

The method of processing concurrent signals (one signal at a time) mirrors the interleaving approach to modelling concurrency observed in certain process algebra models like CCS [42]. In a practical setting where such a system may be implemented, this can be considered less efficient than an approach where multiple signals are processed simultaneously, which corresponds to "true-concurrency" models of concurrency such as Petri Nets [73].

Consider the behaviour of *ATS* from Figure 2.1. It is valid in S_1 to send input signals on both R and T concurrently, or individually. Depending on the delays in lines, and the order of processing by the module, this can lead to different outcomes. In [23], this is referred to as *non-trivial arbitration*. If the order of processing between two signals does not affect the output or resulting state, it is known as *trivial arbitration*. In the case of non-trivial arbitration, the module can be seen to make a "choice" which affects the overall outcome of the computation. Hence non-trivial arbitration, when combined with delay-insensitivity results in a form of non-determinism.

Trivial and non-trivial arbitration, however, are only defined with respect to pairs of signals. In the general case where several signals may arrive concurrently, leading to different possible states where different sets of inputs may be defined, the situation is much more complex. It is not always clear from the definition whether a module exhibits high-level non-deterministic behaviour. High-level "arbitration" in this sense is briefly mentioned by Keller [23], but is not formally defined or further elaborated on.

Constructions by Keller and by Patra and Fussell of arbitrary parallel modules make no clear distinction between those modules which utilised high-level non-deterministic behaviour and those which did not. As a result, all current constructions of parallel modules, whether they possess clear non-deterministic behaviour or not, utilise such modules. By extension, universal sets for parallel modules exist only for all parallel modules, and an identification of which modules are required for only "deterministic" parallel modules is not possible. We note that it is conjectured in [23] that the set of modules {*Merge, Select, Fork, Join*} is universal for the class of modules which do not "require arbitration", while it is noted that this notion has not been formalised.

Keller's construction method is also highly inefficient as it forces all concurrent signals to be processed in a serial manner. The sequential approach to processing

input signals makes determining the overall effects of a set of concurrent input signals to a module difficult. As a result, designing a construction method for an arbitrary module which utilises concurrency appears challenging.

Furthermore, it is also not always clear whether a module exhibits "high-level" reversibility. For example, simply inverting the definition of *Join* to yield its inverse is not possible, as this would result in actions where "empty" input lines produce output sets, which is not a valid definition. However *Join*'s abstract behaviour clearly exhibits a form of reversibility in the sense that an output signal on c is necessarily caused by a single input signal on both a and b each.

Together, these issues suggest that Keller's approach to modelling concurrent delay-insensitive networks (where modules process input signals one at a time) does not naturally correspond to their behaviour as seen by an external observer. A model which implements concurrency more directly would in theory allow us to define reversibility for parallel modules, while allowing a clear distinction between modules which exhibit high-level deterministic and non-deterministic behaviour.

Additionally, interactions of the environment with such modules are described informally. It is reasonable to assume that an environment's only information about a system is deduced from the sequences of input signals it sends and output signals it receives. However there currently exists no means to determine in general what sequences of input signals may be sent to a module without ultimately causing a "clash" on lines (Condition 6) or violating the safety assumption (Definition 2.5). Reasoning about an environment's knowledge of a module's state is difficult due to the presence of empty output sets in actions of a module. This is itself a direct consequence of the manner in which concurrent combinations of input signals give rise to output signals. Despite this, the operating conditions of Keller's sequential machine model (particularly Conditions 8, 9 and 10) imply specific restrictions on how an environment may operate. However, the above-discussed lack of formal rigour makes it difficult to study the limitations resulting from such conditions. For the same reason, it is also not clear how to guarantee that such conditions are enforced when defining modules. We note [8] which defines a notion of "receptiveness" in the context of a trace theory for speed-independent logic circuits, which concerns the ability for a component to respond to any possible inputs which the environment may provide.

Similarly the concept of implementation of a module, using a network of other modules in order to simulate its behaviour (referred to by Keller as a *realisation*), remains an informal concept. This is difficult to define between modules and networks, due to the informal way in which environments and "external behaviour" are defined. As discussed in the previous section, this is compounded by the fact that all possible "valid" behaviours of systems in this model cannot be enumerated.

Finally, the exact set of conditions which are considered essential for correct operation of DI modules and networks varies between publications. The most prominent of these is Keller's safety assumption (Definition 2.5). This was explicitly enforced by Keller. However it is explicitly relaxed in some publications [34], unaddressed in others [59, 60], and is defined but then subtly ignored elsewhere [30]. The result is that there are small differences between the models used and it is not always clear whether results which hold in one version of the model hold in another. A lack of consistent notation for describing module behaviour [23, 30, 60] further compounds this issue and makes the identification of behavioural properties of modules difficult.

We address these shortcomings via the development of a new model for DI networks called the *Set Notation* model. Details can be found in Chapter 3 onwards. The Set Notation model allows modules to accept input signals concurrently, and does not allow empty output sets to be defined in actions of modules. This allows us to define notions of reversibility. Furthermore, the manner in which Set Notation modules process concurrent signals allows us to make a clear distinction between modules exhibiting deterministic and non-deterministic behaviour. This is visible in the notation used to define modules, and is formalised directly as a property of module definitions. Finally, definitions of modules and networks are considered independent of any assumptions about environment behaviour. Instead we formalise the notion of an environment and identify behaviours which result in desirable properties holding for modules and networks. This contrasts with the sequential machine model which attempts to restrict environment behaviours as part of its operating conditions, while also lacking a formal notion of an environment.

2.6 Conclusion

In this chapter we gave an overview of the existing model for DI networks formulated by Keller, which we refer to as the *sequential machine* model. We included all operating conditions and relevant definitions. The general construction method for any module in the sequential machine model was detailed in-depth. Further research in the field of DI networks, including work related to reversibility, was also discussed. Several existing universality results were detailed. We introduced our own reversible memory modules in the sequential machine model, and we used these modules to infer some simple universality results. We finished by discussing what we see as the main shortcomings of the sequential machine model for DI networks. Motivation was given for the development of a new DI model which implements concurrency more directly.

Chapter 3

The Set Notation model

In this chapter we introduce a new model for describing the behaviour of delay-insensitive modules, called *Set Notation* which more naturally models concurrency and notions of reversibility. We define important classes of modules in the Set Notation model, such as the non-arb, eq-arb and arb classes. We define networks of modules in the Set Notation model, along with the execution behaviour of such networks. We define properties of such networks, such as the non-clashing and safety properties. We investigate several further properties of modules in the Set Notation model which limit the behaviour of the environment in unexpected ways. Examples include the auto-firing or 1-step consistency properties of modules.

Set Notation was introduced in [53].

3.1 Basic definitions and conditions

We wish to use a new type of module where for each action, inputs as well as outputs are sets. Each input set will correspond to a valid combination of individual input signals which may be sent in a given state.

Definition 3.1. A *module* is a 4-tuple (Q, I, O, T), where:

1. Q is a finite set of states,
2. I is a set of input lines and O is a set of output lines and $I \cap O = \emptyset$,
3. $T : Q \times (P[I]) \setminus \emptyset \to Q \times (P[O]) \setminus \emptyset$ is a partial map, called the *transition map*, and it assigns an input set in a given state to an output set and a new state.

Informally, T denotes what we call the input/output behaviour of a module. This describes the effects of a concurrent set of input signals to a module. We note that the sequential machine model requires the f and g partial maps to be *partial functions*, whereas Set Notation is more general by also allowing partial

maps which are not partial functions. Also note that the definition of T requires that no element of T contains an empty output set. Empty output sets in actions are necessary in the sequential machine model due to the way in which concurrent input signals together produce output signals. This requires multiple actions, one for each possible input signal. In the Set Notation model, this functionality is expressed directly as a single element of T and hence empty output sets are not required. Informally, any module will always eventually produce at least one non-empty set of output signals in response to a set of input signals. Therefore each element of T can be seen to represent the least required set of signals in order to produce some set of output signals from a module. Interpreting the notation in this manner makes clear why allowing empty output sets for modules does not make sense in this model. Similarly, allowing empty output sets complicates any possible notion of reversibility that may be defined. We will represent elements of T as a tuple. Hence we say $((q, A), (q', B)) \in T$ iff $T((q, A)) = (q', B)$. We will omit the inner most brackets when representing elements of T in tuple form. Hence if $((q, A), (q', B)) \in T$, we will simply write this as $(q, A, q', B) \in T$. Elements of T are also called *transitions*.

As in Chapter 2 concerning the sequential machine model, we use a CCS-like notation to present more easily the definitions of modules.

Definition 3.2. If $(q, A, q', B) \in T$ is defined, then $(A, B).q'$ is called an *action* of q, where (A, B) is an input/output pair and q' is the resulting state.

We specify all actions of q by writing $q = (a_1, B_1).q_1 + \ldots + (a_n, B_n).q_n$ where $(q, A_x, q_x, B_x) \in T$ is defined for all $1 \leq x \leq n$. Then the definition of a module N is given by a set of such equations, one for each state of N. Input and output lines are implicit.

For simplicity of the model, we require that no two different states have the same sets of actions, and each state has at least one action. Sometimes we write $(A, B).q' \in q$ to mean $(A, B).q'$ is an action of q.

Informally, each action corresponds to an element of T and vice versa.

Definition 3.3. A *network* or *circuit* of modules is a collection of instances of modules, each in some state, such that every output line of a module is connected to at most one input line of another module and every input line of a module is connected to at most one output line of another module. Connections between lines of modules are also called *wires*. Each wire has an implicit direction, which is from an output line of the wire to the input line of the wire. The output line of a wire is also known as the *source* of the wire, and the input line of a wire is also known as the *target* of the wire. Each wire may contain any number of signals.

Informally, a signal can be viewed as a discrete "particle" which moves along a wire. Note that unlike in the sequential machine model, the term *line* refers strictly to a component of a module, and does not also refer to the connection between these components through which signals travel. We instead call these wires to avoid confusion. Informally, a network is a collection of modules, each in some state, and wires connecting the lines of these modules, each of which may contain zero or more signals.

Definition 3.4. The *inverse* of $\{Q, I, O, T\}$ is the module $\{Q', I', O', T^{-1}\}$ where $Q' = Q$, $I' = O$, $O' = I$ and T^{-1} is the inverse of T.

Note that since T is a partial map, T^{-1} is defined.

We define the behaviour of wires and modules within a network as follows.

Definition 3.5. execution behaviour: Wires behave as follows. Signals on wires travel towards some module's input line specified as the target of the wire. Following an unbounded but finite amount of time after a signal is placed on a wire, it arrives at the input line specified as the target of the wire. The signal then remains pending at the input line until absorbed by the module.

Modules behave as follows. A module (Q, I, O, T) in state $q \in Q$, with a signal pending at each of its input lines $A \subseteq I$ will, after an unbounded but finite amount of time, non-deterministically and concurrently absorb a set of signals, one pending at each of its input lines in some $A' \subseteq A$, if and only if for some B, q', $(A', B).q'$ is an action of q. After an unbounded but finite amount of time, the module simultaneously places a signal on each wire connected to each line $b \in B$ and changes its state to q'.

When a module places a signal on the wire connected to one of its output lines b, we sometimes say that it has produced a signal *on its output line* b or it has *output on* b. Similarly we say that a module has absorbed a signal *on its input line* a when it absorbs a signal pending on the wire connected to its input line a.

Definition 3.6. We also use the term *pending* to refer to both: signals which have arrived at the input lines specified as the target of the wires but are unabsorbed, and signals which are travelling along wires but have not yet arrived at the input line specified as the target of the wire.

The distinction between the two different scenarios described by the term pending is not made as this makes no difference to the overall behaviour of the network or environment due to delay-insensitivity of wires, and the unbounded amount of time that signals may pend at input lines.

Example 3.7. Consider the module defined partially by the equation $S_0 = (\{a, b, c\}, \{d, e\}).S_1$. This means that in state S_0, if a signal arrives at each of the input lines a, b and c, in any order or concurrently, this will eventually result in the module concurrently producing a single output signal on d and e each while changing to the state S_1.

As a result of the execution behaviour (Definition 3.5), it is clear that networks eventually become other networks. Informally speaking, these possess the same structure as the original network, but the state of each module and the number of signals on each wire may be different. We formalise this notion.

Definition 3.8. Consider some network of modules S. We say a network S' is a *state variation* of S if it is the same as S except:

1. each module in S' may be in a different state to its corresponding module in S,

2. each wire in S' may contain zero or more signals, irrespective of its corresponding wire in S.

Informally, a state variation of a network is the same network, but the state of each module and the number of signals on each wire may be different. It is clear that networks become state variations of themselves as a result of the execution behaviour (Definition 3.5). Note however that just because there exists some network S' which is a state variation of S, does not necessarily mean that it is possible for S to become S' as a result of the execution behaviour. Note also that for all networks S, S is a state variation of itself, and there exists an infinite number of S' such that S' is a state variation of S.

We note that we further formalise the above notions of execution behaviour and state variations using *Structured Operational Semantics* and *Labelled Transition Systems* in the setting of our new DI-Set algebra, which can be found in Chapter 7.

Definition 3.9. Consider some network S. If it is possible for S to become some state variation S' of S as a result of the execution behaviour (Definition 3.5), and there does not exist some state variation S'' of S (where $S'' \neq S'$), such that it is possible to for S' to become S'' as a result of the execution behaviour, we say that S is *deadlocking*. Otherwise it is *non-deadlocking*.

We also assume that Conditions 2, 4, 5 and our New Condition 7 (see Chapter 2) are enforced at all times (with *line* assumed to be replaced by *wire* where appropriate). Conditions 1, 3, 6 and 8-11 do not apply. Condition 1 is unnecessary as Definition 3.1 provides this restriction and Condition 3 is not applicable as

concurrent input signals are absorbed by a module simultaneously according to the execution behaviour (Definition 3.5). The behaviour resulting from Condition 11 is covered by Definition 3.5. The remaining Conditions are intentionally ignored.

We do not require the safety assumption (Definition 2.5 in Chapter 2) to hold as this is not directly applicable under the new concurrent model. Instead we introduce a related property together with a property corresponding to Condition 6.

Definition 3.10. Consider some network S. If for each state variation S' of S which can be reached as a result of the execution behaviour (Definition 3.5) starting from S, at most one signal is present on each wire, we say that S is *non-clashing*; otherwise it is *clashing*.

The non-clashing property corresponds directly with Keller's Condition 6. We define the following property, which we will refer to as the *safety* property.

Definition 3.11. Consider some network S, containing some module $N = (Q, I, O, T)$. If for each state variation S' of S which can be reached as a result of the execution behaviour (Definition 3.5) starting from S, there exists some action $(A, B).q'$ of $q \in Q$, where:

1. q is the state of N in S',
2. $A' \subseteq A$ where $A' \subseteq I$ is the set of input lines of N whose connected wires contain one or more signals in S',

we say that N is *safe* in S. Otherwise it is *unsafe* (or exhibits *non-safety*) in S. Correspondingly, a network S is safe if all modules in S are safe, otherwise it is unsafe (or exhibits non-safety).

Informally, the safety property says that a module does not receive a set of input signals which are not defined as part of some expected input set. The desirability of this property is related to implementation of DI modules as sequential machines (see Chapter 5), and potential physical implementation, where modules without the ability to block undefined input signals should be simpler than those with the ability to force undefined input signals to wait until an appropriate state. Furthermore, meaningful answers to questions regarding behaviour of environments and reversibility of modules can only be formed if at least some restrictions are placed on a module's operation, the simplest of which we consider to be the properties of clashing and safety. Unsafety has a correspondence with the property of "input delaying" in the buffering model used in [34]. As noted above, the safety property corresponds with Keller's safety assumption (Definition 2.5 in Chapter 2).

Recall the lack of Keller's Conditions 8, 9, 10, which correspond to "valid" environment behaviour in the sequential machine model. In the Set Notation model, we

do not assume or require any particular environment behaviour. Instead we will attempt to identify environment behaviour which leads to networks which are always safe and non-clashing. This, together with the safety and non-clashing properties (as opposed to the safety assumption from Definition 2.5 and Condition 6 both in Chapter 2), demonstrates one of the main differences between the sequential machine model and our Set Notation model. Our Set Notation model attempts to formalise preferred behaviour in the form of properties (i.e. safety and non-clashing), as opposed to inherently requiring them as part of the model and then attempting to impose restrictions on modules and environments in order to guarantee the model's correctness.

We give an example of a useful module in the Set Notation model.

Example 3.12. We show the definition of *Join* in Set Notation. It is defined as the one-state module $J = (\{a, b\}, \{c\}).J$, where this indicates that the combination of input signals on $\{a, b\}$ causes an output signal on c.

Remark 3.13. We note that it is possible in some circumstances to view a module definition in Set Notation as an abstraction that represents more naturally the external behaviour of some sequential machine. In this case, the Set Notation module corresponds to some sequential machine if certain conditions are assumed (Section 5.2 in Chapter 5).

Example 3.14. Consider the definition of *Join* given in Example 3.12. Trivially, if non-clashing and safety are to hold for some network containing *Join*, it is necessary that only a single signal arrives on a and b each. No more input signals may arrive until an output signal is sent on c, in which case the same behaviour may repeat indefinitely. In this sense, the external behaviour of the Set Notation *Join* from Example 3.12 is equivalent to the external behaviour of the sequential machine *Join* from Example 2.1 in Chapter 2. Consequently, it is possible to see the Set Notation *Join* as an abstraction of the sequential machine *Join* provided that safety and non-clashing always hold.

We note the following correspondence between certain modules in the sequential machine model and similar modules in Set Notation.

Observation 3.15. Modules in the sequential machine model which contain only singletons in each entry of their A function, and do not contain empty output sets in any actions, can be redefined in Set Notation by simply placing set brackets around the input lines in their CCS-like definitions. The difference in how modules absorb and produce signals between Set Notation (Definition 3.5) and the sequential machine model (Definition 2.3 and Condition 3 in Chapter 2) does not affect the external behaviour of such modules, regardless of whether safety and non-clashing are assumed.

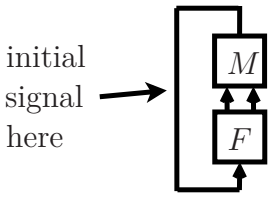


Figure 3.1: A network containing a single *Merge* module and a single *Fork* module. The network is both unsafe and clashing.

Example 3.16. Recall the modules *Fork* and *Merge* defined in the sequential machine model (Figure 2.1 in Chapter 2) which are defined as $F = (a, \{b, c\}).F$ where $A(F) = \{\{a\}\}$ and $M = (a, \{c\}).M + (b, \{c\}).M$ respectively. The corresponding Set Notation definitions of *Fork* and *Merge* are therefore $F = (\{a\}, \{b, c\}).F$ and $M = (\{a\}, \{c\}).M + (\{b\}, \{c\}).M$ respectively. The difference in how modules absorb and produce signals between models does not affect their behaviour, and the new definitions of both modules are trivially equivalent to their sequential machine model counterparts.

We depict the Set Notation modules *Join*, *Fork* and *Merge* using the same symbols used to depict their counterparts from the sequential machine model in Example 2.1 in Chapter 2.

Example 3.17. In Figure 3.1 we show a network which is both unsafe and clashing. When the *Fork* module produces output signals, the *Merge* module is unsafe as it will have signals present on both input lines simultaneously, and there is no action in the definition of *Merge* which contains both input lines. If the *Merge* module then accepts both input signals and produces two output signals, before the *Fork* module can accept one of these on its input line, the wire connecting the *Merge* and *Fork* modules will contain two signals, resulting in clashing.

3.2 Properties of modules

We now outline several key properties of modules defined using Set Notation.

3.2.1 Basic properties and important classes

Definition 3.18. We call a module *serial* if all input and output sets of the module's actions are singletons, and for all states q and all pairs of actions $(A, B).q_1 \in q$ and $(A', B').q_2 \in q$, we have $A \neq A'$. We call a module *non-serial* if it is not serial.

This corresponds to Definition 2.16 in Chapter 2. Note however that the sequential machine model requires that at most one input signal is ever sent to a serial module

at a time, and that these are interspersed with output signals. Definition 3.18 has no such requirement and concerns strictly the definition of a module. Note however that if safety holds then this behaviour is implied.

Correspondingly to Definition 2.30 in Chapter 2, we define reversibility in the context of serial modules in the Set Notation model.

Definition 3.19. We call a serial module (Q, I, O, T) *reversible* if T is a partial bijection, otherwise it is *irreversible*.

Similarly to Observation 3.15, we have the following correspondence between serial modules in the sequential machine model and serial modules in the Set Notation model.

Observation 3.20. Modules in the sequential machine model which contain only singletons in all states of the A function, and singletons in all output sets (i.e. serial modules), can be redefined in Set Notation by simply placing set brackets around each input name in their CCS-like definitions. The resulting Set Notation modules are serial according to Definition 3.18. By Observation 3.15, the difference in how modules absorb and produce signals between Set Notation (Definition 3.5) and the sequential machine model (Definition 2.3 and Condition 3 in Chapter 2) does not affect the external behaviour of such modules, regardless of whether safety and non-clashing are assumed.

Example 3.21. The module *Merge* found by modifying the definition given in Figure 2.1 in Chapter 2 according to Observation 3.20 is irreversible according to Definition 3.19. The module *RT* found by modifying the definition given in Figure 2.4 in Chapter 2 in the same way is reversible according to Definition 3.19.

We now introduce various properties of modules which are not serial.

Definition 3.22. A module N is *arbitrating* (*arb* for short) if there exists some state q with different actions $(A, B).q' \in q$ and $(A', B').q'' \in q$ such that either $A \subseteq A'$ or $A' \subseteq A$. A module is *non-arbitrating* (*non-arb*) if it is not arbitrating.

Informally, arbitration corresponds to a form of non-determinism. As each possible input set in an action corresponds to a set of signals arriving, for the behaviour of a module to be deterministic, no input set of an action can be a subset of an input set of another action in the same state, and no input set can lead to two different output sets or different states.

Example 3.23. All modules defined so far in this chapter are non-arb. An example of a simple arb module is N_0, defined as:

$$S_0 \quad = \quad (\{a, b, c\}, \{x\}).S_0 + (\{a, b\}, \{y\}).S_0$$

Here, the input set $\{a, b\}$ is a subset of the input set $\{a, b, c\}$ in the state S_0. Hence the module may output a signal on y as a result of the environment sending input signals corresponding to either $\{a, b\}$ or $\{a, b, c\}$. In the latter case, following the production of an output signal on y, there is still an input signal pending on c.

Note that all serial modules are non-arb due to the requirement that no two actions in a state contain the same input set.

Example 3.24. Recall the *ATS* module from Figure 2.1 in Chapter 2. We introduce a similar module in Set Notation called *sATS*, defined as:

$$S_1 \quad = \quad (\{T\}, \{T_1\}).S_1 + (\{R, T\}, \{T_0\}).S_1$$

sATS is arb.

It is trivial for an environment, when interacting with a non-arb module directly, to ensure safety and non-clashing. In such a case, the environment simply needs to send input signals corresponding to the input set of some action, and then wait for the corresponding set of output signals. As the module is non-arb, all such input signals are guaranteed to be processed by the module, and the set of output signals is uniquely determined.

However with arb modules, for the environment to ensure safety and non-clashing the situation is more complex, The environment must attempt to deduce which input signals are pending and which output signals to expect, and as a result the set of input signals which it may send such that safety and non-clashing are guaranteed to hold. Note the *sATS* example above where the environment, after sending an input signal on R, is restricted in its ability to send another input signal on R until an output signal is received on T_0.

We make note of a useful subclass of arbitrating modules.

Definition 3.25. A module N is *equal-arbitrating* (*eq-arb* for short) if there exists some state q_1 with different actions $(A, B).q_1' \in q_1$ and $(A, B').q_1'' \in q_1$ and there does not exist some state q_2 with different actions $(A', B'').q_2' \in q_2$ and $(A'', B''').q_2'' \in q_2$ such that either $A' \subset A''$ or $A'' \subset A'$.

Eq-arb modules are those which can be interpreted as having non-determinism purely as a result of an internal choice by the module, rather than non-determinism as a result of either some input signals arriving faster than others or a choice by the module of which input signals to absorb.

We note that modules where all input and output sets of actions are singletons, and there is some state q with two different actions $(A, B).q' \in q$ and $(A, B').q'' \in q$, are not considered serial modules (as they do not fit the requirement of Definition

3.18 that no two actions of the same state contain equal input sets). Instead they are simply eq-arb.

Definition 3.26. A module N is *backwards-arbitrating* (*b-arb* for short) if there exist two states q_1 and q_2 with different actions $(A, B).q \in q_1$ and $(A', B').q \in q_2$, such that $B \subseteq B'$ or $B' \subseteq B$. A module is *non-backwards-arbitrating* (*non-b-arb*) if it is not b-arb.

Many b-arb modules are not logically reversible because their transitions maps are not partial bijections. There are, however, logically reversible b-arb modules, whose inverses are not forwards-deterministic in a DI environment due to the presence of inclusion between input sets in the same state. This is as a consequence of the inverse of a b-arb module being an arb module.

Example 3.27. Consider the module $sATS^{-1}$ defined as:

$$S_1 \quad = \quad (\{T_1\}, \{T\}).S_1 + (\{T_0\}, \{R, T\}).S_1$$

The transition map T given by $(S_1, \{T_1\}, S_1, \{T\})$ and $(S_1, \{T_0\}, S_1, \{R, T\})$ is clearly a partial bijection. However, the inverse of this module is the module *sATS* (Example 3.24). *sATS* exhibits non-deterministic behaviour, as sending input signals corresponding to $\{R, T\}$ concurrently could produce an output signal on T_0 or it could result in an output signal on T_1 and an input signal remaining pending on R.

Observation 3.28. There is a mismatch between the notion of logical reversibility when considering a module's definition, and the notion of a module being "invertible" in a DI environment. We therefore use the term *bijective module* strictly when talking about a module where T is a partial bijection.

A diagram showing the relationships between classes of modules defined using Set Notation can be seen in Figure 3.2. Note that the overlap between the serial and b-arb classes represents irreversible serial modules. The remainder of the serial class represents reversible serial modules.

3.2.2 Advanced properties

We now identify several properties of module definitions which limit the behaviour of the environment. We also define some functions related to these properties to more easily facilitate the generation of environments in the next chapter.

Definition 3.29. A module N is *stable* if for all states q and all different actions $(A_1, B_1).q_1 \in q$ and $(A_2, B_2).q_2 \in q$ such that $A_1 \subseteq A_2$, there exists some action $(A_1', B_1').q_1' \in q_1$ such that $(A_2 \setminus A_1) \subseteq A_1'$. A module is *unstable* if it is not stable.

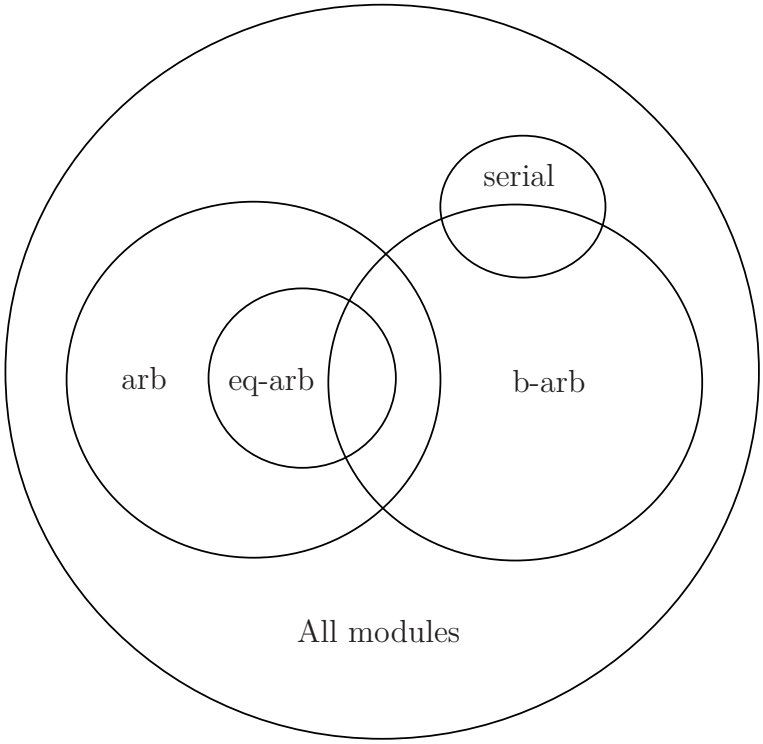


Figure 3.2: Classes of Set Notation modules.

Informally, a module is stable if it means that there is no action such that signalling its input set can result, due to arbitration, in a state where non-safety occurs. If safety is required, then the environment cannot signal such an input set, as there is always the possibility that this may result in non-safety.

Example 3.30. The module N_1 defined as:

$$\begin{aligned} S_0 &= (\{a,b\},\{x\}).S_0 + (\{b\},\{y\}).S_1 \\ S_1 &= (\{c\},\{z\}).S_0 \end{aligned}$$

is unstable, as there is no action in S_1 with an input set that contains a, and hence sending input signals corresponding to $\{a,b\}$ in S_0 may cause the module to receive an input signal on a in the state S_1, resulting in non-safety. However, the module N_2 defined as:

$$\begin{aligned} S_0 &= (\{a,b\},\{x\}).S_0 + (\{b\},\{y\}).S_1 \\ S_1 &= (\{c\},\{z\}).S_0 + (\{a,b\},\{z\}).S_0 \end{aligned}$$

is stable.

Definition 3.31. A module N is *auto-firing* if there exists some state q with different actions $(A_1,B_1).q_1 \in q$ and $(A_2,B_2).q_2 \in q$ such that $A_1 \subset A_2$, and there exists some action $(A_1',B_1').q_1' \in q_1$ such that $A_1' \subseteq (A_2 \setminus A_1)$. A module is *non-auto-firing* if it is not auto-firing.

Informally, a module is auto-firing if it means that there exists some action such that signalling its input set may result in the processing of more than one action in succession before the environment has a chance to send any more input signals.

Example 3.32. The module N_3 defined as:

$$\begin{aligned} S_0 &= (\{a,b\},\{x\}).S_0 + (\{b\},\{y\}).S_1 \\ S_1 &= (\{a\},\{z\}).S_0 \end{aligned}$$

is auto-firing (and stable), as sending input signals corresponding to $\{a,b\}$ in S_0 may result in the module processing the action $(\{b\},\{y\}).S_1$ followed by the action $(\{a\},\{z\}).S_0$, even if no more input signals are sent. The module N_2 in Example 3.30 is non-auto-firing.

Definition 3.33. Given some module $N = (Q, I, O, T)$, a *terminating autopath* for input set $A \subseteq I$ and state $q \in Q$ is a sequence $(A_0, B_0).q_0 \ldots (A_n, B_n).q_n$ where if $A' = \bigcup_{i=0}^{n} A_i$:

1. $(A_0, B_0).q_0$ is an action of q,
2. for all $0 < i \leq n$, $(A_i, B_i).q_i$ is an action of q_{i-1},
3. for all different $0 \leq i, j \leq n$, $A_i \cap A_j = \emptyset$,
4. $A' \subseteq A$,
5. there is no action $(A_{n+1},B_{n+1}).q_{n+1}$ of q_n where $A_{n+1} \subseteq (A \setminus A')$.

For all A, q, we define $autoPaths(A, q)$ to be the set of all possible terminating autopaths for input set A and state q.

Informally, the set $autoPaths(A, q)$ contains all possible sequences of actions which may be processed as a result of signalling the set A in state q. We say that a terminating autopath $p \in autoPaths(A, q)$ is *fired* if the sequence of actions given by the path p is processed as a result of signalling A.

Example 3.34. The set of autopaths $autoPaths(\{a,b\}, S_0)$ for module N_3 in Example 3.32 is the set containing the two terminating autopaths $(\{a,b\},\{x\}).S_0$ and $(\{b\},\{y\}).S_1, (\{a\},\{z\}).S_0$.

We note that signalling the input set of any action of a stable module N, cannot lead to a state of N where non-safety occurs simply as a result of the firing of an autopath.

Definition 3.35. A module $N = (Q, I, O, T)$ is *auto-clashing* if there exists some autopath $p = (A_0, B_0).q_0 \ldots (A_n, B_n).q_n \in autoPaths(A, q)$ for some $q \in Q$ and some $(A, B).q' \in q$, such that $(B_i \cap B_j) \neq \emptyset$ for some different $0 \leq i, j \leq n$. The firing of p is said to cause an *auto-clash*. A module is *non-auto-clashing* if it is not auto-clashing.

Informally, a module is auto-clashing if it is possible to result in two or more signals on a wire connected to one of the module's output lines, simply as a result of the firing of an autopath. We note that if a module is non-auto-firing then it is non-auto-clashing. If non-clashing is required, then the environment must ensure that such an autopath is never fired as a result of any input signals it may send. The only way to guarantee this is to never signal an input set that may result in an auto-clash.

Example 3.36. The module N_4 defined as:

$$\begin{aligned} S_0 &= (\{a, b\}, \{x\}).S_0 + (\{b\}, \{y\}).S_1 \\ S_1 &= (\{a\}, \{y\}).S_0 \end{aligned}$$

is auto-clashing (stable, and by implication auto-firing), as the output line y appears twice in the terminating autopath $(\{b\}, \{y\}).S_1, (\{a\}, \{y\}).S_0 \in autoPaths(\{a, b\}, S_0)$. This implies that signalling the set $\{a, b\}$ in S_0 may lead to multiple signals produced on the output line y. The module N_3 in Example 3.32 is non-auto-clashing.

Definition 3.37. A module N is *1-step consistent* if for all states q, and all different actions $(A_1, B_1).q_1 \in q$ and $(A_2, B_2).q_2 \in q$ such that $A_1 \subseteq A_2$, it is the case that $B_1 \not\subseteq B_2$ and $B_2 \not\subseteq B_1$.

Informally, if a module is 1-step consistent then the environment, after sending input signals corresponding to the input set of an action, can deduce which action was processed by the module, based purely on the observation of output signals from the module. If the module is not 1-step consistent, an environment cannot (due to delay-insensitivity) assume that certain output signals may or may not be travelling along wires, and as a result cannot infer which input signals were processed and the current state of the module. We refer to this run-time phenomenon as *inconsistency*. We will show later in Chapter 4 that inconsistency can arise in delay-insensitive networks even if a module is 1-step consistent, stable, and non-auto-clashing.

Example 3.38. The modules N_1, N_2, N_3 and N_4 in the previous examples are all 1-step consistent. The module N_5 defined as:

$$S_0 = (\{a, b\}, \{x, y\}).S_0 + (\{b\}, \{y\}).S_0$$

is not 1-step consistent. Informally this means that if the environment signals the input set $\{a, b\}$ in S_0, receiving an output signal on y is not enough to deduce which action was processed by the module, and whether or not an input signal is pending on a. It is not possible for the environment to wait for an output signal on x in order to deduce this information, as there is no upper bound for the time taken for an output signal to arrive. The environment must therefore continue its operation while considering the possibility that a signal may or may not be pending on a. In this example, the environment may now only send a new input signal on b if it is to guarantee that safety and non-clashing hold.

3.3 *ATS*, *sATS* and external behaviour

We finish this chapter by informally examining the relationship between *ATS* (Figure 2.1 in Chapter 2) and *sATS* (Example 3.24).

Assuming that there are no input signals pending, consider the possible sequences of signals that the environment may send and receive from *sATS*, such that safety and non-clashing always hold. Sending an input signal on T on its own will always eventually result in an output signal on T_1. Sending an input signal on R on its own will produce no output signal (resulting in an input signal pending on R indefinitely due to an incomplete input set), but subsequently or concurrently sending an input signal on T may (due to delay-insensitivity and arbitration) result in different behaviours (an output signal on T_1 with an input signal pending on R, or an output signal on T_0 with no input signals pending). The environment may send input signals repeatedly on T, in between receiving output signals. However, after sending an input signal on R the environment may not send another input signal on R until it receives an output signal on T_0. Otherwise this may result in non-safety or clashing. The set of possibilities is equivalent to those provided by *ATS* starting in S_1, if the environment does not send any input signals which may result in non-safety or clashing.

However consider that it is possible to initialise *ATS* in state S_0 with no input signals pending, such that the environment may send an input signal on T and it is guaranteed that the next output signal will be produced on T_0. It is not possible to provide this functionality using *sATS*, as even placing an initial signal on the wire connected to R does not ensure that the action $(\{R, T\}, \{T_0\}).S_1$ is processed when an input signal is then sent by the environment on T. However, assuming that *ATS* is initialised in state S_1, *sATS* permits the same possible sequences of input and output signals to be sent and received by the environment, and the environment cannot distinguish one module from the other based purely on its sequences of input and output signals. This is irrespective of any initial signals which may be pending,

as they can also be initialised similarly for *sATS*. Hence *sATS* can be considered to have the same "external behaviour" as *ATS*, provided that *ATS* is never assumed to be initialised in S_0.

We now define a module similar to *sATS* but with an additional state, which we call *mATS*.

Example 3.39. We define *mATS* as:

$$\begin{aligned} S_1 &= (\{T\}, \{T_1\}).S_1 + (\{R, T\}, \{T_0\}).S_1 \\ S_0 &= (\{T\}, \{T_0\}).S_1 \end{aligned}$$

mATS is both arb and b-arb.

mATS is the same as *sATS* but allows us to initialise the module in such a way (via state S_0) that the first input signal sent on T by the environment guarantees an output signal on T_0 (similarly to initialising *ATS* in S_0 with no signals pending). After this, it behaves as *sATS*. Hence S_1 and S_0 of *mATS* possess the same external behaviour as S_1 and S_0 of *ATS* respectively. Interestingly, S_0 of *mATS* is not reachable from S_1 as a result of processing actions.

3.4 Conclusion

In this chapter we introduced a new model for describing the behaviour of delay-insensitive modules, called *Set Notation* which more naturally models concurrency and notions of reversibility. We defined important classes of modules in the Set Notation model, such as the non-arb, eq-arb and arb classes. We defined networks of modules in the Set Notation model, along with the execution behaviour of such networks. We defined properties of such networks, such as the non-clashing and safety properties. We investigated several further properties of modules in the Set Notation model which limit the behaviour of the environment in unexpected ways. Examples included the auto-firing or 1-step consistency properties of modules.

Chapter 4

Environments and Implementation

In this chapter we formalise the notion of an environment for a module in the Set Notation model. We give an algorithm for calculating what is referred to as a *maximal environment* of any non-arb module. An algorithm for calculating a maximal environment of any module is then given. Finally, we use this notion to define implementation of a module using a network of modules in Set Notation.

We also note that the algorithms introduced in this chapter are also implemented in the Delay-Insensitive Network Tool Suite program developed in support of this thesis. Furthermore, the environment definitions given in this chapter, produced by these algorithms, were also generated by the corresponding software implementations. Please see Chapter 10 for details on this software.

4.1 Formalisation of an environment

We first formalise the concept of an environment. For reasons related to practical implementation, we assume that an environment for a DI network can only deduce the state of a system via its output behaviour. Hence an environment has no knowledge of a network other than being able to detect signals on output lines which connect to the environment (i.e. the environment cannot observe signals on internal wires connecting modules within a network).

The simplest and most intuitive way to describe an environment's behaviour is using a similar approach to that used to describe a module. The main difference is that we allow input or output sets of actions to be empty (but not both in the same action), as we wish for an environment to be able to non-deterministically "decide" between which sets of signals to send, based on the previous set of received signals, without needing to define multiple actions containing the same sets. It is possible to maintain the restriction, but this typically results in much larger and more complex definitions. We also do not allow two actions in the same state to have identical input and output sets even if the resulting states of the actions are different, as this

particular form of non-determinism is unnecessary. Empty input sets in actions of the environment, combined with the ability to non-deterministically choose between these actions and then move to various states, provide sufficient non-deterministic capability.

Definition 4.1. An *environment* is a 6-tuple (Q', I', O', T', N, sc), where:

1. Q' is a finite set of states,
2. I' is a set of input lines and O' is a set of output lines, where $I' \cap O' = \emptyset$,
3. $T' : Q' \times P[I'] \to Q' \times P[O']$ is a partial map, called the *transition map*, and it assigns an input set in a given state to an output set and a new state.
4. $N = (Q, I, O, T)$ is a module (Definition 3.1 in Chapter 3), such that $|Q'| \geq |Q|$, $I' = O$ and $O' = I$,
5. $sc : Q \to Q_s$ is the *state correspondence function*, which is a bijection between Q and some $Q_s \subseteq Q'$ where $|Q_s| = |Q|$.

We say that an environment $E = (Q', I', O', T', N, sc)$ is a *corresponding environment* of N.

We require that for all $(q, A, q', B) \in T'$ either $A \neq \emptyset$ or $B \neq \emptyset$, but not both. Finally, we require that for all pairs (q, A, q', B) and $(q'', A', q''', B') \in T'$, if $q = q''$ then either $A \neq A'$ or $B \neq B'$.

We also describe environments using the CCS-like notation used to describe modules, defined in the usual way. We require that no two states have identical sets of actions. We also allow at most one state to have no actions. If a state $q \in Q'$ has no actions, it is referred to as a *deadlock state*.

In the following, we assume that the definition of a network (Definition 3.3 in Chapter 3) is modified to allow up to one environment to be connected within a network, with the same connectivity restrictions (each output line of an environment connects to at most one input line of a module, and each input line of an environment connects to at most one output line of a module). Similarly the execution behaviour (Definition 3.5 in Chapter 3) applies to environments in the same way as modules. The definition of safety (Definition 3.11 in Chapter 3) is also assumed to apply to environments.

Definition 4.2. We say that a network composed of a single module $N = (Q, I, O, T)$ in state $q \in Q$ and a corresponding environment $E = (Q', I', O', T', N, sc)$ in state $sc(q) \in Q'$ is *normally-connected* if all output lines of the module connect to the identically named input lines of the environment, all output lines of the environment connect to the identically named input lines of the module, and there are no signals present on any wires.

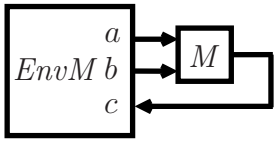


Figure 4.1: Normally-connected network containing *Merge* and a corresponding environment *EnvM*.

Example 4.3. In Figure 4.1 we show the module *Merge* normally-connected to a corresponding environment *EnvM*.

When determining environments for corresponding modules, it makes sense to attempt to define environments which utilise as much functionality of the module as possible. If it is possible for an environment to signal a given input set of a module in a given state, such that safety and non-clashing are guaranteed to hold, then the environment should have the ability to do so. However, we do not want the environment to signal input sets which do not enable at least the possibility of any additional actions to be processed by the module. Similarly, if it is possible for an environment in a given state to receive a given set of output signals from a module, then these should be part of an input set of some action of the environment, otherwise non-safety may occur for the environment. We also do not want the environment to process a partial set of output signals from a module, if it is guaranteed that a superset of these output signals will be sent by the module. We will soon define this notion formally, but first we need to define the notion of a *valid environment*.

Definition 4.4. Consider a normally-connected network composed of a module $N = (Q, I, O, T)$ and a corresponding environment $E = (Q', I', O', T', N, sc)$. We define a *configuration* to be a tuple (q', A, B, q) meaning:

1. E is in state $q' \in Q'$,
2. $A \subseteq I$ is the set of input lines of N (and output lines of E) connected to wires which contain signals,
3. $B \subseteq O$ is the set of output lines of N (and input lines of E) connected to wires which contain signals,
4. N is in state $q \in Q$.

A and B are multisets and the same input or output line may be present multiple times. This corresponds to the wires connected to these lines containing multiple signals.

Informally, a configuration represents a formal notion of a state of a network. However this is limited to networks containing only one module and a corresponding

$$\text{Starting configurations} \quad \frac{q \in Q}{(sc(q), \{\}, \{\}, q) \in V}$$

$$\text{Module input \& output} \quad \frac{(q', A, B, q) \in V \quad (A', B').q'' \in q \quad A' \subseteq A}{(q', A \setminus A', B \cup B', q'') \in V}$$

$$\text{Environment input} \quad \frac{(q', A, B, q) \in V \quad (B', \{\}).q'' \in q' \quad B' \subseteq B}{(q'', A, B \setminus B', q) \in V}$$

$$\text{Environment output} \quad \frac{(q', A, B, q) \in V \quad (\{\}, A').q'' \in q'}{(q'', A \cup A', B, q) \in V}$$

Figure 4.2: Definition of the set of reachable configurations for a module $N = (Q, I, O, T)$ and a corresponding environment $E = (Q', I', O', T', N, sc)$.

environment. It also treats the absorption of signals on input lines and production of signals on output lines by a module as an atomic operation. For the purposes of studying and generating environments this limited notion is sufficient.

Definition 4.5. Given a module $N = (Q, I, O, T)$ and a corresponding environment $E = (Q', I', O', T', N, sc)$, we inductively define the set of *reachable configurations* V for E and N in Figure 4.2.

Informally, each configuration in the set of reachable configurations for some module $N = (Q, I, O, T)$ and environment $E = (Q', I', O', T', N, sc)$ is reached via the execution behaviour, starting from some normally-connected network consisting of N in some state $q \in Q$, and E in $sc(q) \in Q'$.

Definition 4.6. Given a module $N = (Q, I, O, T)$, a corresponding environment $E = (Q', I', O', T', N, sc)$ and the set of reachable configurations V for E and N, we say that V is a *valid interaction* for E and N if for each configuration $(q', A, B, q) \in V$:

1. A and B are sets and not multisets,
2. there is at least one action $(D, C).q'''$ in state $q \in Q$ of N where $A \subseteq D$,
3. if $B \neq \emptyset$, there is an action $(B, \{\}).q''$ in state $q' \in Q'$ of E.

Informally, a valid interaction implies that connecting a module $N = (Q, I, O, T)$ in any state $q \in Q$ to a corresponding environment $E = (Q', I', O', T', N, sc)$ in $sc(q)$ never results in clashing or non-safety for either N or E.

Definition 4.7. Given a module $N = (Q, I, O, T)$, a corresponding environment $E = (Q', I', O', T', N, sc)$, and the set of reachable configurations V for E and N, we say that E is a *valid environment* of N if V is a valid interaction and for all $q' \in Q'$:

1. there exists some configuration $(q', A, B, q) \in V$ for some A, B, q,
2. for all actions $(\{\}, X).q'' \in q'$, there exists some $(q', A, B, q) \in V$ for some A, B, q, and some action $(D, C).q''' \in q$ for some C, q''' such that $D = X \cup A$,
3. for all actions $(B, \{\}).q'' \in q'$, there exists some $(q', A, B, q) \in V$ for some A, q.

Example 4.8. Recall the module *Merge* and corresponding environment *EnvM* from Figure 4.1. We define the behaviour of *EnvM* as:

$$\begin{aligned} EM &= (\{a\}, \{\}).EMa \\ EMa &= (\{\}, \{c\}).EM \end{aligned}$$

where the sc function is defined simply as $sc(M) = EM$. *EnvM* is a valid environment for *Merge*, as the set of reachable configurations $V = \{(EM, \{\}, \{\}, M), (EMa, \{a\}, \{\}, M), (EMa, \{\}, \{c\}, M)\}$ is a valid interaction.

Now that we have the notion of a valid environment for a module, it is important to distinguish those environments which do not utilise the full functionality of a module from those which do wherever possible, such that non-clashing and safety always hold when normally-connecting the module and the environment. We first introduce the notion of an *increase*. In the following we relax the restriction on environments that prohibits multiple states from containing identical sets of actions. We call such environments *pseudo-environments*.

Definition 4.9. We define a *psuedo-environment* similarly to an environment, but allow multiple states to contain identical sets of actions.

Informally, a pseudo-environment may have multiple "functionally equivalent" states. Note that an environment is also a pseudo-environment.

Definition 4.10. Given two pseudo-environments $E = (Q, I, O, T, N, sc)$ and $E' = (Q', I, O, T', N, sc)$ which both correspond to the same module N, we say that E' is an *increase* of E if $Q \subseteq Q'$ and $T \subset T'$.

Informally, a pseudo-environment E' is an increase of a pseudo-environment E if it is possible to get E' by adding at least one action, and any number of new states to E.

In Figure 4.3, we give the *state-merge* algorithm. Informally, the algorithm simply removes each state which contains an identical set of actions to some other

Require: pseudo-environment $E = (Q, I, O, T, N, sc)$
1: **while** there are multiple states with equivalent sets of actions **do**
2: **for** every $q_1 \in Q$ **do**
3: **if** there exists some $q_2 \in Q$ such that $q_1 \neq q_2$ and q_1 and q_2 contain identical sets of actions **then**
4: **for** every $(q, A, q', B) \in T$ where $q = q_1$ **do**
5: remove (q, A, q', B) from T
6: **end for**
7: **for** every $(q, A, q', B) \in T$ where $q' = q_1$ **do**
8: remove (q, A, q', B) from T and add (q, A, q'', B) to T, where $q'' = q_2$, if it does not already exist.
9: **end for**
10: **end if**
11: **end for**
12: **end while**
Ensure: environment $E = (Q', I, O, T', N, sc)$ where $|Q'| \leq |Q|$ and $|T'| \leq |T|$.

Figure 4.3: State-merge algorithm for a pseudo-environment *state-merge*(E).

existing state, and then redirects transitions appropriately. Hence any pseudo-environment definition is converted to an environment definition.

Definition 4.11. Let $E = (Q, I, O, T, N, sc)$ be an environment. We say that an environment E' is an *valid-increase* of E, if $E' \neq E$ and there exists some psuedo-environment E'' such that E'' is an increase of E, and E' can be found by running the state-merge algorithm on E''.

Informally, an environment E' is a valid-increase of an environment E, if it is possible to add actions or states to E, merging state definitions when equivalent, to get E'.

Using this notion of a valid-increase, can now introduce the notions of *sub-maximality* and *maximality* of an environment.

Definition 4.12. Let E be a *valid environment* of some module N. E is *sub-maximal* for N if there exists some E' such that E' is a valid environment of N, and E' is a valid-increase of E. We say that E is a *maximal environment* of N if it is not sub-maximal for N.

Example 4.13. The environment *EnvM* defined in Example 4.8 is sub-maximal for *Merge*, as it is possible to add the action $(\{b\}, \{\}).EMa$ to the state EM, and the resulting definition (which we call *EnvM'*) is a valid environment, as shown by the valid interaction $V = \{(EM, \{\}, \{\}, M), (EMa, \{a\}, \{\}, M), (EMa, \{\}, \{c\}, M), (EMa, \{b\}, \{\}, M)\}$.

Example 4.14. The environment *EnvM'* discussed in Example 4.13, and defined as:

$$\begin{aligned} EM &= (\{a\}, \{\}).EMa + (\{b\}, \{\}).EMa \\ EMa &= (\{\}, \{c\}).EM \end{aligned}$$

such that $sc(M) = EM$ is a valid environment of *Merge* and is not sub-maximal. Therefore it is a maximal environment of *Merge*.

We note that there are an infinite number of maximal environments for any module. Informally however, the nature of the maximal environment is such that it exhausts as much of the functionality of the module as possible. This is achieved by sending as many output signals as possible which present at least the possibility of additional actions being processed, such that safety and non-clashing are guaranteed to hold. The possibilities of which output signals may be sent are decided deterministically by the environment, based on the sets of input signals it receives.

The previous paragraph suggests that, given some module N, all maximal environments of N are semantically equivalent, even if their definitions are not the same. However we have been unable to prove this result formally. In the rest of this thesis, we assume that this is the case.

A maximal environment of a module can often be determined by enumerating all possible actions for the environment such that safety and non-clashing are guaranteed to hold when normally-connecting the environment to the module.

Example 4.15. Consider the definition of *sATS* from Section 3. If the definition of the module is carefully considered, through enumeration of all possibilities for input/output it can be deduced that a maximal environment *EnvsATS* of *sATS* can be defined as follows:

$$\begin{aligned} ES_1 &= (\{\}, \{T\}).ES_1T + (\{\}, \{R, T\}).ES_1RT \\ ES_1T &= (\{\}, \{R\}).ES_1RT + (\{T_1\}, \{\}).ES_1 \\ ES_1RT &= (\{T_0\}, \{\}).ES_1 + (\{T_1\}, \{\}.ES_1R \\ ES_1R &= (\{\}, \{T\}).ES_1RT \end{aligned}$$

where ES_1 corresponds to the module state S_1. After signalling R, it is clear that the environment cannot send another input signal on R until an output signal is received on T_0. Note that a single input signal on R is not sent by the environment in ES_1, as an input signal pending on R of the module in state S_1 is not sufficient to cause the module to process any actions. After sending input signals on $\{R, T\}$, the environment waits for an output signal on T_0 or T_1, and uses this to deduce whether there is an input signal pending on R, and therefore whether it is safe to send an input signal on T, or to return to the original state and have a choice between sending input signals on either T or $\{R, T\}$ again.

Enumerating all possible behaviours in order to determine a maximal environment is much more difficult when modules are auto-firing or not 1-step consistent. Recall the informal notion of inconsistency (Section 3.2 in Chapter 3), which con-

cerns the inability of an environment to deduce the current state of a module and the number of signals on wires connected to its input or output lines.

Example 4.16. Consider the following module N_6 which is not 1-step consistent.

$$S_0 = (\{a,b\},\{x,y\}).S_0 + (\{a,b\},\{x\}).S_0$$

A possible maximal environment for N_6 is defined as:

$$\begin{aligned} ES_0 &= (\{\},\{a,b\}).ES_0ab \\ ES_0ab &= (\{x\},\{\}).ES_0abx + (\{x,y\},\{\}).ES_0 \\ ES_0abx &= (\{y\},\{\}).ES_0 \end{aligned}$$

where ES_0 corresponds with S_0. Consider that the environment may signal $\{a,b\}$ when the module is in S_0, according to the definition of ES_0. If an output signal is received on only x, it is not possible to deduce whether an output signal is also travelling along the wire connected to y. As a result, at this stage there are no input lines of the module that the environment may signal without risking the network becoming clashing. This forces the environment to wait for an output signal from y before sending any more input signals, which may never arrive if the second action of S_0 was processed by the module. Here, inconsistency may cause a permanent deadlock between the module and its environment.

Inconsistency can occur even if a module is 1-step consistent.

Example 4.17. Consider the following module N_7 which is stable, 1-step-consistent and auto-firing.

$$\begin{aligned} S_0 &= (\{a,b\},\{x\}).S_0 + (\{a\},\{y\}).S_1 \\ S_1 &= (\{b\},\{x\}).S_0 \end{aligned}$$

The terminating autopath $(\{a\},\{y\}).S_1, (\{b\},\{x\}).S_0$ may fire if $\{a,b\}$ is signalled by the environment when the module is in S_0. In such a situation, output signals will be sent on y and x in that order. However due to delay-insensitivity, it is possible that the output signal on x arrives at the environment prior to the output signal on y, and the environment (similarly to the previous example) cannot deduce whether or not an output signal was sent on y. Hence, it is not possible for the environment to distinguish this situation from the one where the action $(\{a,b\},\{x\}).S_0$ was instead processed by the module. As there are no other input lines which it may signal, the environment cannot proceed until receiving an output signal on y. As a result, if $(\{a,b\},\{x\}).S_0$ was instead processed by the module, the network permanently deadlocks.

We now infer some results based on the new formal notion of an environment.

Proposition 4.18. Normally-connecting any module $N = (Q, I, O, T)$ in state $q \in Q$ to any maximal environment $E = (Q', I', O', T', N, sc)$ of N in state $sc(q)$ gives rise to a network which is safe and non-clashing.

Proof. If no initial signals are placed on any wires, then the environment E is responsible for initiating and dictating all interaction with the module N. The nature of a maximal environment is such that no signals are sent which cause any possibility of non-safety or clashing on any wires between E and N. The only wires in the network are those which are connected between E and N. □

Theorem 4.19. Normally-connecting a stable, non-auto-firing, and 1-step consistent module $N = (Q, I, O, T)$ in state $q \in Q$ to a maximal environment $E = (Q', I', O', T', N, sc)$ of N in state $sc(q)$, gives rise to a network which is non-deadlocking.

Proof. If a module is non-auto-firing, and is 1-step consistent, then it is always possible to deduce which set of input signals was processed by the module based purely on its set of output signals. Consequently, the environment can wait until all output signals are received from the module before proceeding to send any more input signals. Furthermore, if a module is also stable, by implication it is guaranteed that there is always at least one action defined in each state whose input set is available for the environment to signal. Hence the environment can send and receive signals to the module indefinitely without any possibility of inconsistency or deadlock. □

4.2 Generating maximal environments

We investigate the generation of maximal environments.

Firstly, in Figure 4.4 we give an algorithm for generating a maximal environment for any non-arb module.

We now show a small example of an environment generated by the algorithm.

Example 4.20. Consider the module N_8 defined as:

$$\begin{aligned} S_0 &= (\{a, b, c\}, \{g, e\}).S_1 + (\{a, c, d\}, \{e, f\}).S_0 \\ S_1 &= (\{a\}, \{g\}).S_0 \end{aligned}$$

Require: module $N = (Q, I, O, T)$ which is non-arb
1: create pseudo-environment $E = (Q', I', O', T', N, sc)$ where Q' is any set the same size as Q, sc is any bijection that maps Q to the (current) elements of Q', $I' = O$, $O' = I$, and T' is empty
2: **for** every $q_1 \in Q$ **do**
3: **for** every $(A, B).q_1'$ in q_1 **do**
4: add q_2 to Q' where q_2 is fresh and create action $(\{\}, A).q_2$ in $sc(q_1)$
5: create action $(B, \{\}).q_3$ in q_2 where $q_3 = sc(q_1')$
6: **end for**
7: **end for**
8: run *state-merge*(E)
Ensure: maximal environment $E = (Q', I', O', T', N, sc)$ for N

Figure 4.4: Generating a maximal environment for a non-arb module.

If this module is input to the algorithm, it yields the following environment.

$$\begin{aligned}
ES_0 &= (\{\}, \{a, b, c\}).ES_0abc + (\{\}, \{a, c, d\}).ES_0acd \\
ES_1 &= (\{\}, \{a\}).ES_1a \\
ES_0abc &= (\{g, e\}, \{\}).ES_1 \\
ES_0acd &= (\{e, f\}, \{\}).ES_0 \\
ES_1a &= (\{g\}, \{\}).ES_0
\end{aligned}$$

where $sc(S_0) = ES_0$ and $sc(S_1) = ES_1$.

Proposition 4.21. For any given non-arb module N, the environment given by the algorithm in Figure 4.4 with N as input is a maximal environment of N.

Proof. All non-arb modules are 1-step-consistent, non-auto-firing and stable. Hence signalling an input set leads to exactly one set of output signals being produced with no remaining input signals pending. Therefore all actions in a given state are trivially able to have their input sets signalled by the environment. Due to the deterministic behaviour of non-arb modules, the environment simply needs to signal any input set of any action of the current state, then wait for the corresponding set of output signals before proceeding. The algorithm produces this behaviour for the resulting environment. □

A more complex algorithm is needed to generate a maximal environment for any Set Notation module. Such an algorithm would need to factor in the possibility of auto-firing, and more crucially the possibility of inconsistency. This means generating an environment which can:

- anticipate the set of output signals which may or may not be travelling to the environment and could arrive at any point,

- deduce which input lines are safe to signal after considering the set of states that the module might be in and which wires may already contain signals.

Before introducing the algorithm, we first define an important auxiliary function.

Definition 4.22. Given a terminating autopath $(A_0, B_0).q_0, \ldots, (A_n, B_n).q_n$, we define the set of *accumulated outputs*, given by $accum((A_0, B_0).q_0, \ldots, (A_n, B_n).q_n)$ to be the multiset $B = \bigcup_{i=0}^{n} B_i$.

We utilise a limited version of the configuration data structure (q', A, B, q) from the previous section. However, we do not make use of the environment's state value, and leave it undefined. Hence when we refer to a configuration in the following, we use the tuple (A, B, q).

Definition 4.23. Given a module $N = (Q, I, O, T)$, we define an *uncertainty* to be a finite set of configurations U, where for each $(A, B, q) \in U$, $A \subseteq I, B \subseteq O$ and $q \in Q$.

An uncertainty U represents a collection of states that the module and the wires connected to its input/output lines may be in at a particular point in time. Each state of an environment in the remainder of this section corresponds to an uncertainty. Hence given a module $N = (Q, I, O, T)$, and a corresponding environment $E = (Q', I', O', T', N, sc)$, each $q' \in Q'$ of the environment corresponds to some uncertainty U. Informally, this represents that in state q', the environment is unsure as to which configuration $(A, B, q) \in U$ accurately represents the state of the network. This notion is used to calculate which set of behaviours are available to an environment in each state.

The algorithm for generating a maximal environment for any Set Notation module N is broken into two parts, a parent function and the main recursive function. To improve readability of the algorithm, we assume an implicit correspondence between states of the environment and uncertainties. However we note that it is possible to define an explicit partial bijection which maps states to uncertainties and is initially empty, but adds new mappings as new environment states and uncertainties are created by the algorithm.

We give the parent part of the algorithm in Figure 4.5. This creates the input and output lines of the environment, and the "starting" states which correspond directly to states of the module (via sc). It then calls the recursive part of the algorithm $recurseUncertain(N, E, sc(q), U)$ to generate environment actions for each possible "starting" state $sc(q)$ corresponding to each q of the module N.

The main recursive part of the algorithm is separated into two parts for readability. We give the first half in Figure 4.6 which builds all possible "output" actions

Require: module $N = (Q, I, O, T)$
1: create pseudo-environment $E = (Q', I', O', T', N, sc)$ where Q' is any set the same size as Q, sc is any bijection that maps Q to the (current) elements of Q', $I' = O$, $O' = I$, and T' is empty
2: **for** every state $q \in Q$ **do**
3: create a new uncertainty U which contains a single configuration $(\{\}, \{\}, q)$
4: run $recurseUncertain(N, E, sc(q), U)$ if not yet done so for $sc(q)$ and U
5: **end for**
6: run *state-merge*(E)
Ensure: maximal environment $E = (Q', I', O', T', N, sc)$ for module $N = (Q, I, O, T)$

Figure 4.5: Generating a maximal environment for any module.

for the environment (corresponding to sending input signals to the module) for a given uncertainty U. Each action created by this part of the algorithm is guaranteed to enable the processing of at least one new action of the module for at least one configuration in U, without causing clashing or non-safety of the module, regardless of the configuration in U which accurately represents the state of the network. It is possible that the algorithm creates no "output" actions for the environment, in which case the environment cannot currently signal any input lines of the module, such that it enables at the least the possibility of additional actions to be processed by the module.

The second half of the recursive part is given in Figure 4.7 which builds all possible "input" actions for the environment (corresponding to receiving output signals from the module) for a given uncertainty U. The algorithm considers that for any $(A, B, q) \in U$, output signals corresponding to B may be received by the environment, possibly in combination with new output signals which may be produced via the processing of several actions by the module (for all possible prefixes of all $p \in autoPaths(A, q)$). The algorithm also calculates the new uncertainty U' produced in response to receiving a particular set of output signals from the module. This is done by deciding whether each existing configuration in the old uncertainty should be removed from the new uncertainty, and whether any new configurations should be added. An existing configuration is removed if the input set of the current action we are building is not a subset of the output signals which would arrive if the existing configuration accurately represented the state of the network. In this scenario, the environment can rule out the existing configuration as an accurate representation of the state of the network. A new configuration is added if the input set of the current action we are building is a subset of output signals that would arrive if such a configuration accurately represented the state of the network. In this scenario, the environment cannot rule out such a configuration from being an accurate representation of the state of the network.

Require: module $N = (Q, I, O, T)$, pseudo-environment $E = (Q', I', O', T', N, sc)$, environment state $q_e \in Q'$ and corresponding uncertainty U

1: create the empty set SI of type $P[I]$
2: **for** every configuration $(A_M, B_M, q_M) \in U$ **do**
3: **for** every action $(A_P, B_P).q_P \in q_M$ of N **do**
4: **if** $A_M \subset A_P$ **then** add $A_P \setminus A_M$ to SI **end if**
5: **end for**
6: **end for**
7: **for** every input set $A_I \in SI$ **do**
8: **for** every configuration $(A_M, B_M, q_M) \in U$ **do**
9: **if** there is not some action $(A_P, B_P).q_P \in q_M$, such that $(A_I \cup A_M) \subseteq A_P$ **then** remove A_I from SI and **break** **end if**
10: **if** for some prefix $(A_0, B_0).q_0, \ldots, (A_j, B_j).q_j$ where $(0 \leq j \leq n)$ of some $(A_0, B_0).q_0, \ldots, (A_n, B_n).q_n \in autoPath(A_I \cup A_M, q_M)$, where $A_S = \bigcup_{k=0}^{j} A_k$, there is not some action $(A_P, B_P).q_P \in q_j$ where $((A_I \cup A_M) \setminus A_S) \subseteq A_P$ (i.e. non-safety occurs for N in state q_j) **then** remove A_I from SI and **break** **end if**
11: **if** for some $p = (A_0, B_0).q_0, \ldots, (A_n, B_n).q_n \in autoPath(A_I \cup A_M, q_M)$, $accum(p)$ is not a set but is a multiset or $(accum(p) \cap B_M) \neq \emptyset$, (i.e. there is an auto-clash or the output signals overlap with those already sent by the module) **then** remove A_I from SI and **break** **end if**
12: **end for**
13: **end for**
14: **for** every set $A_I \in SI$ **do**
15: create a new uncertainty U' which is a copy of U
16: **for** every configuration $(A_M, B_M, q_M) \in U'$ **do** add all elements of A_I to set A_M **end for**
17: create state $q'_e \in Q'$ if it doesn't exist, where q'_e corresponds with U'
18: create action $(\{\}, A_I).q'_e$ in state q_e if it doesn't exist
19: run $recurseUncertain(M, E, q'_e, U')$ if not yet done so for q'_e and U'
20: **end for**

Figure 4.6: First half of recursive part of maximal environment algorithm $recurseUncertain(M, E, q_e, U)$.

21: **for** every configuration $(A_1, B_1, q_1) \in U$ **do**
22: let the set $AP_1 = autoPaths(A_1, q_1)$
23: **for** every autopath $p_1 = (A_{1,0}, B_{1,0}).q_{1,0}, \ldots, (A_{1,n}, B_{1,n}).q_{1,n} \in AP_1$, plus once let $p_1 = \emptyset$ **do**
24: **for** every prefix $(A_{1,0}, B_{1,0}).q_{1,0}, \ldots, (A_{1,i}, B_{1,i}).q_{1,i}$ (with $0 \leq i \leq n$) of p_1, (or once if $p_1 = \emptyset$) **do**
25: **if** $p_1 = \emptyset$ **then** let $q' = q_1$, let $A'_1 = A_1$, let $B'_1 = B_1$
26: **else** let $q' = q_{1,i}$, let $A'_1 = A_1 \setminus A''_1$, where $A''_1 = \bigcup_{k=0}^{i} A_{1,k}$, let $B'_1 = B_1 \cup B''_1$ where $B''_1 = \bigcup_{k=0}^{i} B_{1,k}$ **end if**
27: **if** some action $(B'_1, \{\}).q'_e$ already exists in q_e, for some q'_e **then** then skip to the next prefix **end if**
28: create new empty uncertainty U' and add $(A'_1, \{\}, q')$ to U'
29: **for** every configuration $(A_2, B_2, q_2) \in U$ **do**
30: let $AP_2 = autoPaths(A_2, q_2)$
31: **for** every autopath $p_2 = (A_{2,0}, B_{2,0}).q_{2,0}, \ldots, (A_{2,m}, B_{2,m}).q_{2,m} \in AP_2$, plus once let $p_2 = \emptyset$ **do**
32: **for** every prefix $(A_{2,0}, B_{2,0}).q_{2,0}, \ldots, (A_{2,l}, B_{2,l}).q_{2,l}$ (with $0 \leq l \leq m$) of p_2, (or once if $p_2 = \emptyset$) **do**
33: **if** $(A_1, B_1, q_1) = (A_2, B_2, q_2)$, $p_1 = p_2$ and $l \leq i$ **then** skip to next prefix of p_2 **end if**
34: **if** $p_2 = \emptyset$ **then** let $q'' = q_2$, let $A'_2 = A_2$, let $B'_2 = B_2$
35: **else** let $q'' = q_{2,l}$, let $A'_2 = A_2 \setminus A''_2$, where $A''_2 = \bigcup_{k=0}^{l} A_{2,k}$, let $B'_2 = B_2 \cup B''_2$ where $B''_2 = \bigcup_{k=0}^{l} B_{2,k}$ **end if**
36: **if** $B'_1 \subseteq B'_2$ **then** add new configuration $(A'_2, B'_2 \setminus B'_1, q'')$ to U'
37: **end if**
38: **end for**
39: **end for**
40: **end for**
41: create state $q'_e \in Q'$ if it doesn't exist, where q'_e corresponds with U'
42: create action $(B'_1, \{\}).q'_e$ in state q_e if it doesn't exist
43: run $recurseUncertain(N, E, q'_e, U')$ if not yet done so for q'_e and U'
44: **end for**
45: **end for**
46: **end for**
Ensure: maximal environment E behaviour for module N starting from U

Figure 4.7: Second half of recursive part of maximal environment algorithm $recurseUncertain(M, E, q_e, U)$.

We give an example of a module which possesses multiple properties to demonstrate the effectiveness of the algorithm.

Example 4.24. Consider the following module N_9 which is unstable, not 1-step consistent, and auto-clashing:

$$\begin{aligned} S_0 &= (\{a,b\},\{x\}).S_0 + (\{b\},\{x\}).S_1 + (\{b,c\},\{y\}).S_1 + (\{c\},\{z\}).S_1 \\ S_1 &= (\{a\},\{z\}).S_0 + (\{c\},\{x\}).S_0 \end{aligned}$$

Inputting N_9 to the algorithm in Figure 4.5 will yield the following environment. Note that due to the state-merge algorithm, multiple states (corresponding to different uncertainties) have been merged together, and as a result some states may correspond to more than just the uncertainty indicated by its name.

$$\begin{aligned} ES_0 &= (\{\},\{a,b\}).ES_0ab + (\{\},\{b\}).ES_0b + (\{\},\{c\}).ES_0c \\ ES_1 &= (\{\},\{c\}).ES_1c + (\{\},\{a\}).ES_0S_1aS_0z \\ ES_0ab &= (\{x\},\{\}).ES_0S_1aS_0z + (\{x,z\},\{\}).ES_0 \\ ES_0S_1aS_0z &= (\{z\},\{\}).ES_0 \\ ES_0b &= (\{\},\{a\}).ES_0ab + (\{x\},\{\}).ES_1 \\ ES_1c &= (\{x\},\{\}).ES_0 \\ ES_0c &= (\{z\},\{\}).ES_1 \end{aligned}$$

The environment states ES_0 and ES_1 correspond to the module states S_0 and S_1 respectively. Note that in ES_0, the environment cannot signal $\{b,c\}$, as this may cause an auto-clash on the wire connected to the output line x, due to the autopath $(\{b\},\{x\}).S_1, (\{c\},\{x\}).S_0$. Consider the environment state $ES_0S_1aS_0z$. This corresponds to the environment being uncertain as to whether the module is in S_0 with no input signals pending, S_1 with a signal pending on a, or S_0 with an output signal travelling along the wire connected to z. As a result, the environment cannot send any more input signals to the module, and it needs to wait for an output signal on z in order to determine its next move. If an output signal then arrives from z, the environment deduces that the module must now be in S_0 with no signals pending, regardless of which configuration previously represented the state of the network.

Theorem 4.25. For any module N, the environment given by the algorithm in Figure 4.5 with N as input is a maximal environment of N.

Proof. Using the notion of an uncertainty, the algorithm considers all possibilities concerning which signals may be on input or output lines and what state the module may be in (through one configuration in the uncertainty for each possibility) during each of the environment's states. It also factors in whether a module is auto-firing,

auto-clashing or stable, all of which are the limiting factors (other than the set of currently pending input signals) as to whether individual actions are able to be signalled in a given configuration. Using these pieces of information it guarantees that it does not send any signals which have a possibility of causing non-safety or clashing, but sends input signals corresponding to all possible actions in all configurations which are otherwise guaranteed to be "unproblematic". It also uses this information to accommodate all possible arrivals of output signals from the module in all possible configurations, as well as any output signals which may be produced by the module at any time prior to the environment choosing to send any more input signals. It does this by factoring in the potential firing of any autopaths, for any configuration in the current uncertainty. As a result, safety of the environment is guaranteed to hold. □

We note that, as described in the beginning of this chapter, the algorithm in Figure 4.5 is implemented in the Delay-Insensitive Network Tool Suite program developed in support of this thesis. Please see Chapter 10 for details on this software.

4.3 Implementation and universality

Using the new formal notion of a maximal environment, it is now possible to define notions of implementation and universality in the Set Notation model. These correspond to Definitions 2.14 and 2.15 of the sequential machine model respectively (see Chapter 2), but are more precise due to a formal notion of an environment.

Definition 4.26. Consider a module N and a network S. We define an *environment mapping em* between N and S to be an injective function which maps every input line of N to exactly one unconnected input line of some module in S, and every output line of N to exactly one unconnected output line of some module in S.

Informally, given some network S which is intended to implement some module N, an environment mapping is a way of formally expressing which input and output lines of modules in S correspond to input and output lines of N respectively.

Definition 4.27. Consider a module $N = (Q, I, O, T)$, a network of modules S and an environment mapping em between N and S. We say that N and S are *indistinguishable in state* $q \in Q$ *via* em if, for some maximal environment $E = (Q', I', O', T', N, sc)$ of N, the following conditions hold:

1. Connecting E in state $sc(q)$ to S according to em results in a network which is safe and non-clashing,

2. For all possible sequences of actions $(A_1, B_1).q_1, (A_2, B_2).q_2 \ldots$ of E which may be processed when normally-connecting N in state $q \in Q$ to E in state $sc(q)$ (where $(A_1, B_1).q_1 \in sc(q)$), it is possible for the same sequence of actions to be processed when connecting E in state $sc(q)$ to S according to em, and vice versa.

Informally, this says that the series of operations which may be performed by an environment E starting in a particular state $sc(q)$ when normally-connected to a module N starting in state q, does not change when instead connecting the environment starting in $sc(q)$ to a network S which "implements" the module N starting in state q.

Indistinguishability is a more formal notion of external behaviour (Definition 2.14 in Chapter 2) described in the sequential machine model. We note that indistinguishability can also be used to compare two modules, rather than just a module and a network of multiple modules.

Example 4.28. Recall the definitions of *sATS* and *mATS* (Examples 3.24 and 3.39 in Chapter 3). Note that the definitions of S_1 for both modules are the same. Therefore trivially, an *sATS* can always be replaced by an *mATS* in state S_1, and the behaviour of the overall network is unaffected. More formally, it is the case that the module *sATS* and the network containing a single *mATS* are indistinguishable in state S_1 via the mapping which maps all input and output lines of *sATS* to the identically-named input and output lines of *mATS*.

Note however that this definition requires a specific starting state of the module, similarly to the notion of a realisation in the sequential machine model (Definition 2.14 in Chapter 2). This is not general enough to cover implementation of a module in all states, as with some modules, not all states are reachable from all other states as a result of processing actions. An example is *mATS*, where S_0 is not reachable from S_1.

Therefore we need a more general definition which says that a module is implemented by a network with a fixed structure regardless of its starting state.

Definition 4.29. Consider a module $N = (Q, I, O, T)$, and a network of modules S. We say that S *implements* N if:

1. there exists some environmental mapping em between N and S,
2. for some $q \in Q$, N and S are indistinguishable in state q via em,
3. for each $q' \in Q$, there exists some S' which is a state variation of S, and N and S' are indistinguishable in state q' via em.

Informally, a network implements a module if for each possible state q of the module, the network can have any of its modules' internal states modified, and signals can be added or removed from wires, such that the resulting network is indistinguishable from the module in state q. The given network must also be indistinguishable from the module in at least one of the module's states.

Example 4.30. Following on from Example 4.28, the network S containing a single *mATS* module in state S_1, implements *sATS*, as there is only one state (S_1) of *sATS*, and *sATS* and S are indistinguishable in S_1 via the mapping which maps all input and output lines of *mATS* to the identically-named input and output lines of *sATS*.

Note however that the reverse is not true, as there does not exist a state variation S' of the network containing a single *sATS* such that *mATS* and S' are indistinguishable in state S_0 via the mapping which maps all input and output lines of *sATS* to the identically-named input and output lines of *mATS*. Informally, this is impossible because *sATS* is incapable of simulating the functionality provided by *mATS* in state S_0 (Section 3.3 in Chapter 3).

Definition 4.31. A set of modules X is *universal* for some class of modules Y, if any module in Y can be implemented (according to Definition 4.29) by a network containing only modules in X.

The terms *implementation* and *universal* in the context of the Set Notation model should not be confused with the terms *realisation* and *universal* given in Definitions 2.14 and 2.15 in the sequential machine model. The use of the term *realisation* in the sequential machine model corresponds more directly with our notion of *indistinguishability*. We now finish this chapter by highlighting an interesting relationship between modules which share the same maximal environment.

We define a third module similar to *sATS* and *mATS*, which we will call *fATS*.

Example 4.32. We define *fATS* as:

$$\begin{aligned} S_1 &= (\{T\},\{T_1\}).S_1 + (\{R,T\},\{T_0\}).S_1 + (\{R,T\},\{T_1\}).S_0 \\ S_0 &= (\{T\},\{T_0\}).S_1 \end{aligned}$$

fATS is both arb and b-arb.

fATS is similar to *mATS* except there is the additional action $(\{R,T\},\{T_1\}).S_0$ in S_1. As a result, unlike with *mATS*, S_0 is reachable from S_1 in *fATS* as a result of processing actions.

Example 4.33. A possible maximal environment *EnvfATS* =

$(Q', I', O', T', fATS, sc)$ of *fATS*, given by the algorithm in Figure 4.5, is defined as:

$$\begin{aligned}
ES_1 &= (\{\}, \{T\}).ES_1T + (\{\}, \{R, T\}).ES_1TR \\
ES_0 &= (\{\}, \{T\}).ES_0T \\
ES_1T &= (\{\}, \{R\}).ES_1TR + (\{T_1\}, \{\}).ES_1 \\
ES_1TR &= (\{T_1\}, \{\}).ES_1RS_0 + (\{T_0\}, \{\}).ES_1 \\
ES_1RS_0 &= (\{\}, \{T\}).ES_1TR \\
ES_0T &= (\{T_0\}, \{\}).ES_1
\end{aligned}$$

where $sc(S_1) = ES_1$ and $sc(S_0) = ES_0$. Note that *EnvfATS* is also a maximal environment of *mATS*, as given by the algorithm in Figure 4.5 (though state names may differ as a result of the algorithm), with *sc* also defined identically.

Informally, the fact that *EnvfATS* is a maximal environment of both *fATS* and *mATS* means that *EnvfATS* cannot distinguish one module from the other. It is also the case that *mATS* is implemented by the network containing a single *fATS* module, and *fATS* is implemented by the network containing a single *mATS* module. This highlights the following result.

Proposition 4.34. If $E_1 = (Q_1, I_1, O_1, T_1, N, sc_1)$ is a maximal environment of some module $N = (Q, I, O, T)$ and $E_2 = (Q_2, I_2, O_2, T_2, N', sc_2)$ is a maximal environment of some module $N' = (Q', I', O', T')$ where $N \neq N'$, $Q_1 = Q_2$, $I_1 = I_2$, $O_1 = O_2$ and $T_1 = T_2$, then N is implemented by the network containing a single N', and N' is implemented by the network containing a single N.

Proof. As the CCS-like definitions of E_1 and E_2 are the same, then for all states $q_1 \in Q$, there must exist some $q_2 \in Q'$ such that for all possible sequences of actions processed by the environment E_1 when normally-connecting E_1 in $sc_1(q_1)$ to N in q_1, it is possible for the same sequence of actions to be processed when connecting E_1 in $sc_1(q_1)$ to N' in q_2 according to the mapping which maps all input and output lines of N to the identically-named input and output lines of N'. Similarly, for all states $q_2 \in Q'$, there exists some $q_1 \in Q$ such that for all possible sequences of actions processed by the environment E_2 when normally-connecting E_2 in $sc_2(q_2)$ to N' in q_2, it is possible for the same sequence of actions to be processed when connecting E_2 in $sc_2(q_2)$ to N in q_1, according to the mapping which maps all input and output lines of N' to the identically-named input and output lines of N. By Definitions 4.27 and 4.29, this means that N implements N' and vice versa. □

4.4 Conclusion

In this chapter we formalised the notion of an environment for a module in the Set Notation model. We gave an algorithm for calculating what is referred to as a *maximal environment* of any non-arb module. An algorithm for calculating a maximal environment of any module was then given. Finally, we used this notion to define implementation of a module using a network of modules in Set Notation.

Chapter 5

Correspondences between models

In this chapter we compare the sequential machine model for DI networks with the new Set Notation model. We introduce an extension to the sequential machine model called the *ND sequential machine* model. We establish limited correspondences between three models; the sequential machine model, the ND sequential machine model, and the new Set Notation model. We prove universality results for the ND sequential machine model. We give algorithms for converting modules which satisfy certain conditions in the sequential machine model or the ND sequential machine model, to corresponding definitions in the Set Notation model, and vice versa.

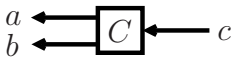
We also note that the algorithms introduced in this chapter are also implemented in the Delay-Insensitive Network Tool Suite program developed in support of this thesis. Furthermore, the module definitions given in this chapter, produced by these algorithms, were also generated by the corresponding software implementations. Please see Chapter 10 for details on this software.

5.1 Non-deterministic sequential machines

We begin by noting that the definitions of modules in the Set Notation model are more expressive than in the sequential machine model, and as a result it is possible to define modules in Set Notation which do not have sequential machine counterparts.

Example 5.1. In Figure 5.1 we introduce the module *Choice* which simulates a non-deterministic binary choice, given a single input signal on c. Note that *Choice* is the inverse of *Merge*.

Recall that, as a result of the sequential machine model requiring that f and g are partial functions, modules such as ATS exhibit non-deterministic behaviour by utilising multiple concurrent input signals combined with non-trivial arbitration (Section 2.5 in Chapter 2). Hence there does not exist some sequential machine



$$C \;=\; (\{c\},\{a\}).C + (\{c\},\{b\}).C$$

Figure 5.1: *Choice* module.

module that has the same behaviour as *Choice* because there is only a single input line (c).

Therefore, in order to identify a correspondence between Set Notation modules and sequential machine modules, we must first generalise the sequential machine model such that modules like *Choice* can be defined.

Definition 5.2. We define the *non-deterministic sequential machine* (*ND sequential machine*) model the same as the sequential machine model (see Chapter 2) with all associated conditions and definitions, except:

1. f and g components of modules do not have to be partial functions, and can be partial maps,

2. A module is serial iff it fits the conditions in Definition 2.16 and f and g are partial functions.

3. A module is parallel iff it fits the conditions in Definition 2.17 and f and g are partial functions.

We call a module a *ND sequential machine* if f and g are partial maps but not partial functions, and simply a *sequential machine* if f and g are partial functions. We refer to the class of all modules in this model as the *class of (ND) sequential machines*.

The use of the terms *realisation* and *universal* in the context of the ND sequential machine model are therefore assumed to refer to Definitions 2.14 and 2.15 in Chapter 2, and should not be confused with the terms *implementation* or *universal* in the context of our Set Notation model as given in Definitions 4.29 and 4.31 in Chapter 4.

The behaviour of modules and networks in the ND sequential machine model is therefore the same as the sequential machine model with one exception. When a module in some state q processes an input signal on some input line a, the set of output signals and resulting state change of the module is decided non-deterministically by selecting randomly any action $(a, B).q' \in q$, for some B, q'.

The second condition in Definition 5.2 requires that modules are only considered serial if they are sequential machines. Informally, this ensures that the various

notions of serial modules across the Set Notation, ND sequential machine and sequential machine models remain consistent and all refer to the same set of module behaviours. Note that this no longer implies that a module is parallel if it is not serial. Instead, this results in a third class of modules in the ND sequential machine model, which are neither serial nor parallel.

Example 5.3. An ND sequential machine counterpart to the Set Notation module *Choice* is found by simply removing the set brackets from the input sets in the CCS-like definition of *Choice* given in Figure 5.1. We also call the resulting module *Choice*.

The ND sequential machine *Choice* is not a serial module as it does not fit the requirement that it is a sequential machine.

We note that it is possible to create arbitrary n-way *Choice* trees by connecting multiple instances of the same module together in a tree formation, similarly to how we create *Merge* and *Fork* trees (see Chapter 2). In the same way, we depict an n-way *Choice* tree using the same symbol as *Choice*, but with extra output lines present on the image.

5.1.1 Universal sets

We will now prove some universality results in the ND sequential machine model, which will be later utilised in Chapter 6 to prove universality results in the Set Notation model. Recall Keller's construction method (Figure 2.3 in Chapter 2), which can be used to realise any sequential machine module (including modules which are not parallel). We now describe how to extend this method to allow the realisation of the class of ND sequential machines.

Recall that for any sequential machine $N = (Q, I, O, f, g, A)$ there is a corresponding serial module N' where output sets of actions are replaced by single output lines, which are in one-to-one correspondence with the output sets of N. Assume instead that N is an ND sequential machine. The module $N' = (Q', I', O', f', g', A')$ is derived from N in the usual way as described in Chapter 2. A' is defined similarly as for serial modules, where for all states $q \in Q$, all elements of $A'(q)$ are singletons, and for all input lines $a \in I'$, there is an entry $\{a\} \in A'(q)$ iff there is an action in state q containing the input line a. Note that if N is a ND sequential machine, then so is N'.

We now define some useful auxiliary functions to help in the description of the construction.

Definition 5.4. Given some ND sequential machine module
$N' = (Q', I', O', f', g', A')$, the following are quantified over all $a \in I'$ and $q \in Q'$:

1. Let $no(a, q)$ refer to the number of actions in state q that contain input a.
2. Let $max(a)$ be the maximum of all $no(a, q)$.
3. If $max(a) > 1$, let $relabel(a) = a_1, \ldots, a_n$, be any sequence of names where:
   - $n = max(a) - 1$,
   - for all $1 \leq i \leq n$: $a_i \notin I'$ and for all $b \in I'$, where $b \neq a$, $a_i \notin relabel(b)$.

   If $max(a) \leq 1$ then $relabel(a)$ is assumed to be undefined.
4. If $no(a, q) > 0$, let $setAct(a, q) = (a_1, B_1).q_1, \ldots, (a_m, B_m).q_m$ be any sequence given by the set of actions in state q which contain the input line a, where $m = no(a, q)$, $a_i = a$ for all $1 \leq i \leq m$, and no two actions in the sequence are equal. If $no(a, q) = 0$ then $setAct(a, q)$ is assumed to be undefined.
5. If $no(a, q) > 1$, and $no(a, q) < max(a)$, let $fill(a, q) = max(a) - no(a, q)$.

If $fill(a, q)$ is defined then it is always greater than 0. Informally, $fill(a, q)$ represents the number of actions we need to add to state q containing the input line a, in order for there to be $max(a)$ actions in q containing a.

Example 5.5. Consider the following module N_{10} defined as:

$$\begin{aligned} S_0 &= (a, \{x\}).S_1 \\ S_1 &= (a, \{x\}).S_0 + (a, \{y\}).S_0 + (a, \{z\}).S_0 \end{aligned}$$

Note that $no(a, S_0) = 1$ and $no(a, S_1) = 3$, and hence $max(a) = 3$. A possible definition of $relabel(a)$ is the sequence a_1, a_2. $setAct(a, S_0)$ is simply $(a, \{x\}).S_1$. A possible definition of $setAct(a, S_1)$ is $(a, \{x\}).S_0, (a, \{y\}).S_0, (a, \{z\}).S_0$. Finally, $fill(a, S_0) = 2$ and $fill(a, S_1)$ is undefined as $no(a, S_1) = max(a) = 3$.

We now show how to derive a standard sequential machine $N'' = (Q'', I'', O'', f'', g'', A'')$ from N', which will be utilised in the modification to Keller's construction. N'' is similar to N', but has duplicate instances of each input line a in each state's list of actions replaced by an alternative name from $relabel(a)$. Hence N'' contains no "internal" non-determinism in terms of actions.

The procedure for converting $N' = (Q', I', O', f', g', A')$ to $N'' = (Q'', I'', O'', f'', g'', A'')$ is as follows. Assume that initially $Q'' = Q'$, $I'' = I'$, $O'' = O'$, $f'' = f'$ and $g'' = g'$, and for all $a \in I'$ and $q \in Q'$, any valid definitions have been calculated for the functions in Definition 5.4.

1. For all $a \in I'$, we add all $a_k \in relabel(a)$ to I'' if $relabel(a)$ is defined.

2. For all states $q \in Q''$, all input lines $a \in I'$ where $no(a, q) > 1$, and all $0 < i \leq k$ where $setAct(a, q) = (a_0, B_0).q_0, \ldots, (a_k, B_k).q_k$, we relabel a_i in $(a_i, B_i).q_i \in q$, to the jth name in the sequence $relabel(a)$, where $j = i - 1$. This ensures that for each state, no input line appears in more than one action in that state.

3. For all states $q \in Q''$ and all input lines $a \in I'$, if $fill(a, q)$ is defined: then for all $1 \leq i \leq fill(a, q)$, add action $(a_k, B_k).q_k$ to q, where:

   - a_k is any name from $relabel(a)$ such that there is not already an action in q with the input a_k,
   - $B_k = B'$, $q_k = q'$ where $(a', B').q'$ is the only action of q with $a' = a$.

The resulting N'' always satisfies the conditions of a serial sequential machine module. We therefore assume that A'' is defined in the usual way for serial modules.

The third step in the above generation of N'' is to facilitate ease of construction. It makes sure that, if some input line a appears in some given state q, then for all possible names a_r given by $relabel(a)$, we make sure that there is also an action in q containing the input line a_r. This adds actions for all possible "alternative" names of a to q. Therefore, if an input line a is in some action in state q of N', and a has alternative names defined by $relabel(a)$, the resulting module N'' also contains an action in q for each alternative name of a.

Example 5.6. Assuming that N' is defined as N_{10} in Example 5.5, then a possible definition of N'' is as follows.

$$\begin{aligned} S_0 &= (a, \{x\}).S_1 + (a_1, \{x\}).S_1 + (a_2, \{x\}).S_1 \\ S_1 &= (a, \{x\}).S_0 + (a_1, \{y\}).S_0 + (a_2, \{z\}).S_0 \end{aligned}$$

This module is a sequential machine, whereas N_{10} was a ND sequential machine. Note that two of the actions in S_1 have had the input names in their actions relabelled from a to a_1 and a_2 respectively, so that now all actions in S_1 contain unique input names. Furthermore, two additional actions have been added to S_0, so that all alternative names of a now appear in actions in all states which originally contained at least one action containing a. The two additional actions in S_0 possess the same output set and resulting state as the original action in S_0. This ensures that signalling either a, a_1 or a_2 in S_0 always has the same effect as signalling a in S_0 of N_{10}.

We show how to modify Keller's construction method for any module (Figure 2.3 in Chapter 2) to accommodate N''. The differences between the construction in Figure 2.3 in Chapter 2, and the modified construction are as follows. In the modified construction:

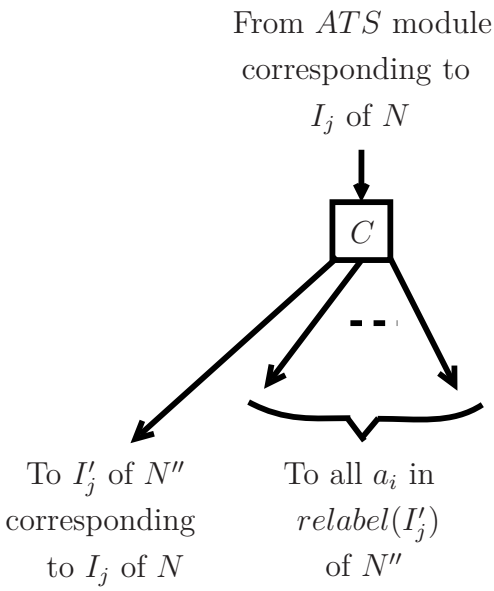


Figure 5.2: Modification to Keller's construction method to accommodate N'' and realise ND sequential machines.

1. We replace N' with N''.

2. For all ATS modules, if its R input line corresponds to I_j of N where $max(I'_j) > 1$ (where I'_j is the input line of N'' which also corresponds with I_j of N),

   - instead of connecting the T_0 output line to I'_j of N'', we connect the T_0 output line of this ATS module to the input line of a n-way *Choice* tree, where $n = max(I'_j)$,
   - we connect any output line of this *Choice* tree to the input line I'_j of N'', where I'_j corresponds with I_j of N,
   - for all remaining output lines b of this *Choice* tree, we connect b to any unconnected a_i of N'', where $a_i \in relabel(I'_j)$.

In Figure 5.2, we depict such a modification.

The result of this modification is such that signalling the input line of the network corresponding to I_j of the original module N, when the network is in the state corresponding to q of N, will non-deterministically result in the processing of some action among n actions of q of N'', where $n = max(I'_j)$ and I'_j of N'' corresponds to I_j of N. One of these actions of N'' is equal to an action in the module N'. Some of these actions may be modifications of actions in N', with the input line of the action I'_j replaced by a alternative name given by $relabel(I'_j)$. Finally, some of these actions may have been added as a result of the duplication of existing actions of N', with the input line I'_j of the action also replaced with an alternative name given by

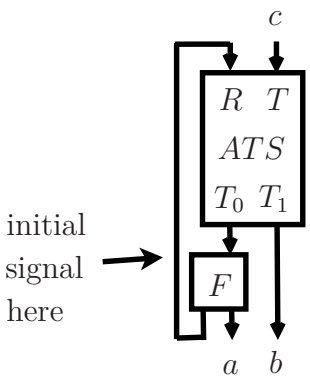


Figure 5.3: *Choice* module realised using *ATS* and *Fork*. *ATS* is in S_1.

$relabel(I'_j)$. This preserves the non-determinism of N resulting from a single input signal, despite N'' being a sequential machine.

This gives us the following universal set for the class of ND sequential machines.

Theorem 5.7. The set of modules {*ATS, Fork, Merge, Select, Choice*} is universal (in the sense of Definition 2.15 in Chapter 2) for the class of ND sequential machines.

Proof. Any ND sequential machine can be constructed by following Keller's general construction method for any module (Figure 2.3 in Chapter 2), and then making the modifications outlined above. This involves replacing the ND sequential machine N' with a sequential machine N'' and then incorporating *Choice* modules into the construction. N'' is a serial sequential machine, and so can be realised with {*Merge, Select*} as in the original construction method. The only new module required is *Choice*. □

In Figure 5.3 we show how to realise *Choice* using only *ATS* and *Fork*. Trivially, any environment of *Choice* is one which simply sends an input signal on c, then waits for an output signal on a or b before repeating this behaviour. Such an environment cannot distinguish between the realisation in Figure 5.3 and the original *Choice* module, as sending an input signal on c may result in an output signal on either a or b, and this can happen repeatedly. Hence the network in Figure 5.3 satisfies the notion of realisation (Definition 2.14 in Chapter 2) used in the sequential machine model. This gives us the following smaller universal set.

Proposition 5.8. The set of modules {*Merge, Select, Fork, ATS*} is universal (in the sense of Definition 2.15 in Chapter 2) for the class of (ND) sequential machines.

Proof. Theorem 2.25 in Chapter 2 proves universality for the class of sequential machines. Theorem 5.7 combined with the realisation of *Choice* using *ATS* and *Fork* proves universality for the class of ND sequential machines. □

This is the same universal set as the one given in Theorem 2.25 in Chapter 2 for the class of sequential machines. This demonstrates an interesting property. Informally, while individual ND sequential machine module definitions are more expressive than sequential machine module definitions, networks of sequential machines are powerful enough to realise any ND sequential machine. Intuitively, it may be more desirable from an engineering perspective to utilise a universal set which contains only sequential machines, as depending on the technology in question, ND sequential machines may be more physically complex due to the presence of explicit internal non-determinism.

Corollary 5.9. Any ND sequential machine can be realised (in the sense of Definition 2.14 in Chapter 2) using a network of sequential machines.

Proof. The set {*Merge, Select, Fork, ATS*} contains only sequential machines and is universal (in the sense of Definition 2.15 in Chapter 2) for the class of ND sequential machines. □

We finish by proving a universal set for both sequential machines and ND sequential machines, which does not contain the irreversible serial module *Select*, and instead replaces it with the reversible serial modules *RT* and *IRT* (Figure 2.4 in Chapter 2).

Theorem 5.10. {*ATS, Fork, Merge, RT, IRT*} is universal (in the sense of Definition 2.15 in Chapter 2) for the class of (ND) sequential machines.

Proof. Proposition 5.8 proves that {*Merge, Select, Fork, ATS*} is universal for (ND) sequential machines. *Select* can be realised using {*RT, IRT, Merge*} (Corollary 2.41 in Chapter 2). □

5.2 Implementing Set Notation modules using (ND) sequential machines

We now investigate the notion of a (ND) sequential machine "implementing" the functionality of a Set Notation module. The motivation for this is regarding physical implementation. Depending on the technology in question, concurrent signals may need to be physically assimilated in a sequential manner. It is for this reason that the ND sequential machine model cannot be discounted due to the introduction of the Set Notation model. We will also use this new notion of implementation to prove a universal set regarding all Set Notation modules in Chapter 6.

Recall that the definition of *Join* in the Set Notation model can be considered "externally equivalent" to the definition of the identically named *Join* in the sequential machine model, provided that the safety and non-clashing properties hold (Example 3.14 in Chapter 3).

We attempt to define a limited one-way version of this correspondence more formally. We first define a variation of the Set Notation model which we call the *Seq/Set model*. The Seq/Set Notation model is used exclusively in this chapter to prove this correspondence between Set Notation modules and (ND) sequential machines.

Definition 5.11. We define the *Seq/Set model* the same as the Set Notation model, except modules may contain empty output sets in their transition maps.

Informally, the Seq/Set model is the same as the Set Notation model, but allows us to utilise (ND) sequential machines directly within the model. This is in order to establish a formal correspondence between Set Notation modules and ND sequential machines. All definitions and notions are otherwise equivalent to the Set Notation model.

When referring to (ND) sequential machines operating under the Seq/Set model, we assume that set brackets are present around the input lines in the CCS-like definitions of modules. Similarly, we assume that the f and g partial functions are combined into the single transition map $T : Q \times (P[I]) \setminus \emptyset \to Q \times P[O]$ as required by the Seq/Set model (note however, input sets are always singletons). The A function is still required for these modules. Hence a (ND) sequential machine is given in the form (Q, I, O, T, A) in the Seq/Set model. Set Notation modules are still defined in the form (Q, I, O, T), and environments are still defined in the form (Q, I, O, T, N, sc).

Definition 5.12. Given a Set Notation module $N = (Q, I, O, T)$ and a (ND) sequential machine module $N' = (Q', I', O', T', A')$, such that $|Q'| \geq |Q|$, we define a *state mapping* function sm between N and N' to be a bijection where, for some $Q_s \subseteq Q'$ with $|Q_s| = |Q|$, $sm : Q \to Q_s$.

We note that the definition of a *normally-connected* network in the Set Notation model (Definition 4.2 in Chapter 2) still applies to networks containing Set Notation modules and their corresponding environments when operating under the Seq/Set Notation model. We introduce a similar notion for (ND) sequential machines operating under the Seq/Set Notation model.

Definition 5.13. We say that a network composed of a single (ND) sequential machine module $N' = (Q, I, O, T, A)$ in some state $q \in Q$ and a corresponding environment $E = (Q', I', O', T', N, sc)$ in some state $q' \in Q'$ of some Set Notation

module N, such that $I' = O$ and $O' = I$, is *identically-connected* if all output lines of the module connect to the identically named input lines of the environment, all output lines of the environment connect to the identically named input lines of the module, and there are no signals present on any wires.

We now introduce the notion of *sequential implementation*.

Definition 5.14. Consider some Set Notation module $N = (Q, I, O, T)$, and a (ND) sequential machine $N' = (Q', I', O', T', A')$. We say that N' *sequentially implements* N if there exists a state-mapping function sm between N and N' such that, given any corresponding maximal environment $E = (Q'', I'', O'', T'', N, sc)$ of N, the following conditions hold for all $q \in Q$:

1. Identically-connecting E in state $sc(q) \in Q''$ to N' in state $sm(q) \in Q'$ results in a network S which is non-clashing and:
   - E is safe in S,
   - for every state variation S' of S which can be reached as a result of the execution behaviour (Definition 3.5 in Chapter 3) starting from S, there exists some $A_1 \in A'(q')$ where q' is the state of N' in S', and $A_2 \subseteq A_1$ where A_2 is the set of input lines of N' whose connected wires contain one or more signals in S'.
2. For all possible sequences of actions $(A_1, B_1).q_1, (A_2, B_2).q_2 \ldots$ of E which may be processed when normally-connecting N in state $q \in Q$ to E in state $sc(q)$ (where $(A_1, B_1).q_1 \in sc(q)$), it is possible for the same sequence of actions to be processed when identically-connecting E in state $sc(q)$ to N' in state $sm(q)$, and vice versa.

Informally, this means that a (ND) sequential machine N' sequentially implements a Set Notation module N if, assuming that the environment is a maximal environment of N, the environment cannot distinguish between N and N'. It is similar to the definitions of indistinguishability and implementation (Definitions 4.27 and 4.29 in Chapter 4), but is modified to account for the fact that the definition of safety in the Set Notation model does not apply to (ND) sequential machines. Instead, the second sub-condition of the first condition in Definition 5.14 requires safety of the (ND) sequential machine according to the requirements outlined in the sequential machine model (Definition 2.5 in Chapter 2).

Example 5.15. The sequential machine *Join* (Figure 2.1 in Chapter 2) sequentially implements the Set Notation module *Join* (Example 3.12 in Chapter 3), where $sm(J) = S_0$. This can be seen by identically-connecting a maximal environment E

of the Set Notation *Join* to the sequential machine *Join*, which results in E processing the same sequences of actions than when normally-connecting E to the Set Notation *Join*. Informally, this is the single infinite sequence where the environment repeatedly sends input signals on $\{a, b\}$ and then receives an output signal from $\{c\}$.

It would be ideal to define a converse notion where a Set Notation module can be considered to "implement" a (ND) sequential machine. However, this requires formal notions of environments and maximal environments for (ND) sequential machines. There are various reasons as to why this is not easy or trivial. Determining, in general, the sequences of input and output signals that an environment may send and receive to and from a (ND) sequential machine module, such that safety and non-clashing always hold appears difficult. Recall the challenges faced with generating environments for Set Notation modules which are auto-firing, auto-clashing or not 1-step consistent (see Chapter 4). These properties, and the abstract behaviour of modules implied by such properties, clearly have counterparts in the (ND) sequential machine model. Defining these notions in the (ND) sequential machine model appears to be more complex than in Set Notation due to the nature of the (ND) sequential machine model's approach to concurrency. This further complicates the issue of identification and generation of valid environments. Environments and maximal environments for (ND) sequential machines are therefore beyond the scope of this thesis.

However, in very simple cases such as the sequential machine *Join* and *ATS*, it is easy to enumerate all possible environment behaviours, and the "maximal" environments for such sequential machine modules are trivial and it is possible to define them informally. See Chapter 3, where we have argued informally that the Set Notation modules *Join* and *mATS* "implement" the behaviour provided by the sequential machine modules *Join* and *ATS* respectively.

Example 5.16. Recall the maximal environment *EnvfATS* of *mATS* (Example 4.33 in Chapter 4). By enumerating all possible sequences of environment behaviours, it can be determined that the sequential machine *ATS* module sequentially implements *mATS* under the above requirements, where $sm(S_1) = S_1$ and $sm(S_0) = S_0$. Moreover, *ATS* sequentially implements *sATS*, where $sm(S_1) = S_1$, as the definition of ES_1 of the maximal environment *EnvsATS* of *sATS* (Example 4.15 in Chapter 4) is the same as ES_1 of *EnvfATS*. However recall that *ATS* initialised in S_0 cannot have its behaviour matched by *sATS* (Section 3.3 in Chapter 3). Hence, while *ATS* sequentially implements *sATS*, it appears that *sATS* cannot be seen to be "equivalent" to *ATS* in general.

5.3 Converting between models

We will give algorithms which will allow us to convert modules from the Set Notation model to the ND sequential machine model and (in a limited fashion) vice versa, such that the (ND) sequential machine module sequentially implements the Set Notation module.

5.3.1 Realisability of Set Notation modules as sequential machines

We first identify which Set Notation modules can be sequentially implemented by sequential machines, and which can only be sequentially implemented by ND sequential machines.

Recall the Set Notation definition of *Join* is $J = (\{a, b\}, \{c\}).J$ (Example 3.12 in Chapter 3). Informally, this means that regardless of the order that input signals arrive on a and b, an output signal is produced on c. A (ND) sequential machine which sequentially implements this module would therefore simply produce an output signal on c regardless of the order that input signals on a and b are processed by the module. To do this, the definition must consider all possible permutations of the set $\{a, b\}$ and produce an output signal on c after processing input signals in either possible sequence. This is exactly the behaviour of the sequential machine *Join* (Figure 2.1 in Chapter 2).

We can conclude that any non-arb module N has some corresponding sequential machine N', such that N' sequentially implements N. The corresponding sequential machine simply needs to process each Set Notation module's input set using sequences of actions corresponding to all possible permutations of the set, and then produce the set of output signals after processing the last input signal in each sequence.

Proposition 5.17. Any non-arb module N can be sequentially implemented by some sequential machine N'.

Proof. For all states q of N, and all actions $(A, B).q' \in q$, the sequential machine N' simply needs to process each signal on each input line in A one at a time in all possible sequences corresponding to permutations of A, and produce the set of output signals on the lines B after processing the last input signal in each sequence. □

We now examine the ability to sequentially implement eq-arb modules using sequential machines via two examples.

Example 5.18. Consider the eq-arb module N_{11} defined as:

$$S_0 \quad = \quad (\{a, b\}, \{x\}).S_0 + (\{a, b\}, \{y\}).S_0$$

A sequential machine which sequentially implements this module would have to account for processing signals in sequences corresponding to both possible permutations of $\{a, b\}$. However it would also have to account for the two different possible output sets which may be produced as a result of processing $\{a, b\}$. The fact that the number of actions containing $\{a, b\}$ is less than or equal to the number of possible permutations of $\{a, b\}$ is critical. This means that each possible output set (and resulting state change) can be produced as a result of at least one processing sequence. Hence the deterministic nature of sequential machines is sufficient to provide the non-determinism exhibited by N_{11}. One possible sequential machine which sequentially implements this module is N'_{11}, defined as:

$$\begin{array}{lcll} S_0 & = & (a, \{\}).S_a + (b, \{\}).S_b & \{\{a, b\}\} \\ S_a & = & (b, \{x\}).S_0 & \{\{b\}\} \\ S_b & = & (a, \{y\}).S_0 & \{\{a\}\} \end{array}$$

The set at the end of each line is the corresponding A function entry for that state. Clearly, if there were instead three or more actions containing the input set $\{a, b\}$ in the same state of N_{11}, it would not be possible to sequentially implement N_{11} using a sequential machine, as there are only 2 possible permutations of this set, and therefore only 2 possibilities for "deciding" the output set and resulting state in the sequential machine.

Note that if the input set of an action is cardinality 3 or more, there are 6 permutations of the set, 6 possible sequences in which the set of signals must be processed, and therefore 6 actions containing the input set would be permitted in the same state of the Set Notation module. The limitation of an eq-arb module to be sequentially implemented by a sequential machine appears to be linked to the number of permutations of each input set, and the number of occurrences of this input set in actions of the same state.

Example 5.19. Consider the input set $\{b, c\}$ which is not a subset of $\{a, b\}$, or vice versa. Now consider the module N_{12} defined as:

$$S_0 = (\{a, b\}, \{x\}).S_0 + (\{a, b\}, \{y\}).S_0 + (\{b, c\}, \{z\}).S_0$$

A possible sequential machine definition which sequentially implements N_{12} is N'_{12},

defined as:

$$\begin{array}{lcll} S_0 & = & (a,\{\}).S_a + (b,\{\}).S_b + (c,\{\}).S_c & \{\{a,b\},\{b,c\}\} \\ S_a & = & (b,\{x\}).S_0 & \{\{b\}\} \\ S_b & = & (a,\{y\}).S_0 + (c,\{z\}).S_0 & \{\{a\},\{c\}\} \\ S_c & = & (b,\{z\}).S_0 & \{\{b\}\} \end{array}$$

Note that the input set $\{b,c\}$ must be processed by N'_{12} in S_0 in both sequences b,c and c,b in order to guarantee that signalling $\{b,c\}$ always results in the output set $\{z\}$ and resulting state S_0, as in N_{12}.

Now consider that, as in N'_{11}, the module N'_{12} must also process the input set $\{a,b\}$ in both sequences a,b and b,a. Each sequence provides the possibility of producing a different output set and state change ($\{x\}$ with S_0 or $\{y\}$ with S_0), which are used to "implement" the non-deterministic behaviour of N_{12} when signalling $\{a,b\}$. Note however that neither of these sequences are prefixes of any sequence corresponding to a permutation of $\{b,c\}$ (or vice versa). As a result, it is possible for N'_{12} to process the input set $\{b,c\}$ in both sequences, resulting in the output set $\{z\}$ in both cases, without modifying the actions from N'_{11} which are responsible for processing $\{a,b\}$. Modifying N'_{11} to "implement" actions from N_{12} containing $\{b,c\}$, to result in N'_{12}, is achieved by only adding actions and states.

We can conclude from this example that the presence of different input sets in actions of the same state of the Set Notation module, which are not strict subsets or strict supersets of each other, does not affect the previous limitation regarding the number of permutations of each input set, and the number of occurrences of this input set in actions of the same state.

Proposition 5.20. An eq-arb module $N = (Q, I, O, T)$ can be sequentially implemented by some sequential machine N' if for all $(q_1, A_1, q'_1, B_1) \in T$, the number of transitions $(q_2, A_2, q'_2, B_2) \in T$, where $q_2 = q_1$ and $A_2 = A_1$ is less than or equal to $n!$ where $n = |A_1|$.

Proof. Similarly to sequential machines which sequentially implement non-arb modules, each input set A_1 of N needs to be processed by some sequential machine N', one signal at a time, in all possible sequences corresponding to permutations of A_1. Each sequence provides an opportunity for non-determinism. The limiting factor to whether some sequential machine N' can sequentially implement an eq-arb module N is therefore whether the number of actions in the same state of N which contain A_1 is larger than the number of permutations of A_1, for all possible A_1. Actions in the same state of N which contain different input sets do not affect this limitation if they are not a strict subset or strict superset of A_1. By definition, an eq-arb module

does not contain actions in the same state such that one input set is a strict subset or strict superset of another. □

We now generalise this to all eq-arb modules and (ND) sequential machines. If the number of actions in a state containing the same input set is larger than the number of permutations of the set, we can utilise the non-deterministic nature of ND sequential machines to provide additional alternatives for the output set and state change at the end of one of the sequences of actions which processes the input set.

Example 5.21. Consider the module N_{13} defined as:

$$S_0 = (\{a,b\},\{x\}).S_0 + (\{a,b\},\{y\}).S_0 + (\{a,b\},\{z\}).S_0 + (\{b,c\},\{z\}).S_0$$

Note that the input set $\{a,b\}$ appears in three actions in S_0, and there are only two permutations of $\{a,b\}$. Hence this cannot be sequentially implemented by a sequential machine. Instead, a possible ND sequential machine definition which sequentially implements this module is N'_{13}, defined as:

$$\begin{array}{lcll} S_0 & = & (a,\{\}).S_a + (b,\{\}).S_b + (c,\{\}).S_c & \{\{a,b\},\{b,c\}\} \\ S_a & = & (b,\{x\}).S_0 + (b,\{z\}).S_0 & \{\{b\}\} \\ S_b & = & (a,\{y\}).S_0 + (c,\{z\}).S_0 & \{\{a\},\{c\}\} \\ S_c & = & (b,\{z\}).S_0 & \{\{b\}\} \end{array}$$

Note that this is similar to N'_{12} from Example 5.19, but an additional action $(b,\{z\}).S_0$ has been added to S_a, resulting in a ND sequential machine. This preserves the possibility that signalling $\{a,b\}$ in S_0 may produce an output on $\{z\}$ and a change to S_0. We also note that an alternative modification to N'_{12} from Example 5.19 which has the same effect involves adding the action $(a,\{z\}).S_0$ to S_b.

Proposition 5.22. Any eq-arb module N can be sequentially implemented by some (ND) sequential machine N'.

Proof. Proposition 5.20 proves the set of eq-arb modules which can be sequentially implemented by sequential machines. If an eq-arb module N cannot be sequentially implemented by a sequential machine, we utilise non-determinism in the definition of N'. This is achieved by adding new actions to N', containing input lines which are already present in actions in the same state, where the output sets and resulting states are that of actions in N which have not yet been "implemented" in N'. This causes N' to non-deterministically "decide" between multiple actions of N during the processing of the final input signal in a sequence of actions which processes an input set of N. □

We wish to generalise this relationship to all Set Notation modules and (ND) sequential machines. We investigate via an abstract example.

Example 5.23. Consider some Set Notation module $N = (Q, I, O, T)$ which contains the input set $\{a, b, c, d, e\}$ in some action in some state $q \in Q$. The set $\{a, b, c, d, e\}$ permits $5! = 120$ processing sequences in the (ND) sequential machine N' which sequentially implements this module, before requiring us to utilise non-determinism in the definition of N'. Hence this input set could appear in 120 actions in q, while still allowing a sequential machine to sequentially implement this module.

However, now consider the possibility of the input set $\{a, b\}$ in some action in q. Note that $\{a, b\}$ is a proper subset of $\{a, b, c, d, e\}$. Clearly, to "implement" the functionality of this action, the (ND) sequential machine must process the input set $\{a, b\}$ in the sequences a, b and b, a. The final action in both of these sequences must produce the output set and resulting state change according to some action in q of N which contains the input set $\{a, b\}$. Otherwise signalling $\{a, b\}$ may lead to no output signals from the sequential machine, which is not allowed by the behaviour of N. Hence this prevents all sequences corresponding to permutations of $\{a, b, c, d, e\}$ which begin with either a, b or b, a from being "available" to the (ND) sequential machine to process the input set $\{a, b, c, d, e\}$ starting in q. In this example, this rules out exactly 12 sequences from the 120 permutations of $\{a, b, c, d, e\}$.

Now consider the possibility that the input set $\{a\}$ is present in some action in q of N, in addition to actions containing the input sets $\{a, b\}$ and $\{a, b, c, d, e\}$ in q of N. By the same reasoning as above, the presence of the input set $\{a\}$ in some action rules out some sequences for the sequential machine to process the input set $\{a, b\}$ in q of N (corresponding to ruling out exactly the one sequence a, b, but leaving the sequence b, a "available"). Additional sequences corresponding to permutations of $\{a, b, c, d, e\}$ which begin with a are also ruled out for processing the input set $\{a, b, c, d, e\}$. Note however that all sequences beginning with a, b have already been ruled out as "available" for processing the input set $\{a, b, c, d, e\}$, as a result of the presence of an action in q of N which contains the input set $\{a, b\}$.

In general, given some Set Notation module N, if for all actions $(A, B).q'$ in all states q of N, after ruling out sequences corresponding to permutations of A which are prefixed by a sequence needed to process some other input set $A' \subset A$ (where there is an action containing A' in q), the number of remaining sequences corresponding to permutations of A is larger than or equal to the number of actions in q containing A, then N can be sequentially implemented by a sequential machine. As with eq-arb modules, the presence of other input sets in actions in q which are not strict subsets or strict supersets of A does not affect this limitation.

To express this notion formally, we must first define some useful functions and predicates.

Definition 5.24. For all pairs of sequences p_1 and p_2, let the predicate $\mathit{prefix}(p_1, p_2)$ hold iff p_1 is a prefix of p_2.

Definition 5.25. Given some Set Notation module $N = (Q, I, O, T)$, the following are quantified over all $A \subseteq I$ and all $q \in Q$:

1. Let $\mathit{sequences}(A)$ be the set of sequences given by all possible permutations of the set A.

2. Let $\mathit{total}(A, q)$ be the total number of actions $(A, B').q''$ in q for some B', q''.

3. Let $\mathit{remaining}(A, q)$ be the largest proper subset of $\mathit{sequences}(A)$, such that for all $p \in \mathit{remaining}(A, q)$ the following condition holds:

   - for all $A' \subset A$, if $(A', B).q'$ is an action of q for some B, q', then for all $p' \in \mathit{sequences}(A')$ it is not the case that $\mathit{prefix}(p', p)$.

Informally, $\mathit{remaining}(A, q)$ is the set of sequences "available" for the sequential machine to process the input set A in state q, after ruling out all sequences which contain prefixes that are required to process proper subsets of A in q.

Proposition 5.26. A Set Notation module N can be sequentially implemented by a sequential machine iff for all $q \in Q$ and all actions $(A, B).q' \in q$, $\mathit{total}(A, q) \leq |\mathit{remaining}(A, q)|$.

Proof. Similarly to sequential machines which sequentially implement non-arb or eq-arb modules, each input set A needs to be processed by some sequential machine N', one signal at a time, in all possible sequences corresponding to permutations of A. Each sequence provides an opportunity for non-determinism. However certain sequences are ruled out as "available" for processing a given input set A in a state q of N whenever there exists a proper subset of A in some action in q. The limiting factor to whether a sequential machine can sequentially implement a module N is therefore whether the number of actions in the same state of N which contain the input set A is larger than the number of "available" sequences which can process the input set A, for all possible A. □

Using a similar reasoning to Proposition 5.22, it can be argued that any Set Notation module can be sequentially implemented using some (ND) sequential machine.

Proposition 5.27. Any Set Notation module N can be sequentially implemented by some (ND) sequential machine N'.

Proof. Proposition 5.26 proves the set of Set Notation modules which can be sequentially implemented by sequential machines. Similarly to Proposition 5.22, if a module N cannot be sequentially implemented by a sequential machine, we utilise non-determinism in the definition of N'. This is achieved by adding new actions to N', containing input lines which are already present in actions in the same state, where the output sets and resulting states are that of actions in N which have not yet been "implemented" in N'. This causes N' to non-deterministically "decide" between multiple actions of N during the processing of the final input signal in a sequence of actions which processes an input set of N. □

5.3.2 (ND) sequential machines to Set Notation modules

We established earlier in this chapter that it is difficult to prove in general that a Set Notation module N has the same "external behaviour" as some (ND) sequential machine N', due to the difficulty of formalising the notion of an environment for a (ND) sequential machine. Regardless, we give a limited algorithm which, given some (ND) sequential machine N' which satisfies certain conditions, we claim produces a Set Notation module N such that N' sequentially implements N.

We introduce some useful notation to help express these conditions. Given some (ND) sequential machine module $N' = (Q, I, O, f, g, A)$, let $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} q_n$ iff for all $1 \leq i < n$, $f(q_i, a_i) = q_{i+1}$.

Given some (ND) sequential machine module $N' = (Q, I, O, f, g, A)$, the following conditions are quantified over all $L \in A(q_1)$ for all $q_1 \in Q$, all sequences $a_1, \ldots, a_n$ given by all possible permutations of L, and all $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_{n+1}$.

- **E1**: $g(q_i, a_i) \cap g(q_j, a_j) = \emptyset$ for all $1 \leq i, j \leq n$.

Informally, this condition states that for all states q and all concurrent sets $L \in A(q)$, signalling the set L in state q cannot result in multiple signals (i.e. clashing) on the wires connected to the module's output lines.

- **E2**: $g(q_i, a_i) \neq \emptyset$ for some $1 \leq i \leq n$.

Informally, this condition states that for all states q and all concurrent sets $L \in A(q)$, signalling the set L in state q must eventually produce a non-empty set of output signals.

For simplicity, the algorithm assumes that the (ND) sequential machine is given in the form (Q, I, O, T, A), that is to say that f and g are combined into the single transition map $T : Q \times I \to Q \times P[O]$, similarly to when encoding (ND) sequential machines in the Seq/Set model. We give the algorithm in Figure 5.4.

Require: (ND) sequential machine $N' = (Q', I', O', T', A')$ which satisfies conditions E1 and E2.

create Set Notation module $N = (Q, I, O, T)$ where $Q = Q', I = I', O = O'$ and $T = \{(q, \{a\}, q', B) : (q, a, q', B) \in T'\}$
let $n = |B|$ for the largest $B \in A'(q')$ for all $q' \in Q'$
for every $2 \leq i \leq n$ **do**
    for every $q \in Q \cap Q'$ **do**
        for every set $B \in A'(q)$ such that $|B| \geq i$ **do**
            let $subs = \{C \subseteq B : |C| \leq i - 1\}$
            for all pairs D, E in $subs$ such that $D \cap E = \emptyset$, and $|D| + |E| = i$ **do**
                for all actions $(D, F).q'$ of q **do**
                    for all actions $(E, G).q''$ of q' **do**
                        create action $(D \cup E, F \cup G).q''$ in q of N
                    end for
                end for
                for all actions $(E, G).q'$ of q **do**
                    for all actions $(D, F).q''$ of q' **do**
                        create action $(D \cup E, F \cup G).q''$ in q of N
                    end for
                end for
            end for
        end for
    end for
end for
delete all actions of N containing empty output sets

Ensure: Set Notation module $N = (Q, I, O, T)$

Figure 5.4: Algorithm for converting a (ND) sequential machine N' to a Set Notation module N.

Optionally, it may also be possible in practice to delete certain states in N' which are unreachable from some assumed starting state, depending on the intended operation of the module. This is due to the fact that these unreachable states may be equivalent to a situation where the module is in a different state with some signals pending, but has not yet absorbed them.

Example 5.28. Inputting the sequential machine *Join* (Figure 2.1 in Chapter 2) to the algorithm would yield the following Set Notation module.

$$\begin{aligned} S_0 &= (\{a,b\},\{c\}).S_0 \\ S_a &= (\{b\},\{c\}).S_0 \\ S_b &= (\{a\}),\{c\}.S_0 \end{aligned}$$

Note that S_a and S_b are not reachable from S_0. These states correspond to S_a and S_b of the sequential machine *Join* respectively and also correspond to a scenario where the sequential machine *Join* has absorbed only one of the two required input signals. However, due to the nature of Set Notation, this situation can also be modelled by placing a signal on one of the wires connected to either a or b when the module is in state S_0. Therefore we can safely remove states S_a and S_b from the above definition, resulting in the single state module $S_0 = (\{a,b\},\{c\}).S_0$, which is equivalent to the Set Notation *Join* (Example 3.12 in Chapter 3, where the state S_0 is instead labelled J).

However it is not clear that it is always possible in general to remove states, and so we do not assume it as part of the algorithm.

Informally, we note that the algorithm is not "minimal" in the sense that inputting N' may not result in the simplest such N, where N' sequentially implements N. An example of this is the *ATS* module. Inputting *ATS* to the algorithm yields the *fATS* Set Notation module (Example 4.32 in Chapter 4), but we have already shown in Example 5.16 that *ATS* also sequentially implements *mATS*. Furthermore, *mATS* and *fATS* implement each other (Section 4.3 in Chapter 4).

Remark 5.29. We note that the conditions $E1$ and $E2$, which limit the modules that may be input to the algorithm in Figure 5.4, guarantee that an auto-clashing Set Notation module is never produced.

Claim 5.30. Given a (ND) sequential machine module N', the algorithm in Figure 5.4 yields a Set Notation module N such that N' sequentially implements N.

5.3.3 Set Notation modules to (ND) sequential machines

We finish this chapter by giving an algorithm for converting any Set Notation module N to a sequential machine module N', such that N' sequentially implements N.

Similarly to the previous section, for simplicity, the algorithm assumes that f and g are combined into the single transition map $T : Q \times I \to Q \times P[O]$. Hence the resulting (ND) sequential machine is given by (Q, I, O, T, A).

Before introducing the algorithm, we give two important auxiliary functions.

Definition 5.31. Given some Set Notation module $N = (Q, I, O, T)$, the following are quantified over all $A \subseteq I$ and all $q \in Q$:

1. Let $S(A, q) = \{(q, A, q', out) : (q, A, q', out) \in T\}$.
2. Let $aggregate(q) = \{S(A, q) : S(A, q) \neq \emptyset\}$.

Informally, $S(A, q)$ returns the set of transitions which share the input set A and the source state q. $aggregate(q)$ returns the set of all possible $S(A, q)$ for a given state q.

Assume in the following that a state name is represented using a string, and two state variables are equal if their strings are equal. Assume also that the ability to concatenate strings is defined in the usual way using the $\cdot$ operator. This allows us to easily encode data in the state names of the module during the algorithm's execution. We require that no state variable initially contains a "—" character in its string representation. We give the algorithm in Figure 5.5. The algorithm converts a module to a sequential machine module if possible, and only converts to a ND sequential machine module if required.

The final step in the algorithm in Figure 5.5 is performed by following the same steps as the *state-merge* algorithm for pseudo-environments (Figure 4.3 in Chapter 4), but modified to account for the fact that transitions of (ND) sequential machines contain single input lines rather than input sets. It is not included as the resulting algorithm is identical with this exception.

Example 5.32. Inputting the Set Notation module *Join* (Example 3.12 in Chapter 3) to the algorithm in Figure 5.5 yields (if we ignore the difference in state names) the sequential machine module *Join* (Figure 2.1 in Chapter 2).

Proposition 5.33. Given any Set Notation module N, the algorithm in Figure 5.5 yields a (ND) sequential machine N' such that N' sequentially implements N. If N can be sequentially implemented by a sequential machine (according to Proposition 5.26), then N' is a sequential machine, otherwise it is a ND sequential machine.

Proof. The algorithm generates the behaviour of N' for each state q of N, by selecting each action $(A, B).q'$ one at a time from the set of actions in q of N which contain A. In general, each $(A, B).q'$ is "implemented" in N' using a single sequence of actions corresponding to a single permutation of A. In the final action in a sequence, the output set B is produced and the state of N' changes to the state which

Require: Set Notation module $N = (Q, I, O, T)$ where no $q \in Q$ contains the character "—"

create (ND) sequential machine module $N' = (Q', I', O', T', A')$ where $Q' = Q, I' = I, O' = O$, T is empty, and for all $q' \in Q'$, $A(q')$ returns the empty set
for every $q_1 \in Q \cap Q'$ **do**
    create empty set U (of type sequence)
    for every $S = \{(q_1, B, q', C) : (B, C).q' \in q_1$ for some $C, q'\} \in aggregate(q_1)$ in increasing size of B, such that $|B| > 1$ **do**
        for some $(q_1, B, q', C) \in S$, add B to $A(q_1)$
        let $S' = S$ and let $P = sequences(B)$ for some $(q_1, B, q_2, C) \in S$
        for every $p_1 \in P$ **do**
            if there exists some $p_2 \in U$, where $prefix(p_2, p_1)$, remove p_1 from P
        end for
        add all elements of P to set U
        for every transition $t = (q_1, B, q', C) \in S'$ **do**
            remove t from S'
            if $|S'| = 0$ **then** let $P' = P$
            else let $P' = P''$ for some $P'' \subseteq P$ where $|P''| = 1$ **end if**
            remove all elements of P' from P if $|P| > 1$
            for every $p = (i_1, i_2, \ldots, i_n) \in P'$ **do**
                create transition $(q_1, i_1, q_2, \emptyset) \in T'$ if it does not exist where $q_2 = (q_1 \cdot$"—"$\cdot i_1)$, and if $q_2 \notin Q'$ it is added to Q' and we set $A(q_2) = \emptyset$
                for every $1 < j < n$ **do**
                    create transition $(q_j, i_j, q_{j+1}, \emptyset) \in T'$ if it does not already exist where $q_j = (q_1 \cdot$"—"$\cdot i_1 \cdot \ldots \cdot i_{j-1}) \in Q'$, and $q_{j+1} = (q_j \cdot i_j)$, and if $q_{j+1} \notin Q'$ it is added to Q' and we set $A(q_{j+1}) = \emptyset$
                    add to $A(q_j)$ the set $D = \{i_j, i_{j+1}, \ldots, i_n\}$ if there does not exist some $E \in A(q_j)$ where $D \subseteq E$
                end for
                create transition $(q_n, i_n, q', C) \in T'$ if it does not already exist where $q_n = (q_1 \cdot$"—"$\cdot i_1 \cdot \ldots \cdot i_{n-1}) \in Q'$ (note $q' \in Q'$)
                add to $A(q_n)$ the singleton set $\{i_n\}$ if there does not exist some $C \in A(q_n)$ where $i_n \in C$
            end for
        end for
    end for
end for
remove duplicate states of N' and associated transitions, and redirect transitions' co-domains appropriately if any states are removed.

Ensure: (ND) sequential machine module $N' = (Q, I, O, T, A)$.

Figure 5.5: Algorithm for converting any Set Notation module N to a (ND) sequential machine module N'.

corresponds to q' of N. If $(A, B).q'$ is the last action in q of N to be "implemented" which contains A, it will be "implemented" in N' using all remaining available sequences corresponding to permutations of A. If there are no available sequences remaining to process A, non-determinism is utilised in N' and the penultimate state in an "already-used" sequence has a new action added containing the final input of that sequence (resulting in N' becoming a ND sequential machine). This new action's output set is B and its resulting state (of N') corresponds to q' of N. Sets of actions from q of N are "implemented" in N' in order of increasing size of input set. This allows the algorithm to rule out sequences which are prefixed by sequences corresponding to permutations of smaller sets in actions in q of N. This guarantees that a ND sequential machine is only produced if N does not fit the conditions given in Proposition 5.26. □

5.4 Conclusion

In this chapter we compared the sequential machine model for DI networks with the new Set Notation model. We introduced an extension to the sequential machine model called the *ND sequential machine* model. We established limited correspondences between three models; the sequential machine model, the ND sequential machine model, and the new Set Notation model. We proved universality results for the ND sequential machine model. We gave algorithms for converting modules which satisfy certain conditions (formalised as E1 and E2) in the sequential machine model or the ND sequential machine model, to corresponding definitions in the Set Notation model, and vice versa.

Chapter 6

Universality and implementing modules using concurrency

In this chapter we investigate inversion of networks of modules. We give some universality results for serial modules in the Set Notation model. We then give a series of detailed construction methods and universal sets for the non-arb and eq-arb classes of modules. We also prove a universal set for all Set Notation modules, by utilising a correspondence between the ND sequential machine and Set Notation models defined in the previous chapter. We demonstrate an interesting property inherent to networks of concurrent DI modules, which is that bijectivity of all modules' transition maps can still result in useful irreversible behaviour at the global level. We compare this with a similar but less general result in the literature.

Part of this chapter was published in [53]. This includes the constructions of arbitrary $M{\times}N\,Join$s (Figure 6.4) and $M{\times}N\,Fork$s and the non-arb b-arb and non-arb non-b-arb construction methods (Figures 6.5, 6.6 and 6.7).

We also note that imperative algorithms which follow the non-arb and eq-arb construction methods in this chapter are implemented in the Delay-Insensitive Network Tool Suite program developed in support of this thesis. Please see Chapter 10 for details on this software.

6.1 Inverting modules and networks

Recall (Example 3.27 and Observation 3.28 in Chapter 3) that it is not possible to simply invert any bijective module, such that the resulting module is guaranteed to operate deterministically given a set of input signals, even if safety holds. We show instead an interesting property regarding non-arb non-b-arb modules. These are logically reversible, and as shown previously in Chapter 3, they are a proper subset of all modules which are bijective. This gives these modules, and certain networks

composed of only these modules a special property, which is that of *invertibility* while maintaining deterministic behaviour.

Observation 6.1. Consider some network S containing some non-arb, non-b-arb module $N = (Q, I, O, T)$ in state $q \in Q$, and assume that signals are present on the wires connected to the set of input lines A, and that for some B, q', $(A, B).q'$ is an action of q. As N is non-arb, the module is guaranteed to eventually absorb all signals on the lines A, produce signals on the output lines B, and move to state q'.

Now assume that we take some network S' containing the inverse $N' = (Q', I', O', T')$ of N in state q'. Note that $I' = O$ and $O' = I$. T' is the inverse of T and is still a partial bijection. Assume now that signals are present on the wires connected to the set of (now input) lines B. Note that N' is also non-arb non-b-arb. Hence, due to the inversion of T, the input set B is defined in exactly one action in the current state q' of N' such that $(B, A).q$ is an action of q'. Hence the module N' is guaranteed to eventually absorb all input signals on the input lines B, produce output signals on the output lines A and move to state q, via the same action as before, but now inverted.

Informally, provided that safety and non-clashing holds, the module N' can be seen to have the "inverse behaviour" of N, even when concurrent signals are involved.

Furthermore, assuming that there are no initial signals on the wires of the network, this phenomenon appears to apply transitively to networks of non-arb, non-b-arb modules. This is due to the fact that the sets of input signals applied to each module in the network, the sets of output signals produced from each module, and the resulting state changes are all deterministic (providing safety and non-clashing hold). Inverting all modules within the network maintains backwards-deterministic behaviour in terms of which actions of each module are processed.

Definition 6.2. We say that S is an *invertible* network if it contains only non-arb non-b-arb modules and no signals present on any wires. We define the *inverse* of S to be the network S', where S' is S except:

1. each module is replaced with its inverse in the same state,

2. the directions of all wires are reversed.

Example 6.3. Recall the 4-way *Join* and 4-way *Fork* trees (Example 2.2 in Chapter 2). Assume that these networks are operating under the Set Notation model, and each *Join* and *Fork* module is replaced with its identically named counterpart (Examples 3.12 and 3.16 in Chapter 3 respectively). Note that all modules in the resulting networks are non-arb non-b-arb and both networks are invertible. Furthermore, each network is the other's inverse.

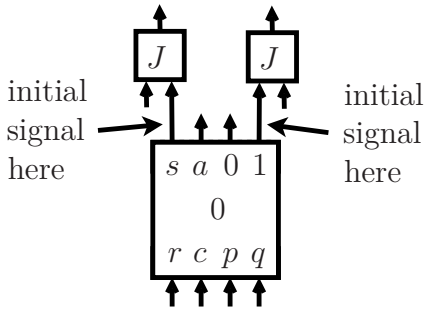


Figure 6.1: A safe non-arb non-b-arb network containing two *Join* modules and a *DM* module in state S_0. Inverting this network results in a network which is unsafe.

Signalling each of the four input lines of the *Join* tree eventually results in an output signal from the network. Similarly, signalling the input line of the *Fork* tree eventually results in a signal on each of the four output lines of the network. In this sense, each network exhibits the inverse behaviour of the other.

We note however that this is not the case in general when there are signals initially present on wires in a network. Sometimes "inverting" such a network may result in non-safety for one of the (now inverted) modules within the network.

Example 6.4. Consider the network in Figure 6.1, which contains two *Join* modules (Example 3.12 in Chapter 3) and the Set Notation counterpart (according to Observation 3.20 in Chapter 3) of the *DM* module (Figure 2.5 in Chapter 2) in state S_0. Note that this network is safe and non-clashing.

Inverting this network will result in a network containing two *Fork* modules (Example 3.16 in Chapter 3), and a single serial module which is the inverse of *DM* (that we note is equivalent to *DM* according to Observation 2.36 in Chapter 2). However, due to the presence of the two signals indicated in the image, the inverted *DM* module will have simultaneous signals on the wires connected to two of its input lines. As this module is serial, this means that the module (and by extension the network) is unsafe.

See Chapter 9 where we exploit the existence of invertible networks in order to design new Self-Timed Cellular Automata which allow the implementation of networks which operate in both the forwards and reverse directions using a single structure.

6.2 Serial universality results

In this section we infer some useful universality results for the classes of reversible serial and serial modules in Set Notation.

Proposition 6.5. The set $\{RT, IRT\}$ is universal for the class of reversible serial modules, where RT and IRT are the Set Notation counterparts (found by Observation 3.20) of the sequential machines RT and IRT (Figure 2.4 in Chapter 2).

Proof. In [48] it is shown that for any reversible serial sequential machine N, a network of sequential machine REs (Example 2.4 in Chapter 2) can be constructed such that the network realises (according to Definition 2.14 in Chapter 2) N in one of its states q. It can also be observed in [48] that the same network can realise the other states of N just by altering the states of the RE modules in the construction (analogously to the concept of a state variation in the Set Notation model). Similarly, in [36] it is shown that RE can be realised in either of its states V or H using a network of RT, IRT, and C-D modules (which we do not define here), and it can also be observed that the same network can realise the other state of RE just by altering the states of the RT, IRT and C-D modules. It is also shown in [36] that a C-D module can be realised in one of its states using a network of RT and IRT modules, and again it can also be observed that the same network can realise all other states of C-D by altering the states of the RT and IRT modules. From this we can conclude that for any reversible serial sequential machine module N, there exists a network S of sequential machine RT and IRT modules which realises N in some state q, and altering the states of the modules can allow the network to realise N in any other of its states.

By Observation 3.20, the behaviour of serial modules in the sequential machine model is not affected by the differences in execution behaviour between the sequential machine model (Definition 2.3 and Condition 3 in Chapter 2) and the Set Notation model (Definition 3.5 in Chapter 3). Set Notation counterparts are considered serial by Definition 3.18. By Definition 2.16 it is necessary that any networks in the sequential machine model containing serial modules must ensure that only one input signal arrives at each serial module at a time. Therefore, replacing each RT and IRT module in S with its Set Notation counterpart (resulting in a network S') and assuming Set Notation execution behaviour (Definition 3.5 in Chapter 3) always results in a safe network. Non-clashing also holds trivially in S' by the requirement of Condition 6 in the sequential machine model. The network S' therefore behaves the same as S, and the network is guaranteed to be safe and non-clashing. The definition of a realisation (Definition 2.14 in Chapter 2) guarantees that the environment of N cannot distinguish N from S. The environment behaviour of a sequential machine serial module is trivially equivalent to the behaviour of any maximal environment of its Set Notation counterpart (which sends a single signal on some valid input line, and then waits for the corresponding output signal). From this we can conclude that N' and S' are indistinguishable in state q, where N' is the Set Notation counterpart of N.

By the same reasoning, we can alter the states of any modules in S in order for the network to realise N in any of its other states q'. Each resulting network possesses a Set Notation counterpart S'', which is a state variation of S', such that N' and S'' are indistinguishable in state q'. This satisfies the conditions for implementation (Definition 4.29 in Chapter 4), and S' implements N'.

Furthermore, all serial modules in the sequential machine model can be expressed in the Set Notation model using a counterpart, and vice versa. Hence any serial module in the Set Notation model can be implemented using a network of the Set Notation modules *RT* and *IRT*. □

Proposition 6.6. The set {*RT, IRT, Merge*} is universal for the class of serial modules, where *RT*, *IRT* and *Merge* are the Set Notation counterparts (found by Observation 3.20) of the sequential machines *RT*, *IRT* and *Merge* (Figures 2.4 and 2.1 in Chapter 2).

Proof. Similarly to Proposition 6.5, in [23] it is shown that for any serial sequential machine N, a network of sequential machine *Select*s and *Merge*s can be constructed such that the network realises (according to Definition 2.14 in Chapter 2) N in one of its states q. It can also be observed in [23] that the same network can realise the other states of N just by altering the states of the modules in the construction (analogously to the concept of a state variation in the Set Notation model). In Figure 2.8 in Chapter 2, it is shown that *Select* can be realised in either of its states S_0 or S_1 using a network of *RDM*s, *Merge*s, and it is shown that the same network can realise the other state of *Select* just by altering the states of the modules. By the same reasoning shown in the proof of Proposition 6.5, *RDM* can be realised in one of its states using a network of *RT* and *IRT* modules, and the same network can realise all other states of *RDM* by altering the states of the *RT* and *IRT* modules. From this we can conclude that for any serial sequential machine module N, there exists a network S of sequential machine *RT*, *IRT* and *Merge* modules which realises N in some state q, and altering the states of the modules can allow the network to realise N in any other of its states.

The remainder of the proof is the same as Proposition 6.5, but it is considered that the sequential machine *Merge* can also be replaced by its Set Notation counterpart without affecting the behaviour of the network. □

6.3 Non-serial universality results and concurrent implementations

Our objective is to identify separate universal sets for different non-serial classes of modules defined in Set Notation, such that modules in a universal set for a class

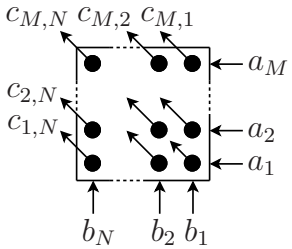
possess desirable properties related to the class. For example, a universal set for non-b-arb modules would ideally contain only modules which are non-b-arb. The importance of creating different sets relates to the implied greater complexity of physical implementation of modules with certain properties (i.e. arb vs. non-arb, b-arb vs. non-b-arb). Recall that it is not simple in the sequential machine model to create different universal sets for various classes of modules which are not serial, as the sequential machine model makes it difficult to classify modules based on their high-level behaviour due to the lack of a distinction between deterministic and non-deterministic behaviour.

We will prove several universal sets for different subclasses of modules in Set Notation. We do this by first introducing general construction methods for both subclasses of non-arb modules. We then show how to extend this to both subclasses of eq-arb modules. We compare our constructions with Keller's method (Figure 2.3 in Chapter 2) and our extension of it (Figure 5.2 in Chapter 5) by noting that those methods require modules which exhibit clear non-deterministic behaviour (such as *ATS*) even when realising modules which exhibit clear deterministic behaviour. Those constructions also processes one signal at a time, in a sequential manner which reflects the behaviour of modules in the sequential machine model. Our new constructions are parallel in nature, thus realising modules' concurrent behaviour more directly. We finish by proving a universal set for all Set Notation modules by utilising the correspondence developed between the Set Notation model and ND sequential machine model in Chapter 5.

We shall use in our main construction arbitrarily-sized $M \times N$ *Join*s (Figure 6.2), where at least one of M, N is greater than or equal to 2. Note that a maximal environment of an arbitrary $M \times N$ *Join* is one which sends a single input signal on some a_i (for some $1 \leq i \leq M$) and a single input signal on some b_j (for some $1 \leq j \leq N$), before waiting for an output signal on $c_{i,j}$. It then repeats this behaviour.

In the remainder of this section, when referring to the module *DM* (Figure 2.5 in Chapter 2), it can be assumed that we are referring to its Set Notation counterpart (found by Observation 3.20 in Chapter 3).

We now demonstrate how to construct arbitrary $M \times N$ *Join*s using only the non-arb non-b-arb set {*DM, Join*}. Figure 6.3 shows how to construct an arbitrary $1 \times N$ *Join* using only {*DM, Join*}. Figure 6.4 shows how to utilise a $1 \times N$ *Join* to construct an arbitrary $M \times N$ *Join*. We compare our construction with that of Keller in [23], and Patra and Fussell in [60] which both utilise *Merge* and therefore do not satisfy any notion of reversibility. Note also that inverting these constructions according to Definition 6.2, which involves replacing *Join* with *Fork* in and relabelling the ports of *DM* (as *DM* is its own inverse by Observation 2.36 in Chapter



$$\begin{aligned} J_{MN} &= (\{a_1, b_1\}, \{c_{11}\}).J_{MN} + (\{a_1, b_2\}, \{c_{12}\}).J_{MN} \\ &+ \ \ldots \ + (\{a_M, b_N\}, \{c_{MN}\}).J_{MN} \end{aligned}$$

Figure 6.2: $M{\times}N\mathit{Join}$ and behavioural specification

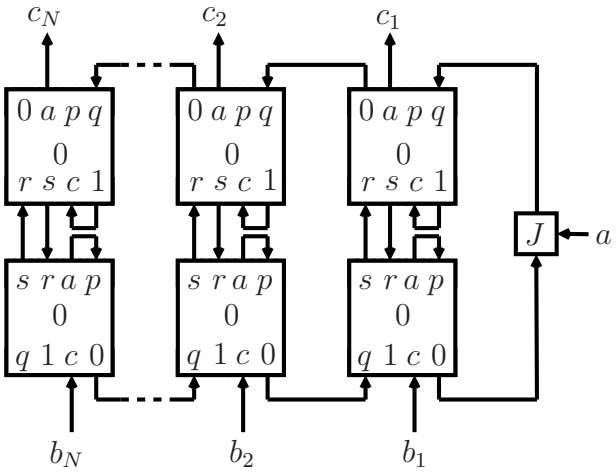


Figure 6.3: Arbitrary $1{\times}N\mathit{Join}$ implemented using $\{\mathit{DM,\ Join}\}$.

2), the inverse of this construction can be achieved, yielding $M \times N\mathit{Fork}$s. Hence the non-arb non-b-arb set $\{\mathit{DM,\ Join,\ Fork}\}$ allows $M \times N\mathit{Join}$s and $M \times N\mathit{Fork}$s of arbitrary size to be constructed.

6.3.1 Universal sets for non-arb modules

In order to construct an arbitrary non-arb module N, we shall use three auxiliary modules defined in terms of N. Firstly, we introduce some helpful functions.

Definition 6.7. Given any Set Notation module $N = (Q, I, O, T)$.

1. Let $\mathit{ISets}_N = \{B : (q, B, q', C) \in T\}$.
2. Let $\mathit{OSets}_N = \{C : (q, B, q', C) \in T\}$.

Informally, ISets_N and OSets_N are the sets of input and output sets which appear in actions of N respectively.

The three auxiliary modules for N are defined as follows.

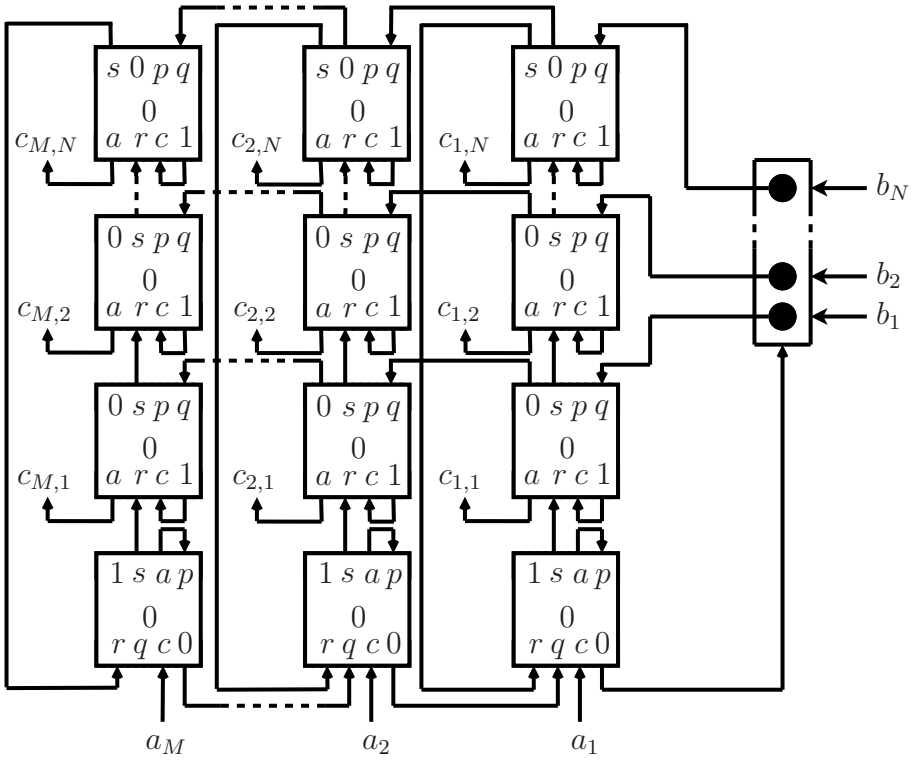


Figure 6.4: Arbitrary $M{\times}NJoin$ implemented using $\{DM,\ 1{\times}NJoin\}$.

Definition 6.8. Given any non-arb module $N = (Q, I, O, T)$, let $Ser_N = (SQ, SI, SO, ST)$ where:

1. $SQ = Q$,
2. $SI = \{I_i : 1 \leq i \leq |ISets_N|\}$,
3. $SO = \{O_i : 1 \leq i \leq |OSets_N|\}$,
4. $ST = \{(q, A, q', B) : (q, mapI_N(A), q', mapO_N(B)) \in T\}$, where $mapI_N$ is any bijection that maps SI to I and $mapO_N$ is any bijection that maps SO to O.

Informally, Ser_N is a serial module which represents the behaviour of N but with input and output sets replaced with singletons. Ser_N is reversible iff N is non-b-arb.

Definition 6.9. Given any non-arb module $N = (Q, I, O, T)$ and $Ser_N = (SQ, SI, SO, ST)$, let $Ser_NQ = (SqQ, SqI, SqO, SqT)$ where:

1. $SqQ = Q$,
2. $SqI = SI \cup \{qi\}$,
3. $SqO = SO \cup \{q_x : x \in Q\}$,

4. $SqT = ST \cup \{(x, qi, x, q_x) : x \in Q\}$.

Informally, $Ser_N Q$ extends the functionality of Ser_N with the ability to query the state of the module on a dedicated set of lines, and this does not modify the state of the module. $Ser_N Q$ is reversible iff N is non-b-arb.

Definition 6.10. Given any non-arb module $N = (Q, I, O, T)$ and $Ser_N = (SQ, SI, SO, ST)$, let $Ser_N Q' = (Sq'Q, Sq'I, Sq'O, Sq'T)$ where:

1. $Sq'Q = Q$,
2. $Sq'I = SI \cup \{q_x : x \in Q\}$,
3. $Sq'O = SO \cup \{qi\}$,
4. $Sq'T = ST \cup \{(x, q_x, x, qi) : x \in Q\}$.

Informally, $Ser_N Q'$ is similar to $Ser_N Q$ but with the query functionality inverted. $Ser_N Q'$ is reversible iff N is non-b-arb.

We help to explain the overall construction method through use of an example. We define a non-arb non-b-arb module P, and its auxiliary modules. We will then describe how to construct P using {*RT, IRT, Join, Fork*}. Our explanation will refer to both the construction of P and the general case for constructing some module N.

Example 6.11. P is given by:

$$\begin{aligned} S_0 &= (\{a,b,c\},\{x,y\}).S_0 + (\{a,c,d\},\{y,z\}).S_1 \\ S_1 &= (\{a,c,d\},\{x,y\}).S_1 + (\{a,b,d\},\{x,z\}).S_0 \end{aligned}$$

The set mappings required for Ser_P, $Ser_P Q$ and $Ser_P Q'$ are:

$$\begin{aligned} mapI_P &= \{(\{I_1\},\{a,b,c\}),(\{I_2\},\{a,c,d\}),(\{I_3\},\{a,b,d\})\} \\ mapO_P &= \{(\{O_1\},\{x,y\}),(\{O_2\},\{y,z\}),(\{O_3\},\{x,z\})\} \end{aligned}$$

Finally:

$$\begin{aligned} Ser_P: \quad S_0 &= (\{I_1\},\{O_1\}).S_0 + (\{I_2\},\{O_2\}).S_1 \\ S_1 &= (\{I_2\},\{O_1\}).S_1 + (\{I_3\},\{O_3\}).S_0 \\ Ser_P Q: \quad S_0 &= (\{I_1\},\{O_1\}).S_0 + (\{I_2\},\{O_2\}).S_1 + (\{qi\},\{q_{S_0}\}).S_0 \\ S_1 &= (\{I_2\},\{O_1\}).S_1 + (\{I_3\},\{O_3\}).S_0 + (\{qi\},\{q_{S_1}\}).S_1 \\ Ser_P Q': \quad S_0 &= (\{I_1\},\{O_1\}).S_0 + (\{I_2\},\{O_2\}).S_1 + (\{q_{S_0}\},\{qi\}).S_0 \\ S_1 &= (\{I_2\},\{O_1\}).S_1 + (\{I_3\},\{O_3\}).S_0 + (\{q_{S_1}\},\{qi\}).S_1 \end{aligned}$$

The construction is divided into two stages. Stage 1 determines the input set which has been signalled and then updates the state of the auxiliary modules in the network accordingly. Stage 2 determines the output set and creates signals on the appropriate output lines. The states of the auxiliary modules remain consistent with each other and correspond to the state of the original module. Stage 1 of the construction of P is shown in Figure 6.5.

In the general case for some module $N = (Q, I, O, T)$, a signal on some input line a_i is forked to several columns of $M{\times}NJoin$s, one for each input set A defined in some action $(A, B).q'$ of the current state of N, such that $a_i \in A$. This is determined by querying an instance of $Ser_N Q$ to determine the current state of N, and hence which columns the signal should be forked to. These columns contain different numbers and sizes of $M{\times}NJoin$s depending on the module being constructed. Each $M{\times}NJoin$ in a given input set's column synchronises an additional input signal. Hence synchronising three signals requires two $M{\times}NJoin$s. We typically synchronise signals in lexicographical order for simplicity, but this is not required. In this example for P it can be observed that the leftmost column, which calculates whether $\{a, b, c\}$ has been signalled, first synchronises signals on a and b, and then following this it synchronises this completed set $\{a, b\}$ with a signal on c.

Eventually exactly one column produces an output on the bottom $M{\times}NJoin$, corresponding to some input set C, for some action $(C, D).q''$ of the current state of N, being satisfied. This signal then removes other signals from the network corresponding to input lines which are part of the C. These are guaranteed to be eventually indefinitely pending on various $M{\times}NJoin$s in other columns. This is achieved by utilising other inputs of the $M{\times}NJoin$s. The order that inputs in a set are synchronised also affects the location of signals which need to be "cancelled", but this is always able to be determined based on the structure of the columns.

In this example for P, if $\{a, b, c\}$ is satisfied in S_0 (the leftmost column), the resulting signal removes other signals corresponding to these input lines from the $M{\times}NJoin$ column corresponding to the (only) other input set in actions of S_0 ($\{a, c, d\}$). This involves removing signals corresponding to a and c from the second column of $M{\times}NJoin$s (but no signals corresponding to b, as there are no sets other than $\{a, b, c\}$ containing b in actions in S_0 of P). However, due to the synchronisation order of the second column, the signals corresponding to a and c, which have been forked to the second column, will eventually be synchronised by the top-most $M{\times}NJoin$ in that column, and hence a single signal corresponding to the completed set $\{a, c\}$ is guaranteed to eventually be pending on the second $M{\times}NJoin$. This signal must be removed. Similarly, if $\{a, c, d\}$ is satisfied in S_0, pending signals corresponding to a on the top $M{\times}NJoin$ and c on the bottom $M{\times}NJoin$ in the leftmost column must be cancelled.

In the general case, it is always possible to cancel other forked input signals either individually or after they have been synchronised, as they will be pending on some fixed combination of $M{\times}N$ *Join*s. As N is non-arb, this is uniquely determined based on the satisfied input set, the structure of the columns, and the current state of N.

After "cancelling" other instances of input signals, the resulting signal is "reversibly merged" using an instance of $Ser_N Q'$, allowing input sets which exist in actions in different states to share a single line. Following this, the resulting signal forks to the input line I_i of all $Ser_N Q$ and $Ser_N Q'$ modules in both stages of the construction, where I_i maps to the satisfied input set C according to $mapI_N(I_i) = C$. This has the effect of updating the state of N (stored consistently across all $Ser_N Q$ and $Ser_N Q'$ modules). Output signals on all O_j of these modules are then synchronised using a *Join* tree. The resulting signal corresponds to an output set of N, and continues to Stage 2.

In this example for P, this can be seen with the set $\{a, c, d\}$, which exists in actions in both S_0 and S_1. Hence two columns, each representing $\{a, c, d\}$ in an action in S_0 and S_1 respectively, are reversibly merged through the middle instance of $Ser_P Q'$. Using a *Fork* tree, the signal then forks to all thirteen instances of $Ser_P Q$ and $Ser_P Q'$ in both stages 1 and 2 (Figure 6.6), inputting on each module's instance of I_2 as $mapI_P(I_2) = \{a, c, d\}$. This updates the state of P according to the action $(\{a, c, d\}, \{y, z\}).S_1$ or $(\{a, c, d\}, \{x, y\}).S_1$ depending on whether the current state of P is S_0 or S_1.

Stage 2 (Figure 6.6 for the example with P) uses a mostly symmetric method to stage 1. In the general case, this is achieved by inverting N, and then following the construction method of stage 1 but we:

1. exclude the "state update" operation (the bottom *Fork* and *Join* trees in Figure 6.5),

2. utilise the existing definitions of $Ser_N Q$ and $Ser_N Q'$ in the construction, rather than those corresponding to the inverse of N.

After this, we mostly invert the construction of this stage, (with $M{\times}N$ *Fork*s replaced with $M{\times}N$ *Join*s, and vice versa) except instead of inverting $Ser_N Q$ and $Ser_N Q'$ modules, we exchange $Ser_N Q$ modules with $Ser_N Q'$ modules and vice versa. This construction for stage 2 is only possible when N is non-b-arb, and hence the inverse of N is non-arb. This allows us to re-use the method for stage 1. If N is non-arb non-b-arb, stages 1 and 2 contain only non-arb non-b-arb modules.

In this example for P, stages 1 and 2 contain only non-arb non-b-arb modules.

This completes the construction for the non-arb non-b-arb module P. Any non-arb non-b-arb module can be constructed by following such a method.

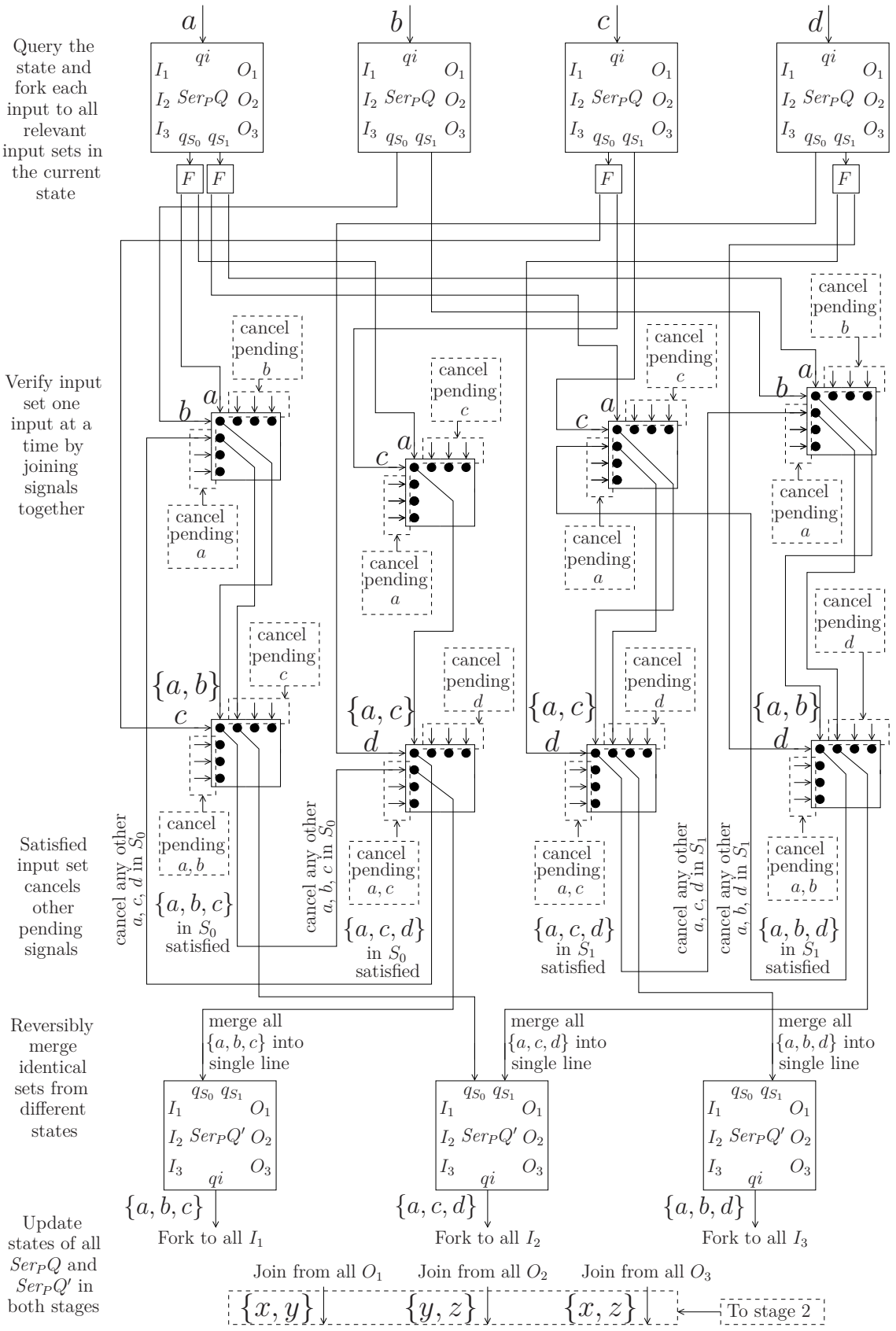


Figure 6.5: Stage 1 (input determination and state update) for P. This stage of the construction contains only non-arb non-b-arb modules.

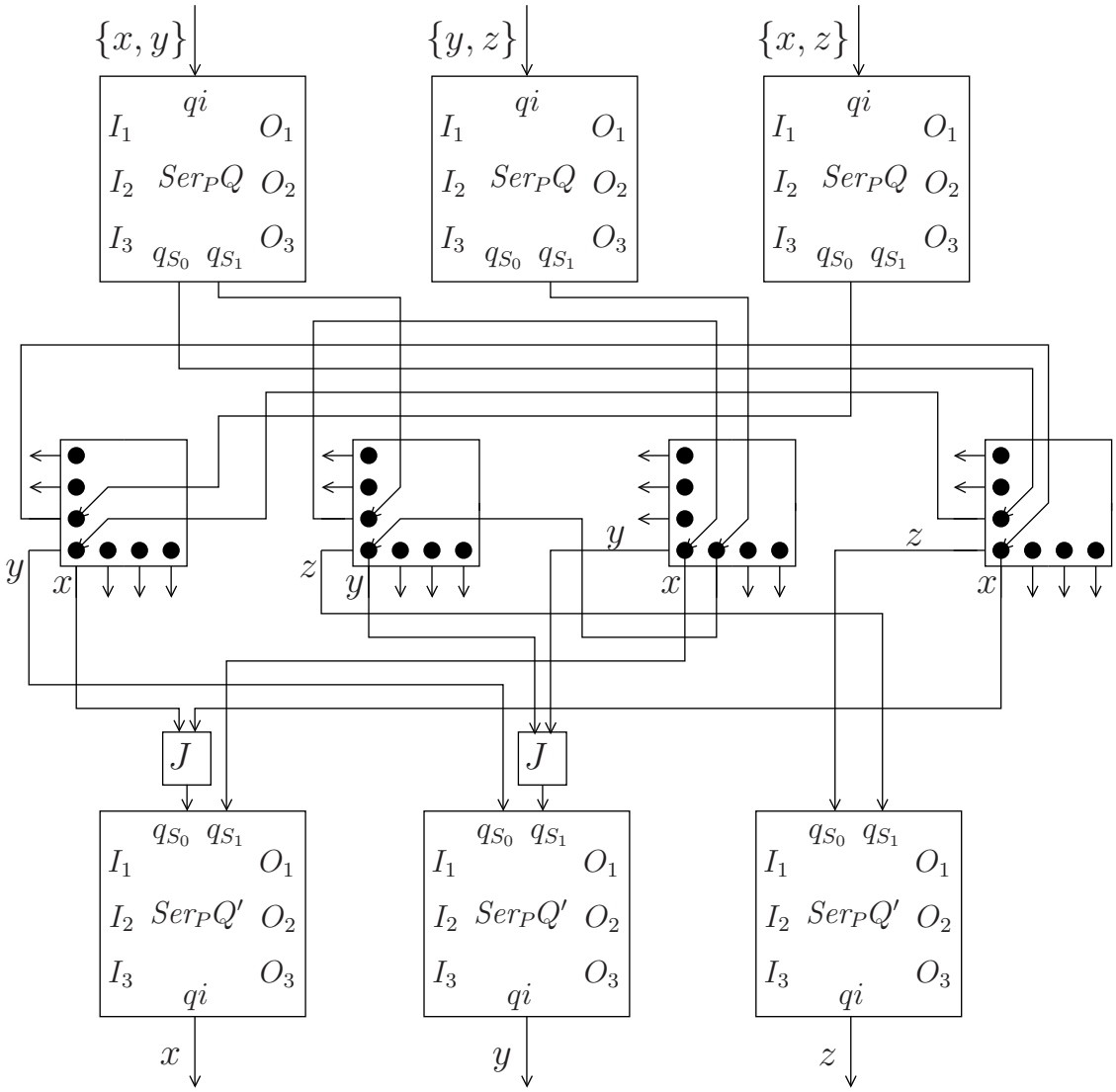


Figure 6.6: Stage 2 (output determination) for P. This stage of the construction contains only non-arb non-b-arb modules.

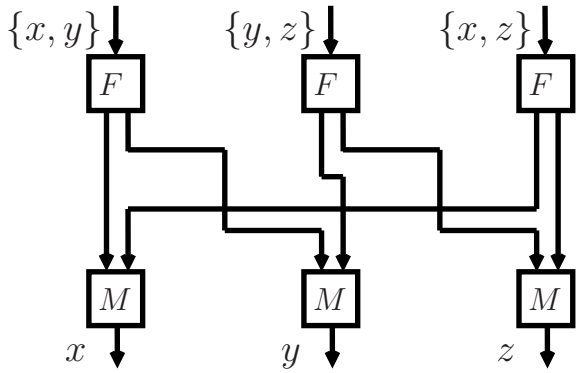


Figure 6.7: Irreversible stage 2 (output determination) for P.

In the general case, an "irreversible" version of stage 2 can be trivially constructed with *Merge*s and *Fork*s, similarly to the bottom section of Figure 2.3 (see Chapter 2). When combined with stage 1, this is used to construct any non-arb module. This version of stage 2 is used when the module is b-arb and the previous technique of inverting N, then following the method for stage 1, is not possible.

Figure 6.7 shows an irreversible stage 2 for P. We note however that this is not necessary as P is non-b-arb and hence we can use the non-b-arb method for generating stage 2 of P (shown in Figure 6.6).

When considering both possible methods for generating stage 2, the construction method can be used to implement any non-arb module. This gives us two universal sets for different classes of non-arb modules.

Theorem 6.12. {*RT, IRT, Fork, Join*} is universal for the class of non-arb non-b-arb modules, and all constructions of such modules contain only non-arb non-b-arb modules. Such constructions are invertible.

Proof. A non-arb non-b-arb module N has reversible serial instances of $Ser_N Q$ and $Ser_N Q'$ which can be implemented using {*RT, IRT*} (Proposition 6.5). Hence non-arb non-b-arb constructions of stages 1 and 2 (Figures 6.5 and 6.6) can be achieved using {*RT, IRT, Fork, Join*} and arbitrarily large $M{\times}N$ *Join*s and $M{\times}N$ *Fork*s. We showed how to construct any $M{\times}N$ *Join* or $M{\times}N$ *Fork* using {*DM, Fork, Join*}. *DM* can also be implemented using {*RT, IRT*} (Proposition 6.5). Such networks do not require any initial signals on wires, and hence are invertible by Definition 6.2. □

Theorem 6.13. {*RT, IRT, Fork, Join, Merge*} is universal for non-arb modules.

Proof. Theorem 6.12 proves that {*RT, IRT, Fork, Join*} can implement any modules in the non-b-arb subclass. Any other module N in this class has instances of $Ser_N Q$ and $Ser_N Q'$ which can be implemented using {*RT, IRT, Merge*} (Proposition 6.6). Hence, constructions of stage 1 (Figure 6.5) can be achieved using {*RT,*

IRT, Fork, Join, Merge} and arbitrarily large $M{\times}N$*Join*s and $M{\times}N$*Fork*s (which can be constructed with {*RT, IRT, Fork, Join*}). Stage 2 can be constructed for any non-arb module using {*Merge, Fork*} (Figure 6.7). □

6.3.2 Universal sets for eq-arb modules

We will show how to modify stage 1 of the construction from the previous section to allow separate realisations of the classes of eq-arb non-arb, and eq-arb b-arb modules.

Recall (Section 5.1.1 in Chapter 5) that we can relabel identical input lines between actions in the same state of an ND sequential machine module such that for all states, no input line appears in more than one action in that state. We follow a similar approach for Set Notation modules, such that the resulting Ser_N generated from an eq-arb Set Notation module N, does not contain multiple actions containing the same input line in any given state. Hence Ser_N remains serial despite N being eq-arb.

Like in Definition 5.4 in Chapter 5, we define some useful auxiliary functions to help in the description of the construction.

Definition 6.14. Given some $Ser_N = (Q', I', O', T')$ for some module $N = (Q, I, O, T)$, the following are quantified over all $a \in I'$ and $q \in Q'$:

1. Let $no(a, q)$ refer to the number of actions in state q that contain input a.
2. Let $max(a)$ be the maximum of all $no(a, q)$.
3. If $max(a) > 1$, let $relabel(a) = a_1, \ldots, a_n$, be any sequence of names where:
   - $n = max(a) - 1$,
   - for all $1 \leq i \leq n$: $a_i \notin I'$ and for all $b \in I'$, where $b \neq a$, $a_i \notin relabel(b)$.

   If $max(a) \leq 1$ then $relabel(a)$ is assumed to be undefined.
4. Let $setAct(a, q) = (a_1, B_1).q_1, \ldots, (a_m, B_m).q_m$, be any sequence given by the set of actions in state q which contain the input line a, where $m = no(a, q)$, $a_i = a$ for all $1 \leq i \leq m$, and no two actions in the sequence are equal. If $no(a, q) = 0$ then $setAct(a, q)$ is assumed to be undefined.

Assume we are given an eq-arb module $N = (Q, I, O, T)$, and we generate $Ser_N = (Q', I', O', T')$ in the usual way described in Definition 6.8. We now show how to modify Ser_N to be a serial module. Assume that for all $a \in I'$ and $q \in Q'$, any valid definitions have been calculated for the functions in Definition 6.14.

1. For all $a \in I'$, we add all $a_k \in relabel(a)$ to I' if $relabel(a)$ is defined.

2. For all states $q \in Q'$, all inputs $a \in I'$ such that $no(a, q) > 1$, and all $0 < i \leq k$ where $setAct(a, q) = (a_0, B_0).q_0, \ldots, (a_k, B_k).q_k$, we relabel a_i in $(a_i, B_i).q_i \in q$ to the jth name in the sequence $relabel(a)$, where $j = i - 1$. This ensures that for each state, no input line appears in more than one action in that state.

Informally, as a result of this modification, each input set of N now instead corresponds to multiple input lines of Ser_N, rather than a single input line of Ser_N as in the case of N being non-arb.

Note that the resulting module is always a serial module. We still generate $Ser_N Q$ and $Ser_N Q'$ from Ser_N in the usual way according to Definitions 6.9 and 6.10.

The modifications to the non-arb construction to accommodate eq-arb modules are similar to those given in Figure 5.2 in Chapter 5, and utilise *Choice* modules (Figure 5.1 in Chapter 5) in order to preserve the non-determination of N. The modifications involve stage 1 only.

In the modified construction:

1. Instead of there existing a corresponding $Ser_N Q'$ module for each input set of the original module N, there exists a corresponding $Ser_N Q'$ module for each input line of Ser_N. Note that some $Ser_N Q'$ still correspond with input sets of N, as some input lines of Ser_N still also correspond with input sets of N.

2. For all $M{\times}NJoin$ columns, if it synchronises some input set A in some state q of N, where A appears in $n > 1$ actions in q of N,

   - instead of connecting the output line of some $M{\times}NJoin$ in some column (corresponding to the current column having finished "removing" pending input signals) to the input line q_x of the $Ser_N Q'$ corresponding to A (where $x = q$), we connect this output line to the input line of a n-way *Choice* tree,
   - we connect any output line of this *Choice* tree to the input line q_x of the $Ser_N Q'$ corresponding to A (where $x = q$),
   - for all remaining output lines b of this *Choice* tree, we connect b to any unconnected q_x of some N_A, where $x = q$, and N_A is an instance of $Ser_N Q'$ which corresponds to some I'_j of Ser_N, such that $I'_j \in relabel(I_j)$ where I_j is the input of Ser_N which corresponds directly with the input set A.

Informally, this ensures that the non-deterministic choice present from signalling A in state q of N is preserved, as the *Choice* tree non-deterministically distributes the

final resulting signal from the $M{\times}N\,Join$ column which synchronises the set of signals corresponding to A, (after removing any pending input signals from other columns) to any of the $Ser_N Q'$ modules corresponding to alternative "choices" resulting from signalling the input set A in state q.

Example 6.15. Consider module P_2 given by:

$$\begin{aligned} S_0 &= (\{a,b,c\},\{x,y\}).S_0 + (\{a,b,c\},\{x,z\}).S_1 + (\{a,c,d\},\{y,z\}).S_1 \\ S_1 &= (\{a,c,d\},\{x,y\}).S_1 + (\{a,b,d\},\{x,z\}).S_0 \end{aligned}$$

This module is similar to P used in the previous section and in Figures 6.5 and 6.6, but instead contains an additional action in S_0 which contains the input set $\{a,b,c\}$. By following the standard method for generating Ser_{P_2} (according to Definition 6.8), it is initially defined as follows.

$$\begin{aligned} S_0 &= (\{I_1\},\{O_1\}).S_0 + (\{I_1\},\{O_3\}).S_1 + (\{I_2\},\{O_2\}).S_1 \\ S_1 &= (\{I_2\},\{O_1\}).S_1 + (\{I_3\},\{O_3\}).S_0 \end{aligned}$$

As I_1 appears multiple times in S_0, a relabelling occurs, and Ser_{P_2} is modified to the following.

$$\begin{aligned} S_0 &= (\{I_1\},\{O_1\}).S_0 + (\{I_1'\},\{O_3\}).S_1 + (\{I_2\},\{O_2\}).S_1 \\ S_1 &= (\{I_2\},\{O_1\}).S_1 + (\{I_3\},\{O_3\}).S_0 \end{aligned}$$

Note that the 2nd instance of I_1 in S_0 has been relabelled to I_1'. $Ser_{P_2}Q$ and $Ser_{P_2}Q'$ are then generated from Ser_{P_2} in the usual way.

The resulting construction of stage 1 for this module is similar to that shown in Figure 6.5 for module P. The differences are as follows.

1. All $Ser_P Q$ and $Ser_P Q'$ modules are replaced in both stages with $Ser_{P_2}Q$ and $Ser_{P_2}Q'$, containing the additional input line I_1'.

2. There exists a new additional instance of $Ser_{P_2}Q'$, such that its output line q_i forks to all I_1' in all $Ser_{P_2}Q$ and $Ser_{P_2}Q'$ in both stages, and all existing *Fork* and *Join* trees also connect appropriately to its input and output lines as with other $Ser_{P_2}Q$ $Ser_{P_2}Q'$ modules.

3. The wire which connects to the input line q_{S_0} of the leftmost $Ser_P Q'$ (now $Ser_{P_2}Q'$) in Figure 6.5, instead connects to the input line of a new 2-way *Choice* module.

4. One output line of the new *Choice* module connects to the q_{S_0} input line of the leftmost $Ser_P Q'$ (now $Ser_{P_2}Q'$) in Figure 6.5.

5. The other output line of the new *Choice* module connects to the q_{S_0} input line of the new additional $Ser_{P_2}Q'$ module.

The method for constructing stage 2 of P_2 is the same as for P (Figure 6.6), except as noted above, all Ser_PQ and Ser_PQ' are replaced with $Ser_{P_2}Q$ and $Ser_{P_2}Q'$ respectively.

The existence of this modified construction method gives us the following result.

Theorem 6.16. {*RT, IRT, Fork, Join, Choice*} is universal for eq-arb non-b-arb modules and {*RT, IRT, Fork, Join, Merge, Choice*} is universal for eq-arb modules.

Proof. Theorems 6.12 and 6.13 prove that {*RT, IRT, Fork, Join*} is universal for non-arb non-b-arb modules and {*RT, IRT, Fork, Join, Merge*} is universal for non-arb modules respectively. We have detailed modifications to stage 1 of the non-arb non-b-arb and non-arb b-arb construction methods to allow the constructions of eq-arb non-arb modules and eq-arb b-arb modules respectively. This involves modifying the Ser_N, Ser_NQ and Ser_NQ' modules generated from the eq-arb module in order for them to remain serial, and hence still able to be implemented by {*RT, IRT*} if they are reversible and {*RT, IRT, Merge*} otherwise. The only additional module required is *Choice*. Hence {*RT, IRT, Fork, Join, Choice*} can be used to construct any eq-arb non-b-arb module, and {*RT, IRT, Fork, Join, Merge, Choice*} can be used to construct any eq-arb module. □

For physical implementation, it may be more desirable to use universal sets which contain only those modules which can be sequentially implemented by sequential machines. Recall that the ND sequential machine version of *Choice* can be realised using *ATS* and *Fork* (Figure 5.3 in Chapter 5). Note that *ATS* is not required to be initialised in state S_0. This allows us to replace the above two sets with the following.

Theorem 6.17. {*RT, IRT, Fork, Join, sATS*} is universal for eq-arb non-b-arb modules and {*RT, IRT, Fork, Join, Merge, sATS*} is universal for eq-arb modules.

Proof. Theorem 6.16 proves that {*RT, IRT, Fork, Join, Choice*} is universal for eq-arb non-b-arb modules and {*RT, IRT, Fork, Join, Merge, Choice*} is universal for eq-arb modules. Recall (Figure 5.3 in Chapter 5) that we have shown how to realise the ND sequential machine version of *Choice* using the sequential machine modules *Fork* and *ATS*. Note that it is possible to replace *Fork* with its Set Notation counterpart (Observation 3.15 in Chapter 3), and *ATS* with *sATS* (Section 3.3 in Chapter 3), and assume Set Notation execution behaviour (Definition 3.5 in Chapter 3) without affecting the overall behaviour of the network. Finally, the Set Notation

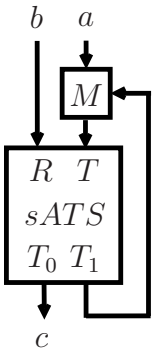


Figure 6.8: *Join* implemented using *sATS* and *Merge*.

version of *Choice* is trivially equivalent to its ND sequential machine counterpart (also by Observation 3.15 in Chapter 3), and has only one state. Hence the Set Notation version of *Choice* can be implemented using {*sATS, Fork*}. □

While these sets only contain modules which can be sequentially implemented by sequential machines, this has the effect of replacing the eq-arb module *Choice* with the more general arb module *sATS*, which may or may not be ideal.

We now show how to replace *Join* in the previous universal sets with *sATS* and *Merge*. See Figure 6.8 where *Join* is implemented using *sATS* and *Merge*. In universal sets where *sATS* and *Merge* are already present, such as the universal set {*RT, IRT, Fork, Join, Merge, sATS*} for eq-arb b-arb modules, this has the effect of reducing the size of the set.

This implies the following universality result.

Proposition 6.18. {*RT, IRT, Fork, Merge, sATS*} is universal for the class of eq-arb modules.

Proof. Theorem 6.17 proves that {*RT, IRT, Fork, Join, Merge, sATS*} is universal for eq-arb modules. Figure 6.8 shows how to implement *Join* using *sATS* and *Merge*. □

We note that as a result of Figure 6.8, it is possible to remove *Join* from the other non-arb universal sets shown in this chapter. However this has the effect of introducing the arb module *sATS* to sets which otherwise contain only non-arb modules.

6.3.3 Universal sets for all modules

We now prove a universal set for all Set Notation modules. This is achieved by utilising the correspondence between Set Notation modules and (ND) sequential

machines, as well as the general construction method for (ND) sequential machines (see Chapter 5).

Recall that {*ATS, Fork, Merge, RT, IRT*} is universal (in the sense of Definition 2.15 in Chapter 2) for (ND) sequential machines (Theorem 5.10 in Chapter 5).

Theorem 6.19. {*sATS, Fork, Merge, RT, IRT*} is universal for all Set Notation modules.

Proof. Recall that we showed that any Set Notation module N can be sequentially implemented using a (ND) sequential machine N', such that any maximal environment of N cannot distinguish between N and N' (Proposition 5.27 in Chapter 5) for all pairs of corresponding states q and $sm(q)$. For any state q' of N', N' can be realised using a network S (following the construction method in Figure 2.3 in Chapter 2 and the modifications in Section 5.1.1 in Chapter 5) containing only the set of sequential machines {*ATS, Fork, Merge, RT, IRT*} (Theorem 5.10 in Chapter 5).

Note that the serial module (which we call N'') in the network S, (which maps its output sets to N'), can be realised using {*RT, IRT*} if it is reversible, and {*RT, IRT, Merge*} otherwise (Proposition 2.33 and Corollary 2.41 in Chapter 2). Note that, as discussed in the proofs of Propositions 6.5 and 6.6, there exists a realisation of N'', such that for any state q'' of N'', it is possible to have the states of its modules modified, such that the resulting network is also a realisation of N'' in state q''.

From this we can conclude that for any N', there exists a network S of sequential machines {*ATS, Fork, Merge, RT, IRT*} which realises N' in some state q, and altering the states of the modules in S can allow the network to realise N' in any of its other states.

Note that the *ATS* modules in S are never required to be initialised in state S_0. Hence it is possible for these modules to be replaced with *sATS* (Section 3.3 in Chapter 3), for *Fork*, *Merge*, *RT* and *IRT* modules to be replaced by their Set Notation counterparts (Observation 3.15 in Chapter 3), and to assume Set Notation execution behaviour (Definition 3.5 in Chapter 3) without affecting the overall behaviour of S. Let S' refer to the modified network containing only the Set Notation modules {*ATS, Fork, Merge, RT, IRT*}.

The network S' therefore behaves the same as S, and S' is also guaranteed to be safe and non-clashing. The definition of sequential implementation (Definition 5.14 in Chapter 5) guarantees that the environment of N cannot distinguish N from N', and by extension N from S or S'. From this we can conclude that N and S' are indistinguishable in state q.

By the same reasoning, we can alter the states of any modules in S in order for the network to realise N' in one of its other states $sm(q''')$ for all q''' of N.

Each resulting network possesses a Set Notation counterpart S'', which is a state variation of S', such that N and S'' are indistinguishable in state q'''. This satisfies the conditions for implementation (Definition 4.29 in Chapter 4), and S' implements N'. □

However, consider that this only shows that a "sequential" style construction exists for any Set Notation module, such that all input signals are forced to be processed one at a time by a serial module inside the construction. This is in contrast to the construction methods given in this chapter, which yield highly parallel networks in order to process multiple inputs concurrently. Hence the only currently known way to construct an arb Set Notation module is to use a highly inefficient serialised implementation which reflects sequential machine behaviour.

Recall also that *Join* is not present in this set. Informally, this implies that all synchronisation operations within any construction using this set are done via *sATS*, as this is the only module in this set which allows two signals to safely arrive concurrently. Note that *Join* is the simplest possible "synchronisation" module, containing only one state, two input lines, a single output line, and one action. The above implementation of *Join* (Figure 6.8) utilises *sATS*, but also requires *Merge*. It is not clear whether an implementation of *Join* exists which does not require *Merge*, and can be achieved using only {*sATS, Fork, RT, IRT*}. However if {*Merge, sATS*} are used in place of *Join*, there will be a large number of non-bijective modules present (due to *Merge*) in implementations of arbitrary modules, even if the module being implemented is bijective. It therefore may be preferable in practice to utilise a *Join* module in order to minimise the number of clear sources of irreversibility in a given construction (which, as discussed in Chapter 1, corresponds to energy dissipation in practice).

6.3.4 Irreversibility from local bijectivity

Recall (Observation 3.28 in Chapter 3) that there is a mismatch between logical reversibility of a module's definition, and the ability to invert a module and obtain forwards-deterministic behaviour. We prove that this phenomenon is sufficient to allow non-trivial irreversible computation.

Recall the definition of $sATS^{-1}$ (Example 3.27 in Chapter 3). Figure 6.9 shows how to implement the *Merge* module using $sATS^{-1}$ and *Join*.

We begin by stating new universality results which are implied by the implementation of *Merge* using $sATS^{-1}$ and *Join*.

Proposition 6.20. The set of bijective modules {$sATS^{-1}$, *Join, RT, IRT*} is universal for the class of serial modules.

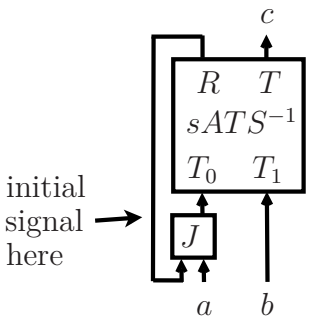


Figure 6.9: *Merge* implemented with $sATS^{-1}$ and *Join*.

Proof. Proposition 6.6 states that {*RT, IRT, Merge*} is universal for the class of serial modules. Figure 6.9 shows that *Merge* can be implemented using {$sATS^{-1}$, *Join*}. □

The full class of Set Notation modules is similarly implied to be realisable using only bijective modules.

Theorem 6.21. The set of bijective modules {*Join, Fork, RT, IRT, sATS,* $sATS^{-1}$} is universal for the full class of Set Notation modules.

Proof. Theorem 6.19 states that {*Merge, Fork, RT, IRT, sATS*} is universal for the class of Set Notation modules. Figure 6.9 shows that *Merge* can be implemented using {$sATS^{-1}$, *Join*}. □

We note that, interestingly, the universal set {*Join, Fork, RT, IRT, sATS,* $sATS^{-1}$} contains three sets of pairs, where each pair represents two modules which are mutual inverses.

As a side result, we briefly note that the sequential machine module *Merge* can be realised using the same network as Figure 6.9, but $sATS^{-1}$ is replaced by its sequential machine counterpart (found according to Observation 3.15), and *Join* is replaced by the sequential machine *Join*. This gives us the following result where $sATS^{-1}$ is the sequential machine counterpart to the Set Notation $sATS^{-1}$.

Corollary 6.22. The set of modules {*ATS, Fork, Join,* $sATS^{-1}$} is universal (in the sense of Definition 2.15 in Chapter 2) for the class of (ND) sequential machines.

Proof. Theorem 5.10 in Chapter 5 states that {*ATS, Fork, Merge, RT, IRT*} is universal for the class of (ND) sequential machines. The sequential machine *Merge* can be realised using the sequential machines $sATS^{-1}$ and *Join*. □

If constructing non-arb non-b-arb modules according to the method demonstrated in Figures 6.5 and 6.6, restricting to the set in Theorem 6.21 would require

that *Join* is substituted with $sATS^{-1}$ and *Merge*, which may not be desirable as this introduces b-arb modules into the construction.

Informally, we note that the only obvious source of irreversibility in the network in Figure 6.9 is the lack of retention of information regarding the timing of signals travelling along wires throughout the network. Consider also that the order in which modules in Figure 6.9 assimilate input signals and produce output signals is fixed. If the input line a is signalled, *Join* always assimilates all input signals and produces an output signal prior to $sATS^{-1}$, followed by no other modules. If the input line b is signalled then $sATS^{-1}$ assimilates this signal and produces two output signals, followed by no other modules. By extension, if an irreversible serial module is implemented using a safe network containing only {*Merge, RT, IRT*} such that all *Merge* modules are then replaced by the implementation using $sATS^{-1}$ and *Join*, the resulting network contains no variations in the order in which modules absorb input signals and produce output signals. This still occurs in a fixed order as outside of the *Merge* implementations, there is a single signal moving throughout the network as the only other modules are the serial modules *RT* and *IRT* and safety is assumed.

These are interesting results for the study of the theoretical limitations of asynchronous circuits. It suggests that a delay-insensitive network has an inherent limitation where the lack of any timing assumptions on its own introduces enough irreversibility to perform useful irreversible computation, regardless of the information-theoretic reversibility (bijectivity) of any modules within the circuit, even if the strong desirable property of safety holds. This is highlighted further when considering the implementation of serial modules, as the set {$sATS^{-1}$, *Join, RT, IRT*} is still universal even though in addition to the above properties, a network can be constructed such that modules assimilate input signals and produce output signals in a fixed order. This rules out the possibility that the cause of this "global" irreversibility is the asynchronous order in which modules process signals relative to each other.

We compare this result with a similar but less general result from [30]. In [30] it is shown that the set of bijective modules {*CDE, T-Join, multiplexer*} is universal for the class of DI modules which are *conservative* when defined using the notation in [62] (where a module is conservative if for all actions $(A, B).q'$ in all states, $|A| = |B|$). However the proof depends on the module *multiplexer* which, when represented using our CCS-like notation, is defined in [30] as follows.

$$e = (S_0, T_0).0 + (S_1, T_1).1$$
$$0 = (I_0, O_0).e$$
$$1 = (I_1, O_1).e$$

It is explicitly stated in [30] that simultaneous input signals are expected on S_0 and S_1 in e, meaning that the set $\{S_0, S_1\}$ may arrive in state e, and an input signal on S_1 (S_0) may also arrive in state 0 (1). When operating in the Set Notation model, this does not satisfy our notion of safety. One may consider altering the definition of *multiplexer* such that it is safe when used as expected under our model. One possible definition of *multiplexer* in our Set Notation model, modified such that signals may arrive on S_0 and S_1 without causing non-safety would be:

$$\begin{aligned} e &= (\{S_0\},\{T_0\}).0 + (\{S_1\},\{T_1\}).1 + (\{S_0,S_1\},\{T_0\}).0S1 \\ 0 &= (\{I_0\},\{O_0\}).e + (\{S_1,I_0\},\{O_0,T_1\}).1 \\ 1 &= (\{I_1\},\{O_1\}).e + (\{S_0,I_1\},\{O_1,T_0\}).0 \\ 0S1 &= (\{I_0\},\{O_0,T_1\}).1 \end{aligned}$$

It can be argued that the behaviour implied by this definition is equivalent to *multiplexer* from [30], and the definition cannot be reduced without reintroducing non-safety during its operation. However, when modelled using this definition, the module is no longer considered bijective (and is also not conservative). Therefore the proof in [30] does not show that the set of modules {*CDE, T-Join, multiplexer*} is for universal for the class of DI modules which are conservative when defined using the notation in [62], while preserving both our notion of safety and bijectivity of its modules at the same time. We conjecture that as a result, the lack of safety of *multiplexer* in [30] introduces additional irreversibility as it is impossible to deduce in which order the "competing" input signals on S_0 and S_1 were assimilated (alternatively, irreversibility is introduced from the non-bijectivity of *multiplexer* if the definition given above is assumed). Compare this with our discovery above, where the only information loss in Figure 6.9 appears to be the timing of signals travelling across wires, and yet this still provides enough expressiveness for full serial universality if used in conjunction with *RT* and *IRT* to implement the set of serial modules.

We also briefly note a similar investigation into this phenomenon in [33], where *Merge* is implemented using *RE*s and *Conservative Join*s, which are similar to *Join*s but contain two output lines instead of one, and produce output signals on both output lines simultaneously. Hence both modules contain bijective transition maps, and when defined using Set Notation are in fact non-b-arb. However, examining the implementation of *Merge* in [33] reveals that safety of the (explicitly serial) module labelled RE_1 is not guaranteed to hold when an input signal is applied to I_1 of the network, followed by receipt of a signal on O by the environment and the sending of an input signal on I_2 of the network.

6.4 Conclusion

In this chapter we investigated inversion of networks of modules. We gave some universality results for serial modules in the Set Notation model. We then gave a series of detailed construction methods and universal sets for the non-arb and eq-arb classes of modules. We also proved a universal set for all Set Notation modules, by utilising a correspondence between the ND sequential machine and Set Notation models defined in the previous chapter. We demonstrated an interesting property inherent to networks of concurrent DI modules, which is that bijectivity of all modules' transition maps can still result in useful irreversible behaviour at the global level. We compared this with a similar but less general result in the literature.

Chapter 7

DI-Set algebra for DI networks

In this chapter we introduce a new process algebra, called *DI-Set algebra*, which is intended to model the behaviour of networks from the Set Notation model. We give examples of encoding modules (such as *Merge*) and networks (such as a network that implements *Merge*) in DI-Set algebra, and define properties of networks discussed in previous chapters (such as safety and non-clashing) more formally in the context of DI-Set algebra. We investigate the use of bisimulation and simulation, and use these together with the aforementioned properties to define more formally the concept of implementation of a module using a network of modules.

We note that a version of DI-Set algebra is implemented in the Delay-Insensitive Network Tool Suite program developed in support of this thesis. Furthermore, the bisimulation and simulation relations defined in this chapter were verified using this software. Please see Chapter 10 for details on this software.

7.1 Syntax and operational semantics

We now introduce DI-Set algebra which will allow us to more formally model delay-insensitive networks. This will let us deduce all possible behaviours of a network and formalise concepts from previous chapters such as safety, non-clashing and more importantly, implementation.

An important feature of DI-Set algebra is that of *module labels*. These allow us to store the set of signals on wires of a network in a single structure (which we call the *communication bus*), while recording which module is the target of each signal. It also allows us to distinguish between modules in the labels of transitions, even if they share the same names of input or output lines. Finally, it allows us to re-use constants (which correspond to CCS-like module state definitions) in order to utilise multiple instances of the same module in a network. For example, we might construct a network containing two *Merge* modules. These would possess different labels (e.g. “1” and “2”). It would then be clear whether a particular signal in the

$$
\begin{array}{rcl}
\langle port \rangle & \models & \langle \text{string} \rangle \\
\langle A \rangle & \models & \emptyset \mid \{\langle port \rangle\} \mid \langle A \rangle \cup \langle A \rangle \\
\langle action \rangle & \models & (\langle A \rangle, \langle A \rangle).\langle D \rangle \mid (\bullet, \langle A \rangle).\langle D \rangle \\
\langle M \rangle & \models & \langle D \rangle \mid \langle action \rangle \mid \langle M \rangle + \langle M \rangle \\
\langle nM \rangle & \models & \langle M \rangle \cdot \langle name \rangle \\
\langle P \rangle & \models & \langle nM \rangle \mid \langle P \rangle | \langle P \rangle \\
\langle name \rangle & \models & \langle \text{string} \rangle \\
\langle nPort \rangle & \models & \langle port \rangle \cdot \langle name \rangle \\
\langle C \rangle & \models & \emptyset \mid \{\langle nPort \rangle\} \mid \langle C \rangle \cup \langle C \rangle \\
\langle wire \rangle & \models & \{(\langle nPort \rangle, \langle nPort \rangle)\} \\
\langle _w \rangle & \models & \emptyset \mid \langle wire \rangle \mid \langle _w \rangle \cup \langle _w \rangle \\
\langle G \rangle & \models & \langle C \rangle \langle _w \rangle \\
\langle L \rangle & \models & \langle P \rangle || \langle G \rangle \\
\langle S \rangle & \models & \langle L \rangle - \langle C \rangle
\end{array}
$$

Figure 7.1: BNF for algebra

bus is an input signal to one of these two modules, as it will have a label associated with it, which could be either "1" or "2".

The structure of our new process algebra is described using *Backus-Naur Form* (*BNF*) [41], which can be seen in Figure 7.1.

The type $\langle \text{string} \rangle$ refers to all possible strings composed only of alphanumeric characters ([a-z],[A-Z],[0-9]) and subscripts. Informally, M represents a *module*, P represents a collection of *named modules*. G represents a *communication bus*, and $_w$ represents the *wire function*. A represents a set of *ports*, C represents a set of *named ports*. L represents a *network*, and S represents a *partially-visible network*. D represents a constant. The $+$ operator represents non-deterministic choice. The $|$ operator composes multiple modules. The $||$ operator composes the set of modules with the communication bus. The $\bullet$ symbol in an action represents that the module has absorbed input signals but not yet produced output signals. Finally, the $-$ operator hides a given set of named ports from being visible in the labels of transitions. It is similar to the hiding operator found in the CSP process algebra [17].

We use the prime operator $'$ and subscripts to refer to different variables of the same type. When referring to $_w$ we generally write w for convenience. We also assume the common set operations of membership ($\in$), union ($\cup$) and difference ($\setminus$) are defined on A, C, w in the usual way. We use k in the following to refer to a variable of type $\langle \text{name} \rangle$. We also require $w(nPort_1) = nPort_2 \iff \{nPort_1, nPort_2\} \in w$.

$$\text{Input} \quad \frac{A \neq \emptyset}{(A, A').D \xrightarrow{?A} (\bullet, A').D}$$

$$\text{Output} \quad \frac{A' \neq \emptyset}{(\delta, A').D \xrightarrow{!A'} D}$$

$$\text{Summation} \quad \frac{M \xrightarrow{*A} M'}{M + M' \xrightarrow{*A} M'}$$

$$\text{Labelled module} \quad \frac{M \xrightarrow{*A} M'}{M \cdot k \xrightarrow{*A \cdot k} M' \cdot k}$$

$$\text{Module collection} \quad \frac{P \xrightarrow{*A \cdot k} P'}{P|Q \xrightarrow{*A \cdot k} P'|Q}$$

$$\text{Enter bus} \quad \frac{}{C'_w \xrightarrow{?C} (C' \cup w(C))_w}$$

$$\text{Leave bus} \quad \frac{}{(C' \cup C)_w \xrightarrow{!C} C'_w}$$

$$\text{Module/Bus} \quad \frac{P \xrightarrow{*C} P', G \xrightarrow{\overline{*}C} G'}{P||G \xrightarrow{*C} P'||G'}$$

$$\text{Network} \quad \frac{L \xrightarrow{*C} L'}{L - C' \xrightarrow{*(C \setminus C')} L' - C'} \bigstar$$

$$\text{Congruence} \quad \frac{S \equiv S', S' \xrightarrow{*C} S'', S'' \equiv S'''}{S \xrightarrow{*C} S'''}$$

Figure 7.2: SOS rules for DI-Set algebra. Let $\overline{!} = ?$ and $\overline{?} = !$, and $* \in \{!, ?\}$. For all $A = \{a_1, \ldots, a_n\}$ and all k, we write $A \cdot k$ as shorthand for $\{a_1 \cdot k, \ldots, a_n \cdot k\}$. Let $\delta \in \{\bullet, \emptyset\}$. The condition $\bigstar$ is, if $C' \setminus C = \emptyset$ then $*(C' \setminus C)$ is τ.

With the exception of Figure 7.2, we do not refer to terms of type L, and we instead always refer directly to components $(P||G)$ of L within the context of a larger term of type S. Hence for simplicity, we refer to partially-visible networks (terms of type S) as network definitions (or simply networks).

To define the behaviour of the syntax, we use *Structured Operational Semantics* (*SOS* for short) [71, 72] which describes the behaviour of a system based on the behaviour of its parts. The SOS of our process algebra is found in Figure 7.2.

We also define several structural congruence equations to provide flexibility when writing terms. These can be found in Figure 7.3.

Typically the set of all behaviours of a network is represented by a *labelled transition system*.

Definition 7.1. A *Labelled Transition System* (*LTS* for short) (S_i, St, T) of a

$$
\begin{array}{ll}
(\emptyset,\emptyset).D \equiv D & (\bullet,\emptyset).D \equiv D \\
P|Q \equiv Q|P & P|(Q|R) \equiv (P|Q)|R \\
M + M' \equiv M' + M & M + (M' + M'') \equiv (M + M') + M'' \\
P||G \equiv G||P & D \equiv M \quad \text{if} \quad D \stackrel{\text{def}}{=} M
\end{array}
$$

Figure 7.3: Structural congruence equations for DI-Set algebra.

network S_i is the smallest such collection of *states* St of type S, and *transitions* $T \subseteq (St \times \{?, !, \tau\} \times C \times St)$, where:

1. $S_i \in St$,
2. $(S', *, C_j, S'') \in T$ iff $S' \stackrel{*C_j}{\rightarrow} S''$ is defined,
3. for all $(S', *, C_j, S'') \in T$, if $* = \tau$ then $C_j = \emptyset$.

The algebra allows us to conveniently encode module definitions given in their CCS-like form directly into the syntax, by defining each state's CCS-like representation as a constant. As discussed previously, we can then instantiate the module by labelling a module's state constant with a unique identifier and then placing it within the larger network term, resulting in a "named" module. This allows us to instantiate a module's state constant multiple times, corresponding to the presence of several instances of the same module in a network. Furthermore, we can encode environment definitions directly within the syntax in the same way as modules. Hence a "module" in DI-Set algebra corresponds with a Set Notation module or environment in a particular state. The wire function w allows us to connect modules and environments within a network, by specifying which output lines of modules are connected to which input lines.

When depicting a network, we do not depict modules which contain actions of the form $(\emptyset,\emptyset).D$ or $(\bullet,\emptyset).D$, instead choosing to rewrite the term using one of the equations from Figure 7.3.

Example 7.2. We show how to encode a single *Merge* module (which we label "1"), with a corresponding maximal environment *EnvM'* (Example 4.14 in Chapter 4), and use the wire function to normally-connect them together. Assume that *Merge* is defined using the constant $M \stackrel{\text{def}}{=} (\{a\},\{c\}).M + (\{b\},\{c\}).M$, analogously to Example 3.16 in Chapter 3. The environment *EnvM'* is defined, analogously to Example 4.14 in Chapter 4, using the constants $EM \stackrel{\text{def}}{=} (\{\},\{a\}).EMa + (\{\},\{b\}).EMa$ and $EMa \stackrel{\text{def}}{=} (\{c\},\{\}).EM$, which we instantiate in state EM with label "E". We define the wire function as $w_1 = \{(a \cdot E, a \cdot 1), (b \cdot E, b \cdot 1), (c \cdot 1, c \cdot E)\}$, which results in a

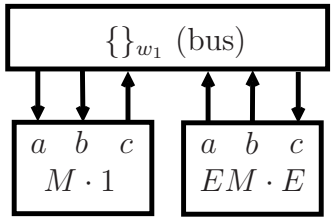


Figure 7.4: Abstract representation of the network containing *Merge* and a corresponding maximal environment *EnvM'* encoded in the algebra as S_1. Signals are sent and received between modules via the bus according to w_1.

normally-connected network. The resulting network term S_1 is as follows.

$$S_1 = M \cdot 1 \mid EM \cdot E \mid\mid \{\}_{w_1} - \{\}$$

The $\{\}_w$ means that there are no signals initially on any wires. Similarly, the $-\{\}$ means that there are no ports being hidden, and hence all signals which are placed or removed from the bus will appear in the labels of transitions. Figure 7.4 shows an abstract representation of the structure of S_1. Informally, the two modules can be considered to be connected to the bus, which is then responsible for the passage of signals between the two modules. A possible series of transitions corresponding to the environment sending an input signal to the module on a, and ultimately receiving an output signal from the module on c is as follows.

$$\begin{aligned}
& S_1 && = M \cdot 1 \mid EM \cdot E \mid\mid \{\}_{w_1} - \{\} \\
\xrightarrow{!\{a \cdot E\}} \quad & S_1^1 && = M \cdot 1 \mid EMa \cdot E \mid\mid \{a \cdot 1\}_{w_1} - \{\} \\
\xrightarrow{?\{a \cdot 1\}} \quad & S_1^2 && = ((\bullet, \{c\}).M) \cdot 1 \mid EMa \cdot E \mid\mid \{\}_{w_1} - \{\} \\
\xrightarrow{!\{c \cdot 1\}} \quad & S_1^3 && = M \cdot 1 \mid EMa \cdot E \mid\mid \{c \cdot E\}_{w_1} - \{\} \\
\xrightarrow{?\{c \cdot E\}} \quad & S_1 &&
\end{aligned}$$

The output transition $!\{a \cdot E\}$ causes a signal $a \cdot 1$ to appear in the bus, due to the relabelling which occurs as a result of the wire function $w_1(a \cdot E) = a \cdot 1$. Similarly, the output transition $!\{c \cdot 1\}$ causes a signal $c \cdot E$ to appear in the bus.

Port hiding can be utilised to hide all "internal" transitions which correspond to the processing of signals by modules within a network.

Example 7.3. We give a modified version of S_1, along with the same series of transitions, except that all ports that are not labelled with "E" are hidden. This causes transitions containing only these ports to become τ, and transitions which contain these ports as well ports with labels other than "E" to contain only the ports with labels other than "E". Let the set of hidden ports be defined by $C_1 =$

$\{a \cdot 1, b \cdot 1, c \cdot 1\}$. The new network is defined as $S_2 = M \cdot 1 \mid EM \cdot E \mid\mid \{\}_{w_1} - C_1$, and the series of possible transitions is as follows.

$$\begin{array}{rl} & S_2 = M \cdot 1 \mid EM \cdot E \mid\mid \{\}_{w_1} - C_1 \\ \stackrel{!\{a \cdot E\}}{\rightarrow} & S_2^1 = M \cdot 1 \mid EMa \cdot E \mid\mid \{a \cdot 1\}_{w_1} - C_1 \\ \stackrel{\tau}{\rightarrow} & S_2^2 = ((\bullet, \{c\}).M) \cdot 1 \mid EMa \cdot E \mid\mid \{\}_{w_1} - C_1 \\ \stackrel{\tau}{\rightarrow} & S_2^3 = M \cdot 1 \mid EMa \cdot E \mid\mid \{c \cdot E\}_{w_1} - C_1 \\ \stackrel{?\{c \cdot E\}}{\rightarrow} & S_2 \end{array}$$

We note briefly that the algebra is flexible enough to allow realisations of several alternative DI models outlined in Section 2.4.2 of Chapter 2. This includes bi-directionality (there is no restriction in the syntax to say that input and output lines cannot share the same name, and we can simply add extra pairs to the w function corresponding to wires simultaneously operating in the reverse direction) and buffering lines (as the bus is a multiset and there is nothing stopping multiple signals on the same line from being present). Finally, the algebra directly supports the modelling of (ND) sequential machines in the syntax, but with the concession that the definition of safety which we define below would not be applicable when considering (ND) sequential machines (as briefly discussed in Chapter 5). These concepts are not explored further as this is beyond the scope of this thesis.

We demonstrate how to model a network containing multiple modules.

Example 7.4. Recall (Figure 6.9 in Chapter 6), that we can implement the *Merge* module using $sATS^{-1}$ and *Join*. The left of Figure 7.5 shows the resulting network after combining Figure 6.9 with *Merge*'s environment *EnvM'* (Example 4.14 in Chapter 4).

We now show how to model this in DI-Set algebra. We define the behaviour of the *Join* and $sATS^{-1}$ modules in DI-Set algebra in the usual way, with the constants $J \stackrel{\text{def}}{=} (\{a, b\}, \{c\}).J$ and $sATS^{-1} \stackrel{\text{def}}{=} (\{T_1\}, \{T\}).sATS^{-1} + (\{T_0\}, \{R, T\}).sATS^{-1}$. Let the instance of *Join* have the label "1", the instance $sATS^{-1}$ have the label "2", and the instance of *EnvM'* have the label "E". Let the wire function w_2 be defined as:

$$w_2 = \{(a \cdot E, b \cdot 1), (b \cdot E, T_1 \cdot 2), (T \cdot 2, c \cdot E), (R \cdot 2, a \cdot 1), (c \cdot 1, T_0 \cdot 2)\}$$

as suggested by the network in the left of Figure 7.5. Let the set of hidden ports be defined as $C_2 = \{a \cdot 1, b \cdot 1, c \cdot 1, R \cdot 2, T \cdot 2, T_0 \cdot 2, T_1 \cdot 2\}$, which corresponds to all non-environment ports being hidden. The network S_3 containing both modules and

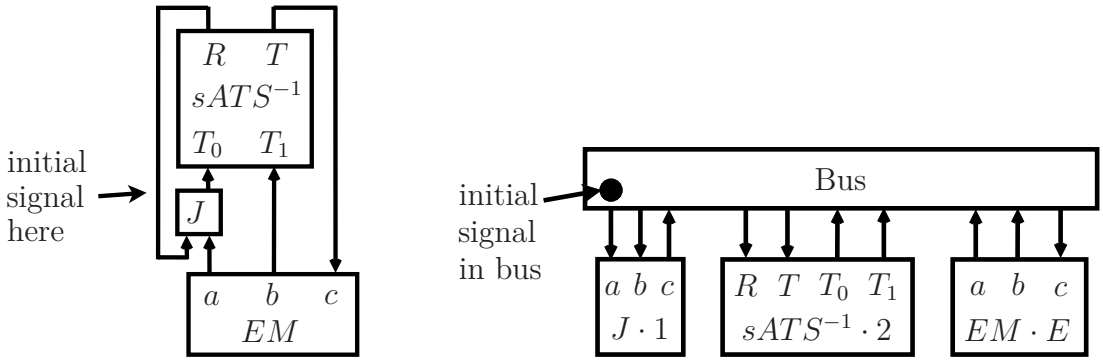


Figure 7.5: (Left) *Merge* implemented with $sATS^{-1}$ and *Join* (as shown in Figure 6.9), combined with *Merge*'s maximal environment *EnvM'* and (right) abstract representation of its encoding in the algebra.

the environment is therefore defined as:

$$S_3 = J \cdot 1 \mid sATS^{-1} \cdot 2 \mid EM \cdot E \mid\mid \{a \cdot 1\}_{w_2} - C_2$$

There is initially a signal present on the wire connected to the input line a of the *Join* (as shown in the left of Figure 7.5), which is modelled by placing the signal $a \cdot 1$ in the bus. The right of Figure 7.5 shows an abstract representation of the structure of S_3. A possible sequence of transitions, again modelling the environment sending an input signal on a and eventually receiving an output signal on c, is as follows.

$$\begin{aligned}
& S_3 && = J \cdot 1 \mid sATS^{-1} \cdot 2 \mid EM \cdot E \mid\mid \{a \cdot 1\}_{w_2} - C_2 \\
\overset{!\{a \cdot E\}}{\rightarrow} \quad & S_3^1 && = J \cdot 1 \mid sATS^{-1} \cdot 2 \mid EMa \cdot E \mid\mid \{a \cdot 1, b \cdot 1\}_{w_2} - C_2 \\
\overset{\tau}{\rightarrow} \quad & S_3^2 && = ((\bullet, \{c\}).J) \cdot 1 \mid sATS^{-1} \cdot 2 \mid EMa \cdot E \mid\mid \{\}_{w_2} - C_2 \\
\overset{\tau}{\rightarrow} \quad & S_3^3 && = J \cdot 1 \mid sATS^{-1} \cdot 2 \mid EMa \cdot E \mid\mid \{T_0 \cdot 2\}_{w_2} - C_2 \\
\overset{\tau}{\rightarrow} \quad & S_3^4 && = J \cdot 1 \mid ((\bullet, \{R, T\}).sATS^{-1}) \cdot 2 \mid EMa \cdot E \mid\mid \{\}_{w_2} - C_2 \\
\overset{\tau}{\rightarrow} \quad & S_3^5 && = J \cdot 1 \mid sATS^{-1} \cdot 2 \mid EMa \cdot E \mid\mid \{a \cdot 1, c \cdot E\}_{w_2} - C_2 \\
\overset{?\{c \cdot E\}}{\rightarrow} \quad & S_3 && = J \cdot 1 \mid sATS^{-1} \cdot 2 \mid EM \cdot E \mid\mid \{a \cdot 1\}_{w_2} - C_2
\end{aligned}$$

We now show the LTSs of S_2 and S_3. We typically depict LTSs by listing all states and numbering them, along with each associated transition and the number of its resulting state. The LTS of S_2 is shown in Figure 7.6, where the top-most state is S_2. We give the LTS of S_3 in Figure 7.7, where the top-most state is S_3. Note that in both networks, all ports not labelled "E" are hidden. Hence in S_3, the only transition labels which are not τ are the same as those in the LTS of S_2 (Figure 7.6).

$$
\begin{aligned}
&0 : M \cdot 1 \mid EM \cdot E \mid\mid \{\}_{w_1} - C_1 \\
&\quad \xrightarrow{!\{a \cdot E\}} 1 \\
&\quad \xrightarrow{!\{b \cdot E\}} 4 \\
&1 : M \cdot 1 \mid EMa \cdot E \mid\mid \{a \cdot 1\}_{w_1} - C_1 \\
&\quad \xrightarrow{\tau} 2 \\
&2 : ((\bullet, \{c\}).M) \cdot 1 \mid EMa \cdot E \mid\mid \{\}_{w_1} - C_1 \\
&\quad \xrightarrow{\tau} 3 \\
&3 : M \cdot 1 \mid EMa \cdot E \mid\mid \{c \cdot E\}_{w_1} - C_1 \\
&\quad \xrightarrow{?\{c \cdot E\}} 0 \\
&4 : M \cdot 1 \mid EMa \cdot E \mid\mid \{b \cdot 1\}_{w_1} - C_1 \\
&\quad \xrightarrow{\tau} 2
\end{aligned}
$$

Figure 7.6: LTS of *Merge* combined with its maximal environment *EnvM'*. The first state (numbered 0) is the network S_2.

$$
\begin{aligned}
&0 : J \cdot 1 \mid sATS^{-1} \cdot 2 \mid EM \cdot E \mid\mid \{a \cdot 1\}_{w_2} - C_2 \\
&\quad \xrightarrow{!\{a \cdot E\}} 1 \\
&\quad \xrightarrow{!\{b \cdot E\}} 6 \\
&1 : J \cdot 1 \mid sATS^{-1} \cdot 2 \mid EMa \cdot E \mid\mid \{a \cdot 1, b \cdot 1\}_{w_2} - C_2 \\
&\quad \xrightarrow{\tau} 2 \\
&2 : ((\bullet, \{c\}).J) \cdot 1 \mid sATS1 \cdot 2 \mid EMa \cdot E \mid\mid \{\}_{w_2} - C_2 \\
&\quad \xrightarrow{\tau} 3 \\
&3 : J \cdot 1 \mid sATS^{-1} \cdot 2 \mid EMa \cdot E \mid\mid \{T_0 \cdot 2\}_{w_2} - C_2 \\
&\quad \xrightarrow{\tau} 4 \\
&4 : J \cdot 1 \mid ((\bullet, \{R, T\}).sATS^{-1}) \cdot 2 \mid EMa \cdot E \mid\mid \{\}_{w_2} - C_2 \\
&\quad \xrightarrow{\tau} 5 \\
&5 : J \cdot 1 \mid sATS^{-1} \cdot 2 \mid EMa \cdot E \mid\mid \{a \cdot 1, c \cdot E\}_{w_2} - C_2 \\
&\quad \xrightarrow{?\{c \cdot E\}} 0 \\
&6 : J \cdot 1 \mid sATS^{-1} \cdot 2 \mid EMa \cdot E \mid\mid \{a \cdot 1, T_1 \cdot 2\}_{w_2} - C_2 \\
&\quad \xrightarrow{\tau} 7 \\
&7 : J \cdot 1 \mid ((\bullet, \{T\}).sATS^{-1}) \cdot 2 \mid EMa \cdot E \mid\mid \{a \cdot 1\}_{w_2} - C_2 \\
&\quad \xrightarrow{\tau} 5
\end{aligned}
$$

Figure 7.7: LTS of *Merge* implementation using $sATS^{-1}$ and *Join*, combined with *Merge*'s maximal environment *EnvM'*. The first state (numbered 0) is the network S_3.

7.2 Properties of networks

We can now formalise certain properties from previous sections in order to establish that desirable conditions hold for networks during execution.

Definition 7.5. We say that an LTS (S_n, St, T) is *deadlocking* iff there exists some state $S_i \in St$ such that for all $(S', *, C', S'') \in T$, $S' \neq S_i$. We call S_i a *deadlock state*. An LTS is non-deadlocking if it is not deadlocking.

Informally, an LTS is deadlocking if it is possible to reach a state where no more transitions are available. The LTSs in Figures 7.6 and 7.7 are non-deadlocking. This has a correspondence with Definition 3.9 in Chapter 3.

Definition 7.6. We say that an LTS (S_n, St, T) is *non-clashing* iff for all $P_i||G_i - C_i \in St$, G_i is a set but not a multi-set. We say it is *clashing* otherwise.

This corresponds directly with Definition 3.10 in Chapter 3. The LTSs of S_2 and S_3 in Figures 7.6 and 7.7 are non-clashing.

Definition 7.7. An LTS (S_v, St, T) is *safe* iff for all $P_l||G_l - C_l \in St$ and for all $0 \leq i \leq n$, where:

1. $P_l = M_0 \cdot k_0| \ldots |M_n \cdot k_n$,
2. $M_i = (A_{i,0}, B_{i,0}).q_{i,0} + \ldots + (A_{i,m}, B_{i,m}).q_{i,m}$.

if $m > 0$ or $A_{i,0} \neq \bullet$, then for some $0 \leq j \leq m$, $G_i \subseteq A_{i,j}$ where $G_i = \{a : (a \cdot k_i \in G_l)\}$.

Informally, this means that an LTS is *safe* if all named modules in the network are safe. It is a further formalisation of Definition 3.11 in Chapter 3. The definition factors in that modules in DI-Set algebra may be in "intermediate" states, having assimilated input signals but not yet produced output signals (i.e. there is only a single action present in the term, with the $\bullet$ symbol in place of an input set). For these intermediate states of a module, the definition ignores the requirement that the set of input signals are a subset of some input set of an action. The LTSs in Figures 7.6 and 7.7 are safe.

Using these properties we can infer some results concerning LTSs in DI-Set algebra. These will correspond with Proposition 4.18 and Theorem 4.19 in Chapter 7.

We first define the previous notion of normal-connectivity (Definition 4.2 in Chapter 4) between a module and a corresponding environment in the context of DI-Set algebra. However in this case, we consider only maximal environments.

Definition 7.8. For all Set Notation modules $N = (Q, I, O, T)$ and for all $q \in Q$, let the constant defined by $CCS(N, q)$ be the CCS-like representation of a Set Notation module $N = (Q, I, O, T)$ in state $q \in Q$. i.e. the list of actions:

$$(A_0, B_0).CCS(N, q_0) + , \ \ldots \ , + (A_n, B_n).CCS(N, q_n)$$

where $(A_i, B_i).q_i \in q$ for all $0 \leq i \leq n$.
Let $CCS(E, q')$ be defined similarly for all environments $E = (Q', I', O', T', N, sc)$ and for all $q' \in Q'$.

Definition 7.9. We define the *normal-execution* (N, q, E) of a module $N = (Q, I, O, T)$ and corresponding maximal environment $E = (Q', I', O', T', N, sc)$ to be the network:

$$S_m = CCS(N, q) \cdot 1 \mid CCS(E, q') \cdot E \mid\mid \{\}_{w_n} - C_n$$

where:

1. $q \in Q$ and $q' = sc(q)$,
2. $w_n = \{(a \cdot 1, a' \cdot E) : a \in O, a' \in I'\} \cup \{(a' \cdot E, a \cdot 1) : a' \in O', a \in I\}$,
3. $C_n = \{a \cdot 1 : a \in (I \cup O)\}$.

This definition corresponds directly with Definition 4.2 in Chapter 4. The new propositions are as follows and correspond directly with Proposition 4.18 and Theorem 4.19 from Chapter 4 respectively.

Proposition 7.10. The LTS for the normal-execution (N, q, E) of a module N and a corresponding maximal environment E is safe and non-clashing.

Proof. By Proposition 4.18. □

Proposition 7.11. The LTS for the normal-execution (N, q, E) of a Set Notation module N which is stable, non-auto-firing, and 1-step-consistent and a corresponding maximal environment E is safe, non-clashing and non-deadlocking.

Proof. By Theorem 4.19. □

We show a useful result regarding infiniteness of LTSs. We first demonstrate the phenomenon through an example.

Example 7.12. Consider the network from Figure 7.8, composed of a single *Fork* and a single *Join* module, with a signal initially present on the wire connected to the input line of the *Fork*. We show how to encode this below using the network S_4,

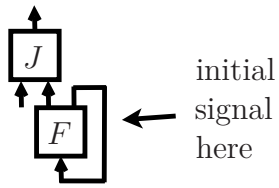


Figure 7.8: A network demonstrating the ability for an infinite number of signals to appear on a wire. The *Fork* module continuously absorbs a signal on its input line, and produces a signal on each of its output lines. This causes the wire connecting the *Fork* module to the *Join* module to repeatedly have signals deposited on it.

where $F \stackrel{\text{def}}{=} (\{a\}, \{b, c\}).F$, $J \stackrel{\text{def}}{=} (\{a, b\}, \{c\}).J$, and $w_3 = \{(c \cdot 1, a \cdot 1), (b \cdot 1, b \cdot 2)\}$. We place an initial signal on the wire connected to the input line of the *Fork* module, by initialising the bus to $\{a \cdot 1\}$. A possible series of transitions starting from S_4 are also shown.

$$
\begin{aligned}
& S_4 && = F \cdot 1 \mid J \cdot 2 \mid\mid \{a, \cdot 1\}_{w_3} - \{\} \\
\stackrel{?\{a \cdot 1\}}{\rightarrow} \quad & S_4^1 && = ((\bullet, \{b, c\}).F) \cdot 1 \mid J \cdot 2 \mid\mid \{\}_{w_3} - \{\} \\
\stackrel{!\{b \cdot 1, c \cdot 1\}}{\rightarrow} \quad & S_4^2 && = F \cdot 1 \mid J \cdot 2 \mid\mid \{b \cdot 2, a \cdot 1\}_{w_3} - \{\} \\
\stackrel{?\{a \cdot 1\}}{\rightarrow} \quad & S_4^3 && = ((\bullet, \{b, c\}).F) \cdot 1 \mid J \cdot 2 \mid\mid \{b \cdot 2\}_{w_3} - \{\} \\
\stackrel{!\{b \cdot 1, c \cdot 1\}}{\rightarrow} \quad & S_4^4 && = F \cdot 1 \mid J \cdot 2 \mid\mid \{b \cdot 2, b \cdot 2, a \cdot 1\}_{w_3} - \{\}
\end{aligned}
$$

Note that the states S_4, S_4^2 and S_4^3 are examples of the network returning to the same state, with the exception that another signal appears on the wire connecting an output line of the *Fork* to an input line of the *Join* (specified as $b \cdot 2$) with each "iteration". As a result, the LTS of S_4 contains an infinite number of states.

We now give the following general result. This assumes that the network S_D contains only Set Notation modules and environments, and these are encoded in the expected way using state constants defined with CCS-like notation.

Proposition 7.13. An LTS (S_D, St, T) contains an infinite number of states in St iff there exists two states $S_i = P_i||G_i - C_k$ and $S_j = P_j||G_j - C_k$, in St such that:

1. $P_i = P_j$,
2. $G_i \subset G_j$,
3. there exists a sequence of transitions leading from S_i to S_j.

Proof. For simplicity, assume that the LTS is depicted in a form similarly to a tree where the initial state S_D is at the top. All states which are reached from the initial state by a single transition are depicted one level underneath; then all states

which are reachable from these states via a single transition are depicted one further level underneath, and so on. However a state cannot appear more than once, and a transition to a state which has already been listed simply connects back to its existing entry, otherwise a new entry is created on the next level below. Consider now that there exists a finite number of transitions starting from each state (i.e. the LTS has finite "breadth" at each level). If there is an infinite number of states in the LTS, there must exist an infinite number of levels (or "depth") to the tree-like representation, and so there must be at least one sequence of transitions starting from the initial node which has infinite length and does not pass through the same state twice. Let p be any such sequence of transitions. Let the set of states visited by following p, (note there is no looping back to an earlier state by following p) be denoted with St_I. Recall that each module in a DI network can be in only a finite number of states. Hence in St_I, there must exist an infinite set of LTS states $St_M \subseteq St_I$ where the states of the modules are equivalent, but they differ only in the contents of the bus. Due to the finite number of wires in the network however, there must be an infinite number of states in St_M where signals are duplicated in the bus. Let $St_B \subseteq St_M$ be any infinite set where the states of the modules are equivalent, but for every pair of states $S_m, S_n \in St_B$, either the bus of S_m is a strict superset of the bus of S_n or the bus of S_n is a strict superset of the bus of S_m. Due to the infinite size of St_B it must be that for any $S_o \in St_B$:

1. there exist a finite number of states $St_S \subset St_B$ such that the bus of S_o is a strict superset of the bus of all $S_p \in St_S$ and hence,

2. there exist infinite number of states $St_L \subset St_B$ such that the bus of S_o is a strict subset of all $S_q \in St_L$.

Note that as St_L is infinite, and there can only be a finite number of states "prior" to any $S_o \in St_B$ in the sequence p, it must be that for all $S_o \in St_B$ there is a path of transitions leading from S_o to all S_r in some infinite set $St_A \subseteq St_L$.

This covers the forwards direction of the two-way implication. For the reverse of the implication, consider also that adding signals to the bus of a network does not "reduce" the number of input or output "transitions" that an individual module may perform. Hence, for all $S_o \in St_B$ it is possible for the same "sequence" of transitions (ignoring starting and ending network states of transitions, but considering simply the labels of transitions) from S_o to some $S_r \in St_A$ to also occur starting from S_r, leading to some S_u where the difference in bus contents between S_u and S_r is the same as the difference in bus contents between S_r and S_o. This can repeat indefinitely, guaranteeing an infinite number of states. □

Informally this means that it is possible for the bus of some state S_i in the LTS of S_D to "grow" infinitely, such that the states of all modules keep returning to the

same as in S_i, but more signals become present in the bus (corresponding to signals on wires) with each iteration. Note that this also implies that if an LTS has an infinite number of states, then it is clashing.

Corollary 7.14. If an LTS has an infinite number of states, then it is clashing.

Proof. By the proof of Proposition 7.13. □

The converse of the implication is also useful to note. If an LTS is non-clashing, then it does not have an infinite number of states. This will simplify our formal notion of implementation, as we will require (like Definition 4.29 in Chapter 4) that networks are non-clashing, thereby guaranteeing that we do not need to accommodate infinite-sized LTSs.

7.3 Bisimulation and simulation

We now examine notions of bisimulation and simulation, in order to facilitate a notion of implementation.

In the following, we define $\stackrel{\alpha}{\Longrightarrow}$ as $(\stackrel{\tau}{\rightarrow})^* \stackrel{\alpha}{\rightarrow} (\stackrel{\tau}{\rightarrow})^*$ if $\alpha \neq \tau$. Otherwise, $\stackrel{\alpha}{\Longrightarrow}$ is simply $(\stackrel{\tau}{\rightarrow})^*$. Let S_X, S_Y be any two network definitions. Bisimulation is defined as follows.

Definition 7.15. Relation R is a *bisimulation* if for all $(S_X, S_Y) \in R$ and all $A \cdot k$:

1. if $S_X \stackrel{!A\cdot k}{\rightarrow} S'_X$ then there exists some S'_Y where $S_Y \stackrel{!A\cdot k}{\Longrightarrow} S'_Y$ and $R(S'_X, S'_Y)$,
2. if $S_X \stackrel{?A\cdot k}{\rightarrow} S'_X$ then there exists some S'_Y where $S_Y \stackrel{?A\cdot k}{\Longrightarrow} S'_Y$ and $R(S'_X, S'_Y)$,
3. if $S_X \stackrel{\tau}{\rightarrow} S'_X$ then there exists some S'_Y where $S_Y \stackrel{\tau}{\Longrightarrow} S'_Y$ and $R(S'_X, S'_Y)$,
4. if $S_Y \stackrel{!A\cdot k}{\rightarrow} S'_Y$ then there exists some S'_X where $S_X \stackrel{!A\cdot k}{\Longrightarrow} S'_X$ and $R(S'_X, S'_Y)$,
5. if $S_Y \stackrel{?A\cdot k}{\rightarrow} S'_Y$ then there exists some S'_X where $S_X \stackrel{?A\cdot k}{\Longrightarrow} S'_X$ and $R(S'_X, S'_Y)$,
6. if $S_Y \stackrel{\tau}{\rightarrow} S'_Y$ then there exists some S'_X where $S_X \stackrel{\tau}{\Longrightarrow} S'_X$ and $R(S'_X, S'_Y)$.

We say that networks S_X, S_Y are *bisimilar* if there exists some bisimulation R such that $R(S_X, S_Y)$.

For convenience, we define relations by referring to the numbered states in a network's LTS.

Example 7.16. The following relation R_1 represents a bisimulation of the two networks S_2 and S_3, whose LTSs are given in Figures 7.6 and 7.7. For each component

(i, j) in R_1, i is the state numbered i from the LTS of S_2, and j is the state numbered j from the LTS of S_3.

$$R_1 = \{(0,0), (1,1), (2,2), (3,3), (3,4), (3,5), (4,6), (5,7)\}$$

This example may suggest that bisimulation is an ideal basis for a notion of equivalence between two networks, and could be used to formally define implementation in the context of DI-Set algebra. However we now argue that this may not be the case.

Recall the definitions of *mATS* (Example 3.39 in Chapter 3) and *fATS* (Example 4.32 in Chapter 4). We have shown that both of these modules possess the same maximal environment *EnvfATS* (Example 4.33 in Chapter 4). Furthermore, each implements the other (Proposition 4.34 in Chapter 4). Informally, this implies that replacing one module with the other in a network does not affect the operation of the network.

We therefore check whether the two networks, each consisting of one of these modules in state S_1 normally-connected to *EnvfATS*, are bisimilar.

Example 7.17. Consider in Figure 7.9 the LTS of the network S_5 (shown as the top state) which is the normal-execution $(mATS, S_1, EnvfATS)$ consisting of module *mATS* in state S_1 together with the maximal environment *EnvfATS* of *mATS* in state $sc(S_1)$. w_5 and C_5 are defined appropriately according to Definition 7.9. Assume that the constants given in Figure 7.9 are defined as follows.

- $mATS1 \stackrel{\text{def}}{=} CCS(mATS, S_1)$
- $ES_1 \stackrel{\text{def}}{=} CCS(EnvfATS, ES_1)$
- $ES_1T \stackrel{\text{def}}{=} CCS(EnvfATS, ES_1T)$
- $ES_1TR \stackrel{\text{def}}{=} CCS(EnvfATS, ES_1TR)$
- $ES_1RS_0 \stackrel{\text{def}}{=} CCS(EnvfATS, ES_1RS_0)$

Example 7.18. Consider in Figure 7.10 the LTS of the network S_6 (shown as the top state) which is the normal-execution $(fATS, S_1, EnvfATS)$ consisting of module *fATS* in state S_1 together with a maximal environment *EnvfATS* of *fATS* in state $sc(S_1)$. Assume that $fATS1 \stackrel{\text{def}}{=} CCS(fATS, S_1)$ and $fATS0 \stackrel{\text{def}}{=} CCS(fATS, S_0)$. Let w_5, C_5, ES_1, ES_1T, ES_1TR and ES_1RS_0 be defined equivalently to in Example 7.17.

$0 : mATS1 \cdot 1 \mid ES_1 \cdot E \mid\mid \{\}_{w_5} - C_5$
$\quad \xrightarrow{!\{T \cdot E\}} 1$
$\quad \xrightarrow{!\{R \cdot E, T \cdot E\}} 6$
$1 : mATS1 \cdot 1 \mid ES_1T \cdot E \mid\mid \{T \cdot 1\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 2$
$\quad \xrightarrow{!\{R \cdot E\}} 6$
$2 : (\bullet, \{T_1\}).mATS1 \cdot 1 \mid ES_1T \cdot E \mid\mid \{\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 3$
$\quad \xrightarrow{!\{R \cdot E\}} 7$
$3 : mATS1 \cdot 1 \mid ES_1T \cdot E \mid\mid \{T_1 \cdot E\}_{w_5} - C_5$
$\quad \xrightarrow{?\{T_1 \cdot E\}} 0$
$\quad \xrightarrow{!\{R \cdot E\}} 4$
$4 : mATS1 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{T_1 \cdot E, R \cdot 1\}_{w_5} - C_5$
$\quad \xrightarrow{?\{T_1 \cdot E\}} 5$
$5 : mATS1 \cdot 1 \mid ES_1RS_0 \cdot E \mid\mid \{R \cdot 1\}_{w_5} - C_5$
$\quad \xrightarrow{!\{T \cdot E\}} 6$
$6 : mATS1 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{R \cdot 1, T \cdot 1\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 7$
$\quad \xrightarrow{\tau} 8$
$7 : (\bullet, \{T_1\}).mATS1 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{R \cdot 1\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 4$
$8 : (\bullet, \{T_0\}).mATS1 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 9$
$9 : mATS1 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{T_0 \cdot E\}_{w_5} - C_5$
$\quad \xrightarrow{?\{T_0 \cdot E\}} 0$

Figure 7.9: LTS of *mATS* combined with its maximal environment *EnvfATS*. The first state (numbered 0) is the network S_5.

$0 : fATS1 \cdot 1 \mid ES_1 \cdot E \mid\mid \{\}_{w_5} - C_5$
$\quad \xrightarrow{!\{T \cdot E\}} 1$
$\quad \xrightarrow{!\{R \cdot E, T \cdot E\}} 6$
$1 : fATS1 \cdot 1 \mid ES_1T \cdot E \mid\mid \{T \cdot 1\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 2$
$\quad \xrightarrow{!\{R \cdot E\}} 6$
$2 : (\bullet, \{T_1\}).fATS1 \cdot 1 \mid ES_1T \cdot E \mid\mid \{\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 3$
$\quad \xrightarrow{!\{R \cdot E\}} 7$
$3 : fATS1 \cdot 1 \mid ES_1T \cdot E \mid\mid \{T_1 \cdot E\}_{w_5} - C_5$
$\quad \xrightarrow{?\{T_1 \cdot E\}} 0$
$\quad \xrightarrow{!\{R \cdot E\}} 4$
$4 : fATS1 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{T_1 \cdot E, R \cdot 1\}_{w_5} - C_5$
$\quad \xrightarrow{?\{T_1 \cdot E\}} 5$
$5 : fATS1 \cdot 1 \mid ES_1RS_0 \cdot E \mid\mid \{R \cdot 1\}_{w_5} - C_5$
$\quad \xrightarrow{!\{T \cdot E\}} 6$
$6 : fATS1 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{R \cdot 1, T \cdot 1\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 7$
$\quad \xrightarrow{\tau} 8$
$\quad \xrightarrow{\tau} 10$
$7 : (\bullet, \{T_1\}).fATS1 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{R \cdot 1\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 4$
$8 : (\bullet, \{T_0\}).fATS1 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 9$
$9 : fATS1 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{T_0 \cdot E\}_w - C_m$
$\quad \xrightarrow{?\{T_0 \cdot E\}} 0$
$10 : (\bullet, \{T_1\}).fATS0 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 11$
$11 : fATS0 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{T_1 \cdot E\}_{w_5} - C_5$
$\quad \xrightarrow{?\{T_1 \cdot E\}} 12$
$12 : fATS0 \cdot 1 \mid ES_1RS_0 \cdot E \mid\mid \{\}_{w_5} - C_5$
$\quad \xrightarrow{!\{T \cdot E\}} 13$
$13 : fATS0 \cdot 1 \mid ES_1TR \cdot E \mid\mid \{T \cdot 1\}_{w_5} - C_5$
$\quad \xrightarrow{\tau} 8$

Figure 7.10: LTS of *fATS* combined with its maximal environment *EnvfATS*. The first state (numbered 0) is the network S_6.

The two LTSs in Figures 7.9 and 7.10 are not bisimilar. Informally, this is because the existence of a bisimulation reflects the ability for two networks to "match" each other's visible transitions at each step. However note that in state 12 of the LTS in Figure 7.10, it is guaranteed that if the environment sends a single signal to *fATS* on T only, then the next visible transition will be an output from T_0 (in state 9). Such a situation never occurs for the LTS in Example 7.17, as whenever the environment sends a single signal to *mATS* on T only, there is always the possibility that the next visible transition will be an output from T_1.

It is possible that bisimulation is therefore a too fine relation to be an appropriate notion of equivalence for DI networks. We define the slightly coarser relation of simulation.

Definition 7.19. Relation R is a *simulation* if for all $(S_X, S_Y) \in R$ and all $A \cdot k$:

1. if $S_X \xrightarrow{!A \cdot k} S'_X$ then there exists some S'_Y where $S_Y \xRightarrow{!A \cdot k} S'_Y$ and $R(S'_X, S'_Y)$,
2. if $S_X \xrightarrow{?A \cdot k} S'_X$ then there exists some S'_Y where $S_Y \xRightarrow{?A \cdot k} S'_Y$ and $R(S'_X, S'_Y)$,
3. if $S_X \xrightarrow{\tau} S'_X$ then there exists some S'_Y where $S_Y \xRightarrow{\tau} S'_Y$ and $R(S'_X, S'_Y)$.

We say that network S_Y *simulates* S_X if there exists some simulation R such that $R(S_X, S_Y)$.

Informally, simulation is like bisimulation but does not require that all visible transitions of the "right" network in the relation must always be "matched" by the "left" network in the relation (but optionally preceded or succeeded with one or more τ transitions).

Definition 7.20. We say that networks S_X, S_Y are *similar* if S_X simulates S_Y, and S_Y simulates S_X.

Two networks are similar if each can simulate the other.

Example 7.21. The following relation R_2 is a simulation between S_5 from Example 7.17 and S_6 from Example 7.18, such that S_6 simulates S_5. For each component (i, j) in R_2, i is the state numbered i from the LTS of S_5, and j is the state numbered j from the LTS of S_6.

$$R_2 = \{(0,0), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7), (8,8), (9,9)\}$$

The following relation R_3 is a simulation between S_6 from Example 7.18 and S_5 from Example 7.17, such that S_5 simulates S_6. For each component (i, j) in R_3, i is

the state numbered i from the LTS of S_6, and j is the state numbered j from the LTS of S_5.

$$R_3 = \{(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,6),(11,6),(12,5),(13,6)\}$$

From this we can conclude that networks S_5 and S_6 are similar.

This raises an interesting question as to whether simulation or bisimulation is more appropriate for DI networks. We note informally that the conditions outlined in Definition 4.27 (see Chapter 4) for indistinguishability appear to reflect those required for simulation.

Consider the above example regarding networks S_5 and S_6 containing *mATS* and *fATS*, with both modules initially in S_1. These networks are not bisimilar as, informally speaking, it is possible for a situation to arise where *fATS* can guarantee an output signal on T_0 as a result of the next input signal on T. Such a situation cannot arise with *mATS*. However, even when considering the module *fATS* only, the environment cannot identify when this particular situation occurs. From the environment's perspective, it is never clear whether it may or may not be possible for an output signal on T_0 to be produced as a result of the next input signal on T. This is why *mATS* and *fATS* possess an identical maximal environment (given as *EnvfATS*).

We believe that this particular property, and the assumption that an environment has limited knowledge of the state of the network as a result of delay-insensitivity, means that simulation is more appropriate as a basis for a notion of "equivalence" between DI networks.

7.4 Implementation

We can now define a notion of implementation in DI-Set algebra. We first define *indistinguishability* between a Set Notation module and network, both encoded in DI-Set algebra. This corresponds directly with Definition 4.27 in Chapter 4.

Definition 7.22. Given a module $N = (Q, I, O, T)$, a network $S_D = P_D \,||\, C'_{w_D} - \{\}$ and some $em = w_I \cup w_O$, where:

1. $P_D = CCS(N_{x,0}, q_{x,0}) \cdot k_{x,0} \mid \ldots \mid CCS(N_{x,n}, q_{x,n}) \cdot k_{x,n}$,
2. for all $0 \leq i \leq n$, $N_{x,i} = (Q_{x,i}, I_{x,i}, O_{x,i}, T_{x,i})$ and $q_{x,i} \in Q_{x,i}$,
3. all $k_{x,0}, \ldots, k_{x,n}$ are distinct and $k_{x,i} \neq E$ for all $0 \leq i \leq n$,
4. $w_I = \{(a \cdot E, b \cdot k') : a \in I$ and for some $0 \leq i \leq n$, $b \in I_{x,i}$ and $k' = k_{x,i}\}$,

5. $w_O = \{(b \cdot k', a \cdot E) : a \in O$ and for some $0 \leq i \leq n$, $b \in O_{x,i}$ and $k' = k_{x,i}\}$,

we say that N and S_D are *indistinguishable in state* $q \in Q$ *via em* if given any maximal environment $E' = (Q', I', O', T', N, sc)$ of N, the network S'_D and the normal-execution (N, q, E') are similar and, the LTS of (N, q, E') and the LTS of S'_D are safe and non-clashing, where:

1. $S'_D = P_D \mid CCS(E', q') \cdot E \mid\mid C'_{w_V} - C_V$
2. $sc(q) = q'$,
3. $C_V = \{a \cdot k' :$ for some $0 \leq i \leq n$, $a \in (I_{x,i} \cup O_{x,i})$ and $k' = k_{x,i}\}$,
4. $w_V = w_D \cup em$ is a partial bijection.

Note that the above definition requires that no module in the network S_D is labelled with "E", as this is "reserved" for the label of an environment.

Example 7.23. Let $S'_3 = J \cdot 1 \mid sATS^{-1} \cdot 2 \mid\mid \{a \cdot 1\}_{w_N} - \{\}$ where $w_N = \{(R, \cdot 2, a \cdot 1), (c \cdot 1, T_0 \cdot 2)\}$, and the constants J and $sATS^{-1}$ are defined as in Example 7.4. Let $em = \{(a \cdot E, b \cdot 1), (b \cdot E, T_1 \cdot 2), (T \cdot 2, c \cdot E)\}$.

Note that S_2 (Figure 7.6) is a normal-execution of *Merge* in its only state M with corresponding maximal environment *EnvM'*. Furthermore S'_3 is S_3 (Figure 7.7) from Example 7.4 with the environment *EnvM'* (encoded as $EM \cdot E$) and associated wires em removed (as $w_N = w_2 \setminus em$).

Since, S_2 and S_3 are shown to be bisimilar in Example 7.16 (which means that they are also similar), *Merge* and S'_3 are indistinguishable in state M via em.

Recall (Section 4.3 in Chapter 4) that indistinguishability can be viewed as a state-specific notion of implementation. Hence, correspondingly to Definition 4.29 in Chapter 4, we also define the concept of implementation in DI-Set algebra, which is independent of any starting state.

Definition 7.24. Given a network $S_I = P_I \mid\mid C'_{w_D} - \{\}$ where:

1. $P_I = CCS(N_{x,0}, q_{x,0}) \cdot k_{x,0} \mid \ldots \mid CCS(N_{x,n}, q_{x,n}) \cdot k_{x,n}$,
2. for all $0 \leq i \leq n$, $N_{x,i} = (Q_{x,i}, I_{x,i}, O_{x,i}.T_{x,i})$ and $q_{x,i} \in Q_{x,i}$,
3. all $k_{x,0} \ldots k_{x,n}$ are distinct, and $k_{x,i} \neq E$ for all $0 \leq i \leq n$,

we say that S_I *implements* module $N = (Q, I, O, T)$ iff there exists some em where, S_I and N are indistinguishable in state q via em for some $q \in Q$, and for all $q' \in Q$ there exists some $S'_I = P'_I \mid\mid C''_{w_F} - \{\}$ where:

1. $w_F = w_D$,

2. $P'_I = CCS(N_{y,0}, q_{y,0}) \cdot k_{y,0} \mid \ldots \mid CCS(N_{y,m}, q_{y,m}) \cdot k_m$ where $m = n$, and for all $0 \leq j \leq n$, $N_{y,j} = N_{x,j}$, $k_{y,j} = k_{x,j}$ and $q_{y,j} \in Q_{x,j}$,

3. S'_I and N are indistinguishable in state q' via em.

This corresponds directly to Definition 4.29 in Chapter 4. It can also be seen to formalise the notion of a state variation (Definition 3.8 in Chapter 4), as the network S'_I is found by modifying the "states" of "modules" in the term S_I, as well as the contents of the bus (corresponding to the presence of signals on wires).

We note that if a module N has only one state q, and a network S_I is indistinguishable from N in q via some em, then trivially, S_I implements N.

Example 7.25. Since *Merge* has only one state (defined by the constant M), and the network S'_3 from Example 7.23 is indistinguishable from *Merge* in state M via w_m, then S'_3 implements *Merge*.

Informally, when considering Definition 4.29 in Chapter 4 which concerns the abstract setting of Set Notation networks, and Definition 7.24 above which concerns encoding Set Notation networks in DI-Set algebra, it is clear that we have strong formal notions of what it means for a network of modules to implement a module. We believe that such definitions are rigorously defined, and there is no ambiguity as to what constitutes an implementation. Compare this with the notion of a realisation (Definition 2.14 in Chapter 2), which uses informal language and is potentially open to interpretation.

7.5 Conclusion

In this chapter we introduced a new process algebra, called *DI-Set algebra*, which is intended to model the behaviour of networks from the Set Notation model. We gave examples of encoding modules (such as *Merge*) and networks (such as a network that implements *Merge*) in DI-Set algebra, and defined properties of networks discussed in previous chapters (such as safety and non-clashing) more formally in the context of DI-Set algebra. We investigated the use of bisimulation and simulation, and used these together with the aforementioned properties to define more formally the concept of implementation of a module using a network of modules.

Chapter 8

Background to STCA

In this chapter we give an introduction to *Self-Timed Cellular Automata* and related concepts, including the definitions of important properties such as *local reversibility* and *local determinism*. We define new versions of *global reversibility* and *global determinism*, which are related to similar properties in the literature. However these new versions are more flexible properties which are based on notions of convergence, and are appropriate for STCAs which simulate concurrent DI networks. We also give an example of an existing STCA in the literature.

Material in this chapter has been published in [54, 55].

8.1 Introduction

Definition 8.1. A *Self-Timed Cellular Automaton* (*STCA*), introduced in [65], is a special type of asynchronous cellular automaton. It is given by a set of *update rules* together with a two-dimensional infinite array of *cells*. In Figure 8.1, adopted from [32], a cell is depicted as a square (for example, the square containing triangles with a, b, c, d). Each cell is divided into four subcells which are depicted in Figure 8.1 as small triangles, each of which can be in one of two states, 0 or 1. We depict the state 0 with a clear triangle, and the state 1 with a black triangle. The default state of a subcell is 0 and is known as the *quiescent* state. The state of all cells and their subcells in the two-dimensional array is known as a *configuration*, and the state of cells and subcells in the initial array is called the *initial configuration*. Configurations are ranged over by $C, C' \ldots$ and $D, D' \ldots$. In this thesis we identify an STCA with the set of its update rules R.

A configuration in the context of STCA is not to be confused with the definition of a configuration in Chapter 4, which concerns normally-connected networks composed of a single module and a corresponding environment.

Definition 8.2. A set of subcells in a configuration may be involved in an *up-*

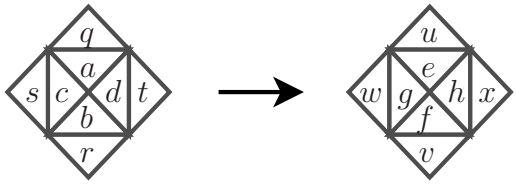


Figure 8.1: Depiction of an update rule.

date, where the states of subcells are modified according to one of the *update rules*. An update involves a full cell (comprised of its four subcells) together with the cell's four adjacent neighbouring subcells on the two-dimensional plane. Figure 8.1 shows how a general update rule is depicted (image adopted from [32]), where $a, b, c, d, e, f, g, h, q, r, s, t, u, v, w, x \in \{0, 1\}$. This means that subcells a, b, c, d, q, r, s, t of a configuration are updated to e, f, g, h, u, v, w, x, respectively, giving a new configuration. Following [65], an update of a set of subcells may only occur if an update rule is defined for the current state of the given subcells.

Subcells are assumed to update instantaneously and randomly at any time if a corresponding update rule is defined. However, as two adjacent cells share subcells in their update codomain, we assume, following [66], that no two adjacent cells may update simultaneously. The issues regarding simultaneous updating of adjacent cells are discussed in [66].

Definition 8.3. A set of update rules is *locally reversible*, if no two update rules have identical right-hand sides. A set of update rules is *locally deterministic* if no two update rules have identical left-hand sides. An update causes the current configuration to change to a new configuration. An *execution* of a configuration C is a sequence of configurations $C \to C' \to C'' \ldots$, where $\to$ represents that one or more updates have occurred simultaneously. The reflexive and transitive closure of $\to$ is denoted by $\to^*$. Configuration C' is *reachable* from C if $C \to^* C'$, and we call C' a *derivative* of C.

In this thesis we assume that configurations give rise only to those executions that satisfy weak fairness [25].

Definition 8.4. An execution $C_1 \to C_2 \to \ldots$, with $C = C_1$, is *weakly fair* whenever if:

1. there are different C_k, C_l in the execution with $k < l$ such that $C_{l+1} = C_k$ (a "loop" containing C_k and C_l is reachable from C),

2. there is $D \neq C_i$, for all $k \leq i \leq l$, such that $C_j \to D$, for some $k \leq j \leq l$, (D is not one of the configurations of this loop and the execution can leave the loop by updating to D),

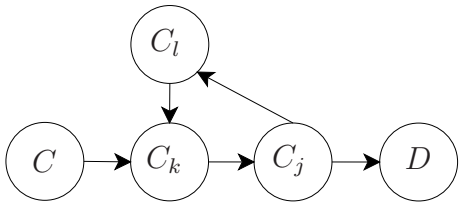


Figure 8.2: A weak-fair execution. Each node represents a unique configuration.

then $C_m = D$ for some $m > l$ (the execution leaves the loop eventually).

Informally, once an execution reaches a loop, if it is possible to break from the loop by updating to a configuration outside the loop, then the configuration will be reached eventually. An example of this can be seen in the execution graph in Figure 8.2, where each node represents a unique configuration. Weak fairness allows us to guarantee that a module eventually produces a set of output signals in response to a set of valid input signals, as is assumed by the execution behaviour (Definition 3.5 in Chapter 3) of the Set Notation model.

Next, we define two new important properties of STCAs.

Definition 8.5. Let C be a configuration of an STCA.

1. C is *globally deterministic* if there exists a configuration D such that, for all C', if $C \to^* C'$, then $C' \to^* D$.

2. C is *globally reversible* if there exists a configuration D' such that, for all C', if $C' \to^* C$, then $D' \to^* C'$.

Informally, if C is globally deterministic then all executions from C must eventually reach the configuration D required by Definition 8.5. Correspondingly, if C is globally reversible then all executions to C originate from some configuration D' as required in Definition 8.5. This is a modification of global reversibility defined in [32], and is made here to accommodate for parallel signals travelling through an STCA and looping execution sequences.

Remark 8.6. There is some similarity between the global determinism and global reversibility of configurations in Definition 8.5 and the *Forward Diamond* (FD) and *Reverse Diamond* (RD) properties of Labelled Transition Systems in [68, 69] (not to be confused with the LTSs from Chapter 7). There is also some similarity between global determinism and the *global confluence* property defined in [24].

For the purpose of this thesis, we assume that configurations are finite two-dimensional arrays such that the four "edges" of the grid are rows of cells which are not involved in updates. As the configurations simulate DI network behaviour, this

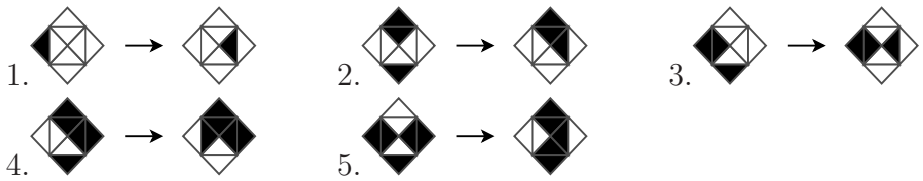


Figure 8.3: The five classes of update rules for STCA from [32]. Rotation-symmetric equivalences are omitted. The rules represent, from left-to-right: 1. Signal propagation; 2. Right turn; 3. Alternative right turn; 4. Left turn; 5. Memory toggle.

has the effect of causing what are analogous to signals to "stop" at the edges of the grid. This allows us to more easily satisfy the above notions of global reversibility and global determinism.

8.2 Existing CA for DI networks

We briefly discuss some existing cellular automata intended for the implemented of DI networks.

We first give an example of an existing STCA intended for the implementation of networks of reversible serial modules.

Example 8.7. The particular STCA we note here is the one given in [32]. Rules are locally deterministic and locally reversible. It is a relatively simple STCA with 2 states per subcell and containing only 5 different classes of rules. Each class contains a rule along with its three rotation-symmetric equivalences. The 5 classes of rules are shown in Figure 8.3, but rotation-symmetric equivalences are omitted.

Other examples of STCAs developed for the simulation of networks of reversible serial modules can be found in [30, 36, 37]. The STCAs in [30, 37] each utilise four classes of rotation-symmetric rules. Furthermore, each STCA contains locally reversible and locally deterministic rules. The STCA in [36] requires five classes of rotation-symmetric and reflective-symmetric rules. Its rules are locally deterministic but are not locally reversible.

We also note the *Partitioned Cellular Automaton* (*PCA*) in [27] which is similar to a STCA but has a smaller codomain for the update function, and three states per subcell. This PCA implements the class of sequential machine modules, but its rules are not locally reversible. Finally, we note the ACA in [74] that simulates the boolean *NAND* gate via the implementation of sequential machine modules.

8.3 Conclusion

In this chapter we gave an introduction to the concepts related to *Self-Timed Cellular Automata*, including the definitions of important properties such as *local reversibility* and *local determinism*. We defined new versions of *global reversibility* and *global determinism*, which are related to similar properties in the literature. However these new versions are more flexible properties which are based on notions of convergence, and are appropriate for STCAs which simulate concurrent DI networks. We also gave an example of an existing STCA in the literature.

Chapter 9

Implementation in STCA and direction-reversibility

In this chapter we introduce four novel STCAs for implementing DI networks, including two STCAs for reversible serial and non-arb non-b-arb networks. The two main STCAs have several very useful properties, they are locally deterministic, locally reversible and support what we call *direction-reversibility*. This allows us to operate a network in reverse by changing the direction of signals and utilising its output lines as input lines (and vice versa). This removes the need for separate constructions to implement the inverse of a network. The new notions of global determinism and global reversibility are proven to hold for these two STCA. We also introduce two further extensions to the STCAs which simulate irreversible serial and non-arb b-arb networks. These two additional STCAs are shown to be locally deterministic and globally deterministic. Finally, we prove that the third and fourth STCAs can be used to implement any module in either Set Notation or the ND sequential machine model.

Recall (Section 6.1 in Chapter 3) that inverting a network of non-arb non-b-arb modules results in a network of non-arb non-b-arb modules. Our objective in this chapter is to exploit this property to aid in simplicity of implementation.

Material in this chapter with the exception of Theorems 9.26 and 9.27 has been published in [54, 55]. We also note that the various STCA and constructions introduced in this chapter are also implemented in the STCA Simulator program developed in support of this thesis. Please see Chapter 10 for details on this software.

In this chapter, all modules and classes can be assumed to refer to the Set Notation model unless otherwise stated.

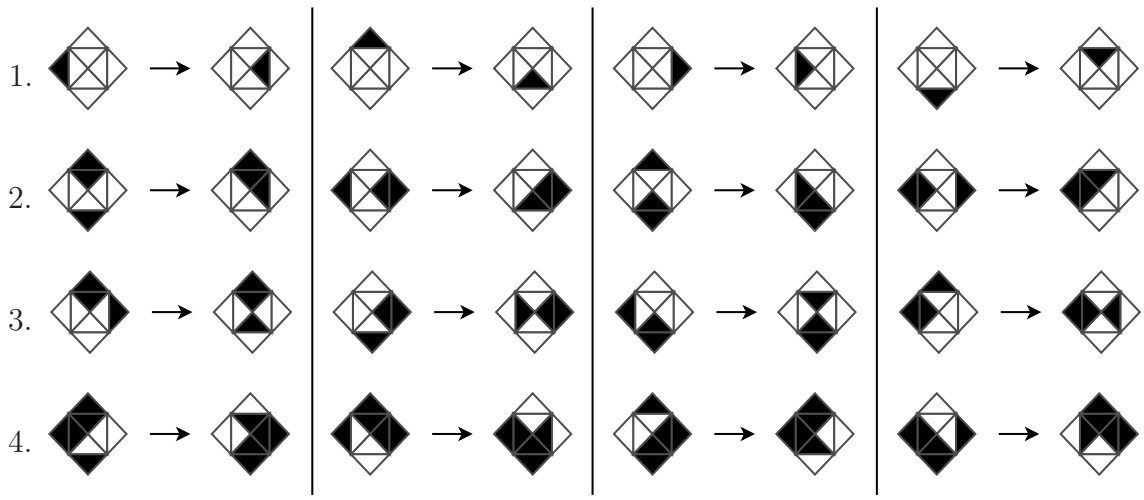


Figure 9.1: The set of rules RS for reversible serial modules. Each class 1-4 consists of a single rule (on the left) together with three of its rotations (on the right). The classes represent the following behaviours: 1. Signal movement; 2. Signal right turn; 3. Signal left turn (direction-reversal of class 2); 4. Toggle of a *memory structure* by a signal.

9.1 Direction-reversible STCA for serial modules

In Figure 9.1, we give four different rules along with three rotations by the multiples of 90 degrees. Hence each line in Figure 9.1 represents an equivalence class of rules. In the context of DI networks, a signal is represented by a single subcell in state 1, with the subcell adjacent to its longest side in the quiescent state. Signals are considered to "point" in the direction perpendicular and away from the subcell's longest side such that the path does not pass through the subcell.

Definition 9.1. Let the set of rules in Figure 9.1 (and the STCA defined by this set) be referred to as RS (for reversible serial).

Definition 9.2. Given an update rule r, $\delta(r)$ is the update rule obtained from r by:

1. *inverting* r, namely swapping the left and right-hand sides,
2. *inverting the direction of the signal* in the resulting rule, namely swapping the states of subcells inside the squares given by the following pairs (q, a), (c, s), (r, b), (d, t), (u, e), (g, w), (v, f), (h, x) in Figure 8.1.

The rule $\delta(r)$ is called the *direction-reversal* (rule) of r and, given the set of rules R, $\delta(R)$ is the set of direction-reversal rules of the rules in R. An STCA with the set of rules R is *direction-reversible* if $R = \delta(R)$.

Example 9.3. The direction-reversal of rule 2 in Figure 9.1 is rule 3, namely $\delta(2) = 3$, and vice-versa. Also, $\delta(1)$ is the third rule in class 1 and $\delta(4)$ is the third rule in class 4.

Proposition 9.4. Let $r \in RS$. Then $\delta(r) \in RS$ and, hence $\delta(RS) = RS$.

Proof. By Definition 9.2. □

Observation 9.5. Proposition 9.4 implies that for each construction in RS which performs an operation on a single signal, inverting the direction of the signal in the rules has the same effect as using the inverses of rules.

Moreover, we have the following result.

Proposition 9.6. Rules in RS are locally reversible and locally deterministic.

Proof. By Definition 8.3. □

Figure 9.2 shows a single construction which acts as either RT or IRT, depending which lines are used as inputs. This is a consequence of the direction-reversibility of RS. Informally, we say that such constructions are *direction-reversible*.

Definition 9.7. We say that a construction implementing a module N is *direction-reversible* if it also implements N^{-1}, and:

1. the input lines to the construction corresponding to the input lines of N also correspond to the output lines of N^{-1}, and

2. the output lines from the construction corresponding to the output lines of N correspond to the input lines of N^{-1}.

Figure 9.2 implies that RS can be used to simulate any reversible serial module.

Theorem 9.8. Any reversible serial module can be implemented by a configuration in RS. Such configurations, and their derivatives, are globally deterministic, globally reversible, and direction-reversible.

Proof. Proposition 6.5 in Chapter 6 states that any reversible serial module can be implemented using a network of RT and IRT modules. Composing multiple instances of the construction in Figure 9.2 allows a realisation of such a network. Local reversibility and local determinism of the rules, together with the presence of a single signal in the network, results in a unique execution sequence, thus guaranteeing global reversibility and global determinism. Direction-reversibility of RT/IRT, local reversibility and local determinism guarantee direction-reversibility. □

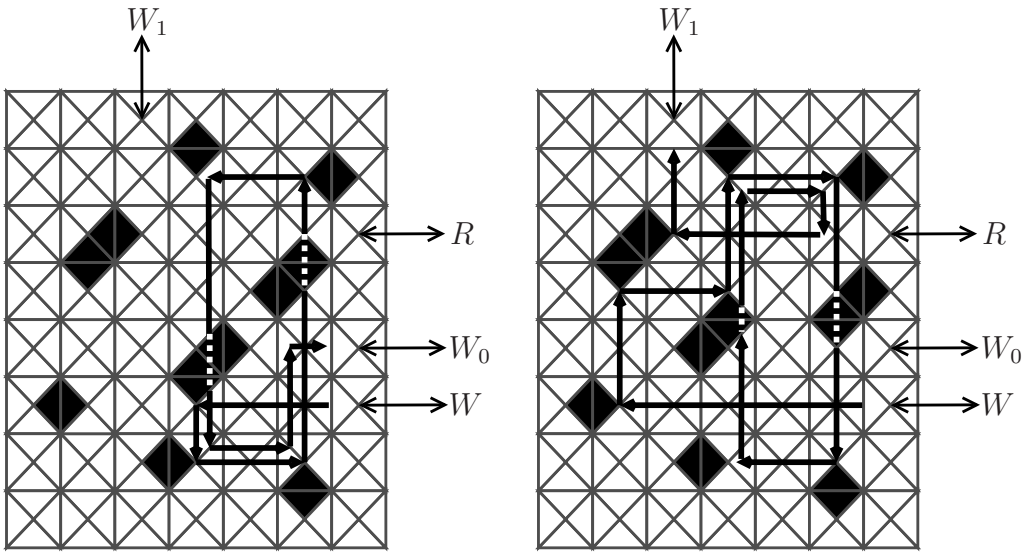


Figure 9.2: Left: *RT/IRT* in state 0. Right: *RT/IRT* in state 1. When R and W are used as input lines, the construction acts as *RT* with W_0 and W_1 as output lines, and vice-versa for *IRT*. The images show the path of a signal on W when the construction is used as *RT*. A dotted white line through a memory structure indicates that the memory's state is toggled and the signal continues in the same direction.

Observation 9.9. As both *RT* and *IRT* are implemented by a single construction, separate constructions are not required to implement the inverse of a network which implements a reversible serial module. Inverting a network in *RS* requires simply changing the direction of the signal. This bidirectional nature implies a potential advantage when considering physical implementation.

Our STCA *RS* uses only four classes of rotation-symmetric rules. The only STCA for implementing reversible serial networks which utilises four classes of rotation-symmetric rules that we are aware of appears in [30, 37], but direction-reversibility is not supported. The STCA in [36] supports a notion of direction-reversibility, but the rules are not locally reversible or locally deterministic. We are not aware of any other STCA which supports a notion of direction-reversibility. Furthermore, the STCA in [36] requires five classes of rotation-symmetric and reflective-symmetric rules.

We now demonstrate a useful construction in *RS*. In [3] it is shown that any irreversible function (which we call I) of the type $Input \rightarrow Output$ can be converted to a reversible function which simulates I. In order to be left with a garbage-less result, this requires a complex series of operations A, B and C (see Figure 9.3), where:

1. A is a reversible version of the original irreversible function I which performs the operation of I while recording the computation history,

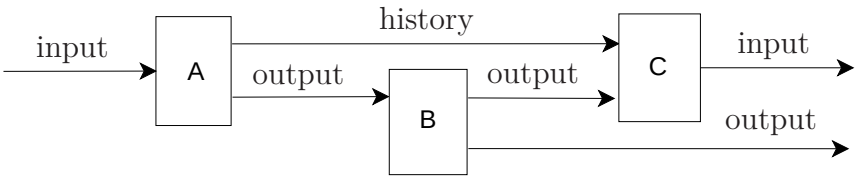


Figure 9.3: Reversible garbage-less implementation of an irreversible function I. A is the reversible version of I, B clones the output of A and C is the inverse of A.

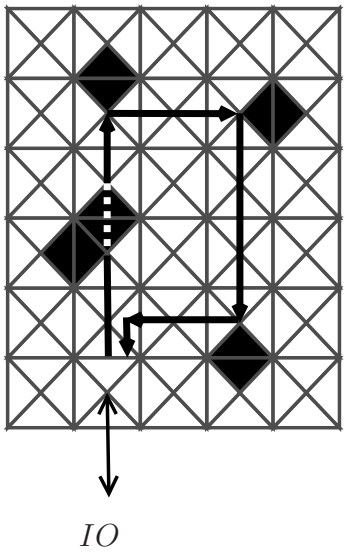


Figure 9.4: A *Direction-Reverser*. An input signal on IO causes a change in the state of the memory structure (with a white dotted line) and an output on IO.

2. B is a function which copies the output (excluding the computation history),

3. C is the inverse of A which removes the computation history and the first copy of the output, while reproducing the original input. This results in a new function of the type $Input \to (Input + Output)$ which simulates I.

Example 9.10. In order to facilitate implementations of these functions, in Figure 9.4 we show a construction called *Direction-Reverser* (*DR* for short) which takes an input signal on IO, toggles the internal state, and then produces an output on IO. Hence, this construction can be used to reverse the direction of a signal. If *DR*s are placed at the output lines of a network as in Figure 9.5, due to the direction-reversible nature of *RS*, an input signal to the network will eventually result in the output being "recorded" in one of these constructions, and the signal being returned to the original input line (but facing the opposite direction). The internal state of the rest of the network is also returned to its original configuration. Hence, when implementing reversible versions of irreversible functions in *RS*, it suffices to implement only the function A.

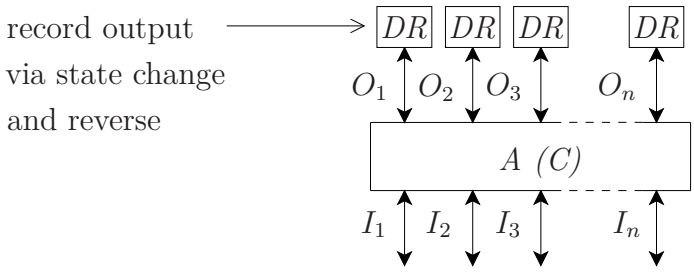


Figure 9.5: Direction-reversible STCA implementation of a reversible version of an irreversible function in Figure 9.3. The main part of the network implements both A and C.

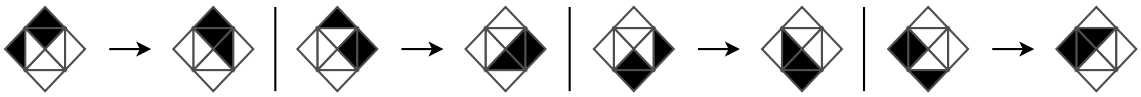


Figure 9.6: The set of rules M. The rotation-symmetric equivalences are included for ease of implementation, but are not required if turns (classes 2 and 3 in RS) are utilised.

Definition 9.11. Let the set of rules in Figure 9.6 be referred to as M.

Informally, the new rules extend the left/right-turn structure so that a signal approaching from the previously unused side is (irreversibly) forwarded to the opposite side. Hence the left/right turn structure can now operate directly as a *Merge* module.

Proposition 9.12. Rules in M are locally deterministic.

Proof. By Definition 8.3. □

Definition 9.13. Let the STCA S be $RS \cup M$ (where S is for serial).

The set S supports the reversible constructions demonstrated in this section. Interestingly, direction-reversibility is maintained for all of these constructions. However, attempting to perform a direction-reversal on constructions which utilise *Merge* (which is an irreversible serial module) may result in unexpected behaviours.

Theorem 9.14. Rules in S are locally deterministic. Any serial module can be implemented by a configuration in S. Such configurations, and their derivatives, are globally deterministic. Configurations which implement reversible serial modules, and their derivatives, are direction-reversible.

Proof. By Proposition 6.6 in Chapter 6, {*RT, IRT, Merge*} is universal for the class of serial modules. Composing multiple instances of the construction in Figure 9.2 and *Merge* structures allows us to implement any serial module. Local determinism

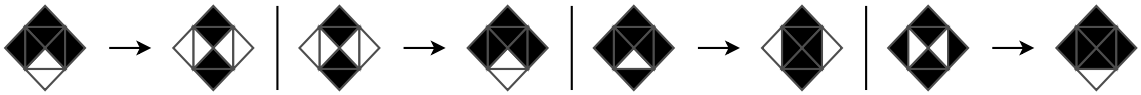


Figure 9.7: The set of rules P. The rules represent from left to right: (p1) *Fork* to *Join* evolution; (p2) *Join* to *Fork* evolution; (p3) *Fork* to *Join* evolution while an input produces two outputs; (p4) *Join* to *Fork* evolution while two inputs produce an output. In the third rule (p3), an input signal (the bottommost black subcell in the source of the rule) arrives at a *Fork*, produces two outputs (the leftmost and the rightmost black subcells in the target of the rule) and changes the configuration to a *Join*. In the last rule (p4), two input signals (the leftmost and the rightmost black subcells in the source of the rule) arrive at a *Join*, produce an output (the bottommost black subcell in the target of the rule) and change the configuration to a *Fork*.

of the rules, together with the presence of a single signal, results in a unique execution sequence, thus guaranteeing global determinism. The behaviour of the RT/IRT construction is unaffected by the additional rules in M. Hence, by Theorem 9.8, compositions of RT/IRT are direction-reversible. □

9.2 Extending to non-serial DI modules

We now show how RS and S can be extended to implement networks that contain more than one signal at a time. Recall that {*RT, IRT, Fork, Join*} is universal for the class of non-arb non-b-arb modules (Theorem 6.12 in Chapter 6). Hence it suffices to add rules which will allow us to implement *Fork* and *Join*. These are given in Figure 9.7.

Definition 9.15. Let the set of rules in Figure 9.7 be referred to as P (for "parallel").

Proposition 9.16. Rules in P are locally deterministic and locally reversible.

Proof. By Definition 8.3. □

Remark 9.17. In order to maintain local reversibility and local determinism, rotation equivalent rules in P are not permitted. So, there is a design constraint on the layout of such networks, and it implies that *Fork* and *Join* constructions must be oriented appropriately in order to function correctly. This is overcome by using left/right turn constructions when designing networks. We note that the direction-reversal version of each rule in P is also in P, thus $\delta(P) = P$.

Example 9.18. Figure 9.8 shows a *Fork* and *Join* construction in P. As *Fork* and *Join* are each other's inverses, they can be constructed with a single direction-reversible construction. This is achieved by two simple structures that evolve constantly, one into the other. The central structure is an *evolving structure* (as opposed

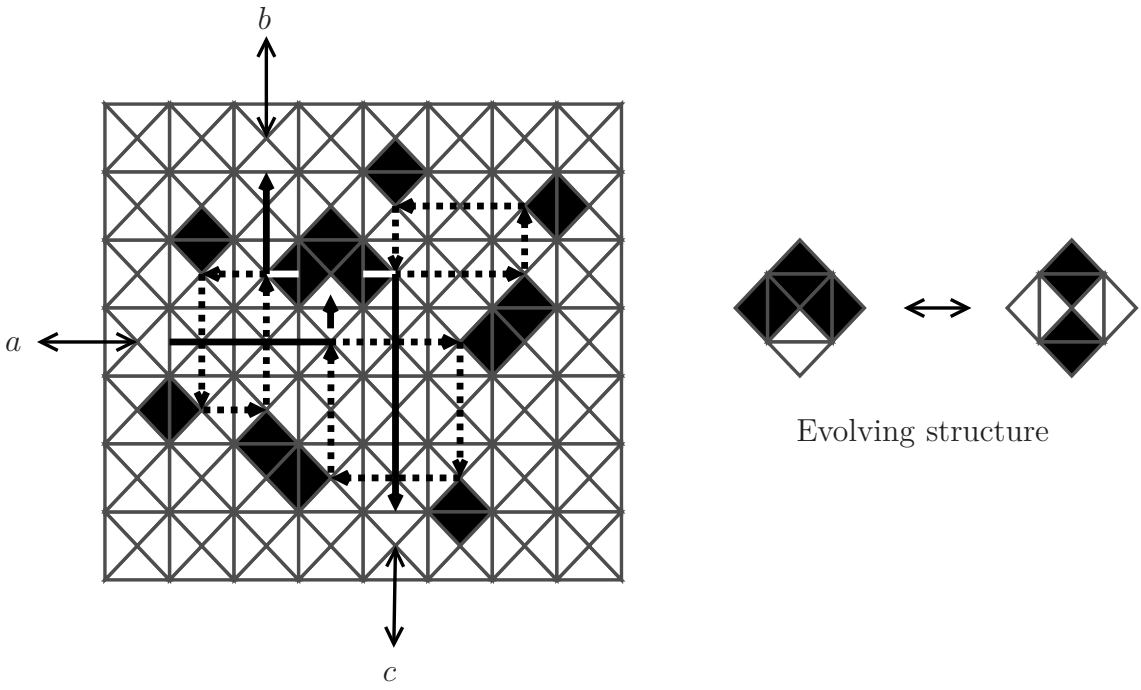


Figure 9.8: *Fork* and *Join*. The central structure evolves between *Fork* and *Join* patterns according to p1 and p2 in Figure 9.7. When a (correspondingly b, c) is used as an input line, the construction acts as *Fork* (*Join*) with b and c (correspondingly a) as output lines. The image shows the path of signals when used as a *Fork*. Signals loop along the dotted lines until they encounter the evolving structure in the correct state. This will eventually happen due to the weak fairness assumption (see Chapter 8).

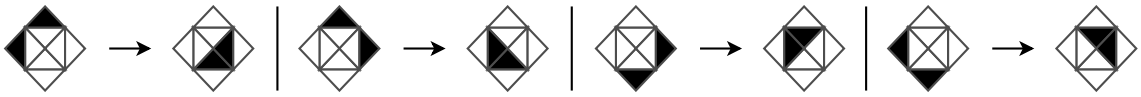


Figure 9.9: The set of rules C (for crossing) to implement the crossing of signals.

to previously seen *static structures*), as updates can occur continuously even when no signals are present. It can be verified that the construction is globally deterministic and globally reversible when signals are applied as intended.

Figure 9.9 contains rules C for crossing of signals. Note that $\delta(C) = C$.

Definition 9.19. Let the set of rules in Figure 9.9 be referred to as C (for "crossing").

Proposition 9.20. Rules in C are locally deterministic and locally reversible.

Proof. By Definition 8.3. □

We now define our third STCA.

Definition 9.21. Let *NANBP* be $RS \cup P \cup C$, (where *NANBP* is for "non-arb non-b-arb parallel").

Since the construction for *Fork/Join* in Figure 9.8 is direction-reversible, globally deterministic and globally reversible when operated with appropriately placed signals, it is easy to see that when combining the construction with that of *RT/IRT* and connecting lines appropriately as in networks of Set Notation modules, the resulting configuration is also direction-reversible, globally deterministic and globally reversible.

Theorem 9.22. Rules in *NANBP* are locally deterministic and locally reversible. Any non-arb non-b-arb module can be implemented by a configuration in *NANBP*. Such configurations, and their derivatives, are globally deterministic, globally reversible and direction-reversible.

Proof. By Theorem 6.12 in Chapter 6, {*RT, IRT, Fork, Join*} is universal for the class of non-arb non-b-arb modules (by following the construction method in Figures 6.5 and 6.6 in Chapter 6). Composing multiple instances of the constructions in Figures 9.2 and 9.8 allows an implementation of such a network. Global determinism and global reversibility of both types of construction, together with the delay-insensitive property of the general construction (Figures 6.5 and 6.6 in Chapter 2) and the lack of any initial signals on wires (see Section 6.1 in Chapter 6), guarantee that such a network is also globally deterministic and globally reversible when used as an implementation of a non-arb non-b-arb module. Direction-reversibility, globally determinism and globally reversibility of *RT*/*IRT* and *Fork*/*Join*, together with local reversibility and local determinism guarantee direction-reversibility. □

We now define our fourth and final STCA.

Definition 9.23. Let *NAP* be $RS \cup P \cup C \cup M$ (for "non-arbitrating parallel").

Correspondingly to Theorem 9.14, we have the following.

Theorem 9.24. Rules in *NAP* are locally deterministic. Any non-arb module can be implemented by a configuration in *NAP*. Such configurations, and their derivatives, are globally deterministic. Configurations which implement non-arb non-b-arb modules, and their derivatives, are direction-reversible.

Proof. By Theorem 6.13 in Chapter 6, {*RT, IRT, Fork, Join, Merge*} is universal for the class of non-arb modules (by following the construction method in Figures 6.5 and 6.7 in Chapter 6). Composing multiple instances of the constructions in Figure 9.2, Figure 9.8 and *Merge* structures using the rules in Figure 9.6 allows an implementation of such a network. Global determinism of these constructions,

together with the delay-insensitive property of the general construction (Figures 6.5 and 6.7 in Chapter 2), guarantee that such a network is globally deterministic. The behaviour of the RT/IRT and $Fork/Join$ constructions is unaffected by the additional rules in M. Hence, by Theorem 9.22, compositions of RT/IRT and $Fork/Join$ are direction-reversible. □

We will finish by demonstrating a useful construction which implements the sequential machine module ATS and related modules.

Example 9.25. In Figure 9.10, we show how the STCAs $NANBP$ and NAP can implement the sequential machine module ATS (Figure 2.1 in Chapter 2). The construction contains two consistent memory structures which can hold one of two values, 0 or 1 (represented by the grey and corresponding black subcells). The held values of 0 and 1 correspond to ATS in state S_0 and S_1 respectively. An input signal on R may only arrive when the memory structures hold a value of 1. In this case it can be verified that the signal sets the two memory structures to 0 and pends indefinitely at the $Join$ structure to the right of the construction. An input signal on T will query the value in the upper memory structure. A value of 1 will cause the signal to leave on the T_1 output line. A value of 0 will cause the signal to enter the $Join$ structure, synchronising with the pending R signal, before resetting the values to 1 and outputting on the T_0 output line. A race condition may arise between an input signal on T attempting to query the upper memory, and an output signal on R attempting to modify the value. A collision will not occur as it is assumed (see Chapter 8) that no two adjacent cells may update simultaneously. It can be verified that this construction correctly implements the behaviour of ATS.

Note that, due to the correspondence between ATS and the Set Notation module $sATS$ (Section 3.3 in Chapter 3) this construction can also be seen to implement $sATS$ in S_1, if we set the held value to 1 (which also corresponds to S_1 of ATS).

Furthermore, if the held value is set to 1 (corresponding to S_1 of both ATS and $sATS$), and we utilise T_1 and T_0 as input lines, and R and T as output lines, the behaviour of the construction is as follows. Sending an input signal on T_1 will result in a single output signal from T, without changing the held value. Sending an input signal on T_0 will eventually result in one output signal from R and T each, with the held value set to 1. Hence this construction can also be seen to implement the Set Notation module $sATS^{-1}$. The sequential machine $sATS^{-1}$ (found by removing the brackets around the input sets in the CCS-like definition according to Observation 3.15 in Chapter 3) is also implemented by this construction.

Based on the the construction in Figure 9.25, we can conclude two universality results.

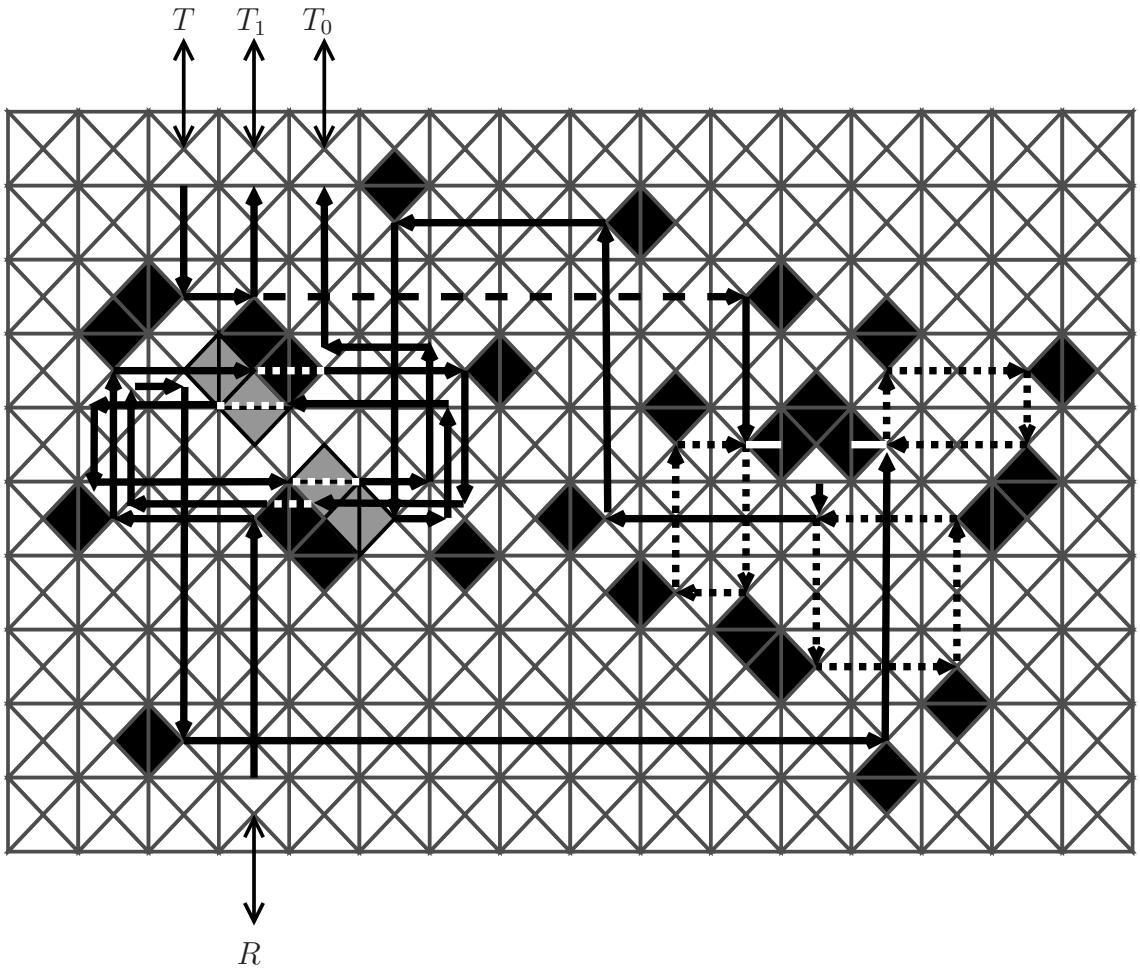


Figure 9.10: Construction which implements ATS, $sATS$ and $sATS^{-1}$ modules. The two memory structures from the left of the figure hold a value of 1 (depicted by the black subcells) or a value of 0 (depicted by the grey subcells), corresponding to the ATS states S_1 and S_0 respectively. The black and grey subcells both denote the subcell state of 1 but are used to depict the differing location of the memory structure depending on the held value. A white dotted line indicates that a signal toggles the memory structure and then continues in the same direction. When the held value is set to 1, the construction also acts as an implementation of the Set Notation modules $sATS$ and $sATS^{-1}$, and the sequential machine $sATS^{-1}$.

Theorem 9.26. Any (ND) sequential machine module can be implemented by a configuration in *NANBP* or *NAP*.

Proof. By Corollary 6.22 in Chapter 6, {*ATS, Fork, Join, sATS*$^{-1}$} is universal (in the sense of Definition 2.15 in Chapter 2) for the class of (ND) sequential machines. By Theorems 9.22 and 9.24, *NANBP* and *NAP* are able to implement the Set Notation modules *Fork*, *Join*, *RT* and *IRT*. The behaviours of the sequential machine versions of *Fork*, *RT*, and *IRT* are trivially equivalent (Observation 3.15 in Chapter 3) to their Set Notation counterparts. As discussed in Example 3.14 in Chapter 3, the behaviours of the Set Notation module *Join* and the sequential machine module *Join* are equivalent as long as safety and non-clashing hold within the network. Hence, *NANBP* and *NAP* are able to implement the sequential machine modules *Join*, *Fork*, *RT* and *IRT*. Figure 9.10 shows how to implement *ATS* and the sequential machine *sATS*$^{-1}$, also with *NANBP* or *NAP*. □

Theorem 9.27. Any Set Notation module can be implemented by a configuration in *NANBP* or *NAP*.

Proof. By Theorem 6.21 in Chapter 6, {*Join, Fork, RT, IRT, sATS, sATS*$^{-1}$} is universal for the class of Set Notation modules. By Theorems 9.22 and 9.24, *NANBP* and *NAP* are able to implement the Set Notation modules *Fork*, *Join*, *RT* and *IRT*. Figure 9.10 shows how to implement *sATS* and *sATS*$^{-1}$, also with *NANBP* or *NAP*. □

This shows an interesting result, that *NANBP* is as expressive as *NAP* and is capable of implementing all (ND) sequential machines and all Set Notation modules. This is despite *NANBP* not containing the set of rules *M*. As discussed at the end of Chapter 3, bijective modules combined with multiple signals can result in networks which implement useful irreversible modules (such as *Merge*), and are universal for all modules. It would appear from Theorems 9.26 and 9.27 that a corresponding property exists for STCA, which is that local reversibility of all STCA rules, when combined with parallelism (in the form of multiple "signals") can result in useful irreversible computation.

9.3 Conclusion

In this chapter we introduced four novel STCAs for implementing DI networks, including two STCAs for reversible serial and non-arb non-b-arb networks. The two main STCAs have several very useful properties, they are locally deterministic, locally reversible and support what we call *direction-reversibility*. This allows us to operate a network in reverse by changing the direction of signals and utilising

its output lines as input lines (and vice versa). This removes the need for separate constructions to implement the inverse of a network. The new notions of global determinism and global reversibility were proven to hold for these two STCA. We also introduced two further extensions to the STCAs which simulate irreversible serial and non-arb b-arb networks. These two additional STCAs were shown to be locally deterministic and globally deterministic. Finally, we proved that the third and fourth STCAs can be used to implement any module in either Set Notation or the ND sequential machine model.

Chapter 10

Software tools

In this chapter we detail the two pieces of software developed in support of this thesis. The first, called *Delay-Insensitive Network Tool Suite* contains implementations of the maximal environment generation algorithms in Chapter 4, conversion algorithms in Chapter 5 and algorithms which follow the non-arb and eq-arb construction methods in Chapter 6. It also implements a version of the DI-Set algebra in Chapter 7 with interactive execution and LTS generation with property and bisimulation/simulation checking. The second piece of software, called *STCA Simulator*, implements the four direction-reversible STCA in Chapter 9 with an interactive graphical interface. It also contains the constructions from that chapter. A brief description of how important features are implemented is given for each piece of software.

The two programs were developed in the Java programming language [14], and run on the Java SE platform [1]. They require the installation of the Java Runtime Environment in order to launch, which can be found at [1]. No other external libraries are utilised and all code other than built-in Java SE libraries is our own. Both pieces of software can be found with their respective source codes at [51].

10.1 Delay-Insensitive Network Tool Suite

In this section, we describe the software which was developed to aid in the research and study of abstract DI networks.

The graphical front-end of the program is divided into four main distinct tabs, each reflecting the different research areas in this thesis. They are the Conversion tab, Construction tab, Environment Generation tab, and the DI-Set Algebra tab, which reflect the material in Chapters 5, 6, 4 and 7 respectively. The CCS-like definitions of modules is assumed when entering definitions into the software.

Usage instructions can be found in the included README file which accompanies the program. The software source code is also bundled [51] and extensively

commented, so the interested reader may explore the implementation details for the following features in more depth. Additionally, the various module definitions given in this thesis, and network definitions given in Chapter 7 are included as examples in the program.

10.1.1 Overview of features

We now list the main features of the program.

1. Each tab contains the ability to save/load various definitions which are relevant to the tab's functionality. Furthermore, a console window displays extra information to the user which is not displayed in the main window (Figure 10.1). This includes additional properties of modules, and the actions taken by algorithms as they are executing.

2. The Conversion tab (Figure 10.2) contains the ability to convert (ND) sequential machine module definitions to Set Notation module definitions and vice versa, according to the algorithms in Chapter 5.

3. The Construction tab (Figure 10.3) allows the user to input any non-arb or eq-arb module definition. The software then generates a construction according to the methods in Chapter 6. The construction contains the irreversible version of Stage 2 (Figure 6.7 in Chapter 6) only if the module is b-arb. The construction is given as a list of modules and wires. It also gives the DI-Set algebra's network definition of the construction, including all required constant definitions and the wire function.

4. The Environment Generation tab (Figure 10.4) allows the user to input any Set Notation module. It will then calculate which properties outlined in Chapter 4 hold (i.e. if it is 1-step consistent, auto-firing, auto-clashing or stable). The user may then generate a maximal environment of the module using one of the two algorithms in Chapter 4, depending on whether or not it is non-arb.

5. The DI-Set Algebra tab implements a slightly modified version of the DI-Set algebra in Chapter 7, which is more friendly to typical keyboard-based input. "$\cdot$" is replaced with ":" (with the SOS rules from Figure 7.2 assumed to be modified appropriately). We also allow a wildcard operator "$*$" when defining the hidden ports of a network, allowing multiple ports with the same label to be hidden with a single entry (e.g. "$* : 1$" where "1" is a module label). This section of the software contains the largest number of features, which we detail below. The tab is separated into multiple screens: the Main screen (Figure 10.5); the GUI Input screen (Figure 10.6); the Interactive Execution screen

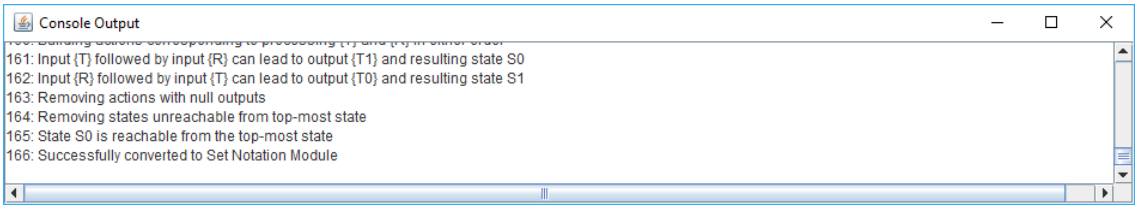


Figure 10.1: Delay-Insensitive Network Tool Suite Console Window.

(Figure 10.7) and the LTS screen (Figure 10.8). The main features of this tab are as follows.

- Maintain two separate network definitions (with associated constant definitions and wire function) given in a variation of DI-Set algebra.
- Enter network definitions manually using the keyboard (as in Figure 10.5) or with an interactive GUI-based term generator (Figure 10.6) where the user selects modules, names them, and then specifies wire connections, initial signals and port hidings.
- Hide multiple ports with the same label by utilising a wildcard "*" operator in place of literal port names.
- Execute a network one transition at a time (Figure 10.7), by viewing all available transitions at each state and selecting which one to apply.
- Generate the LTS for a network definition (Figure 10.8), including the option to automatically halt this procedure if the software detects that the current LTS being generated has an infinite number of states.
- Analyse an LTS and check if the properties of safety and non-clashing hold (Figure 10.8).
- Given a set of state pairs for two generated LTSs, validate whether it represents one network simulating the other, or if it represents a bisimulation between them (Figure 10.8).

10.1.2 Implementation details

In Figure 10.9 we show the major relationships between the packages which make up the software. Excluding the GUI, GUIListeners and CommonStructures packages, each package is one of two types: "Operations" or "Structure".

An "Operations" package contains key technical algorithms corresponding with the material in this thesis. An exception to this is the InputOutputOperations package which contains algorithms for storing definitions on the hard disk and algorithms

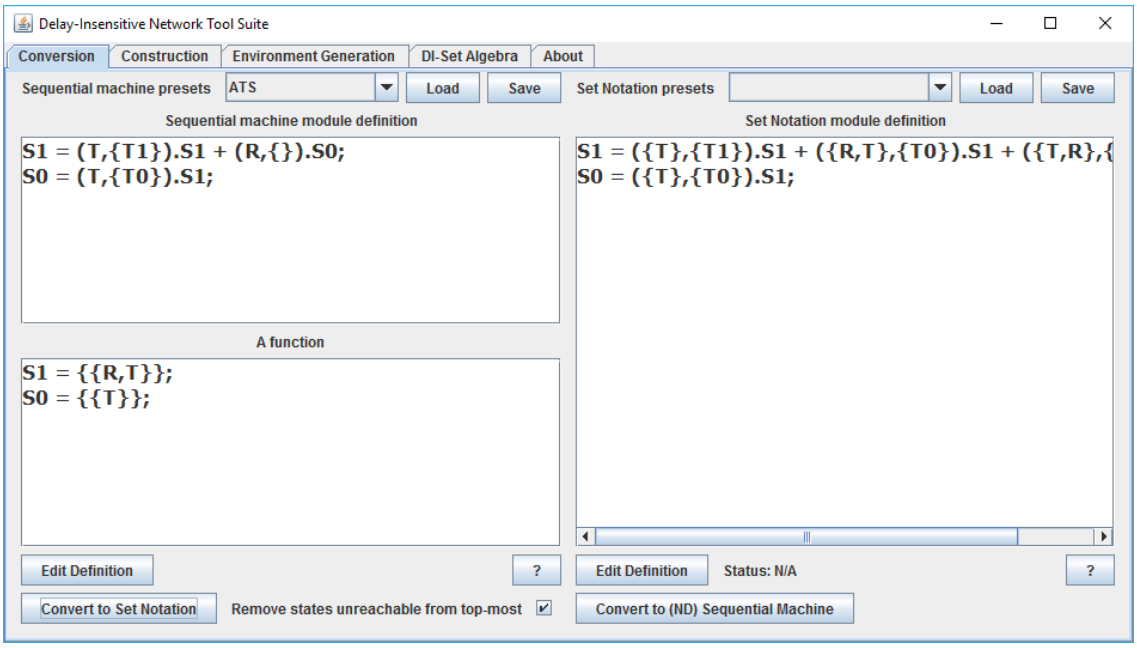


Figure 10.2: Delay-Insensitive Network Tool Suite Conversion tab.

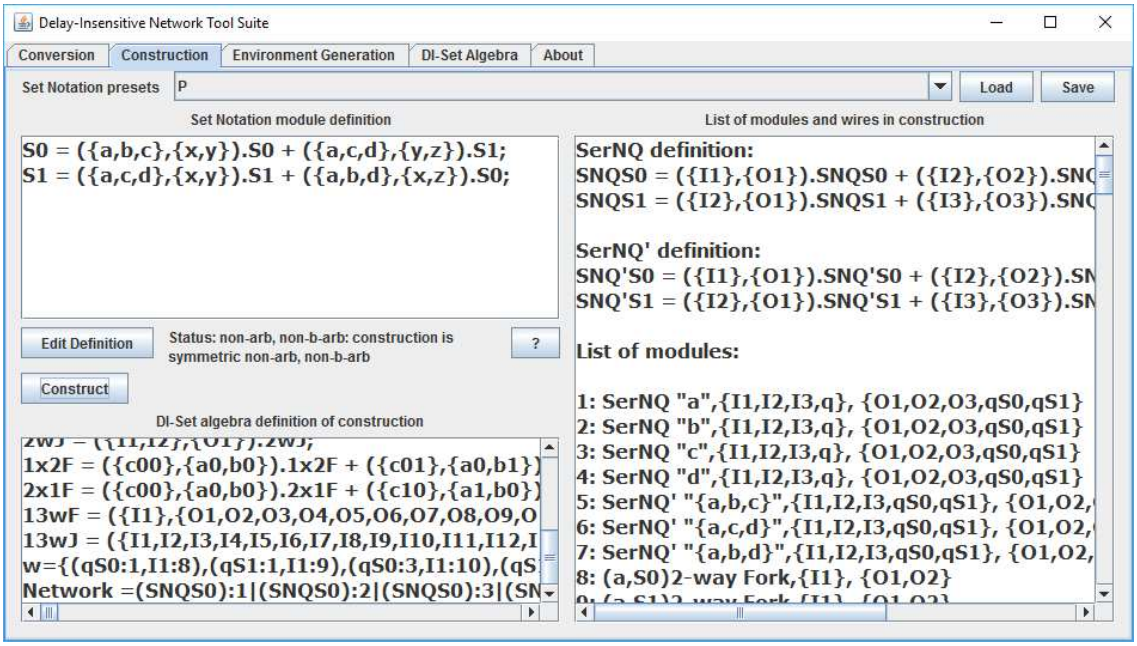


Figure 10.3: Delay-Insensitive Network Tool Suite Construction tab.

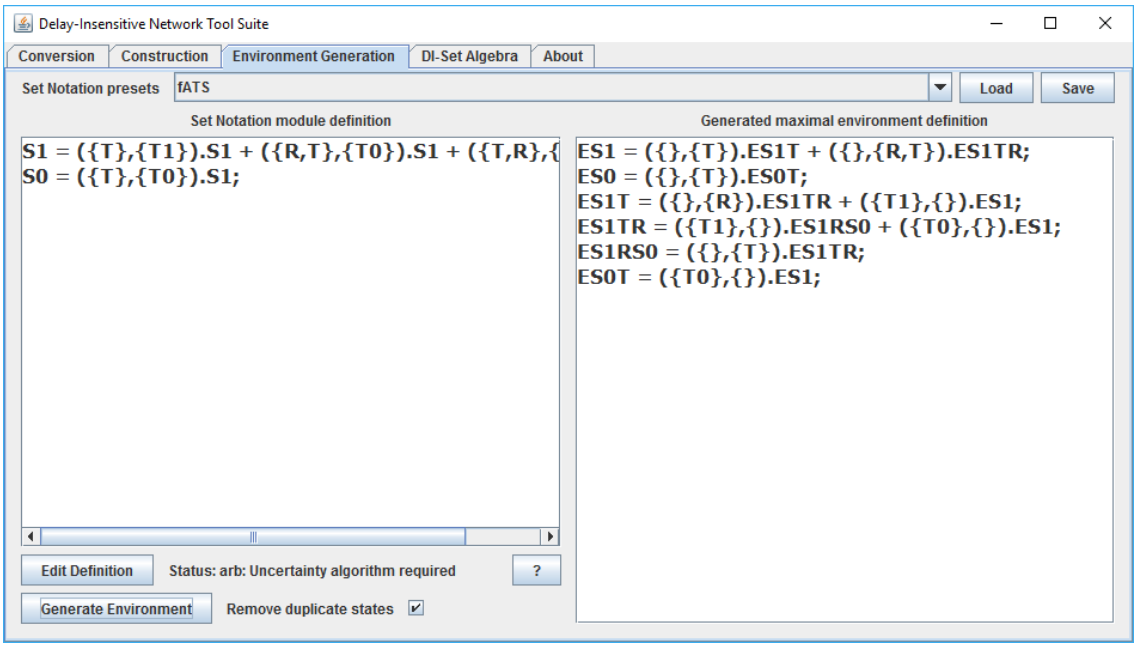


Figure 10.4: Delay-Insensitive Network Tool Suite Environment Generation tab.

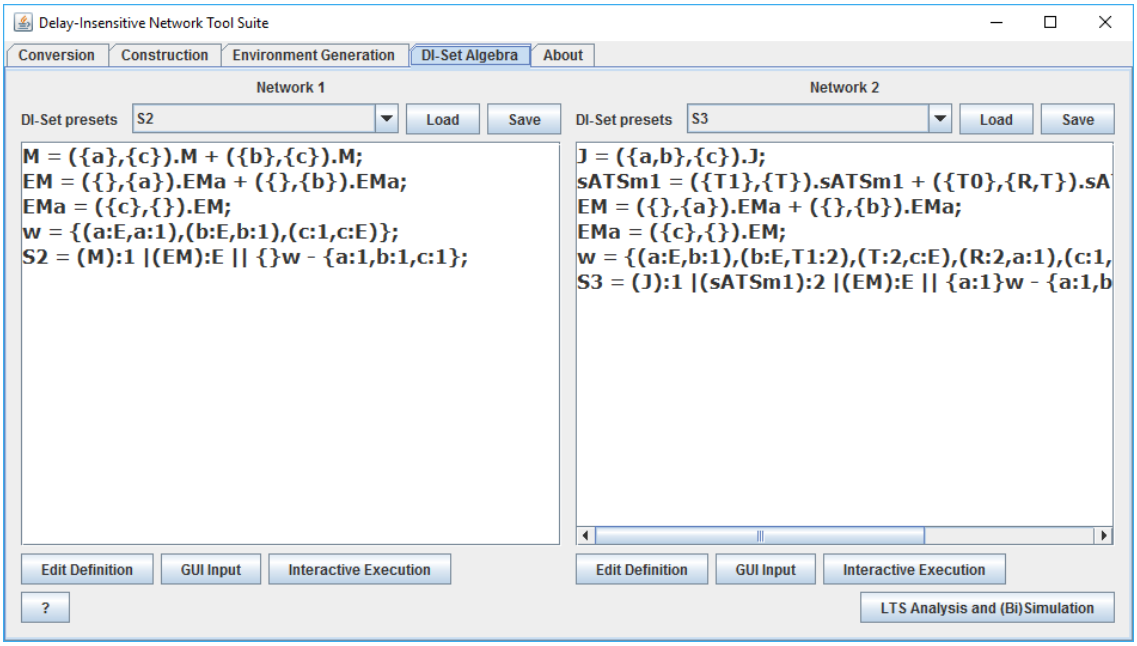


Figure 10.5: Delay-Insensitive Network Tool Suite DI-Set Algebra tab Main screen.

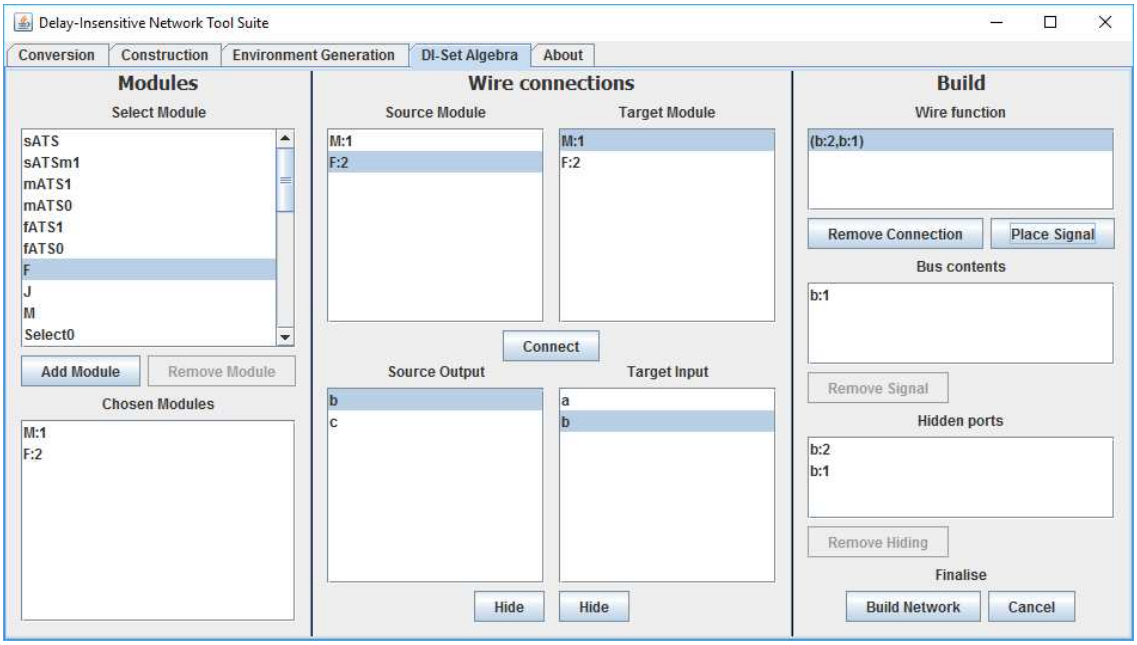


Figure 10.6: Delay-Insensitive Network Tool Suite DI-Set Algebra tab GUI Input screen.

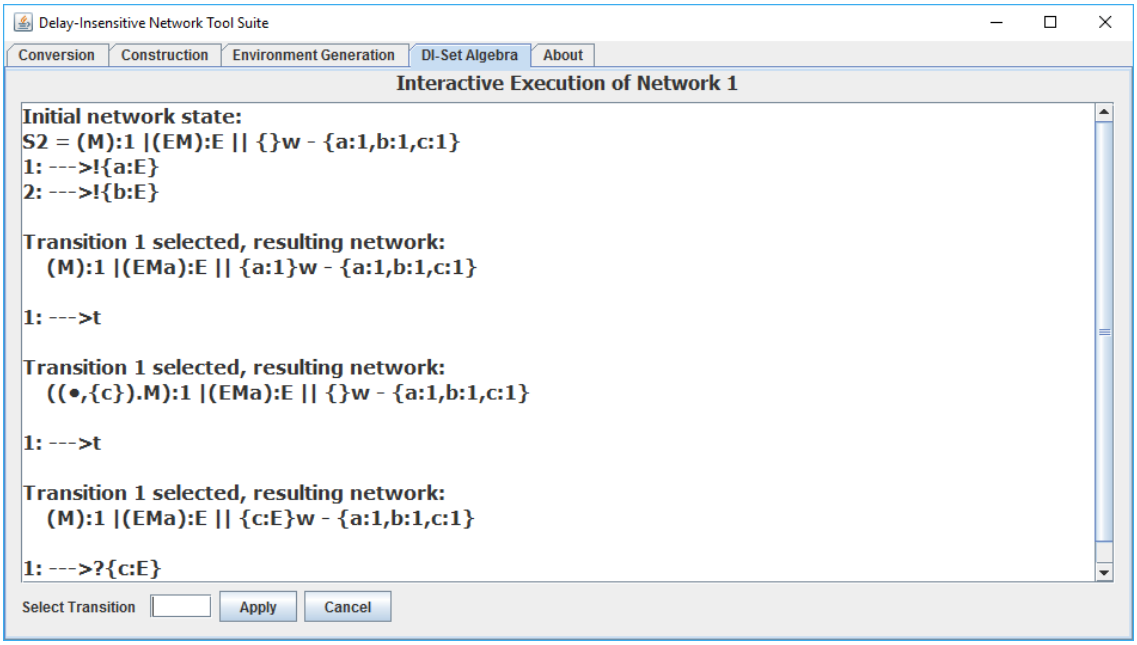


Figure 10.7: Delay-Insensitive Network Tool Suite DI-Set Algebra tab Interactive Execution screen.

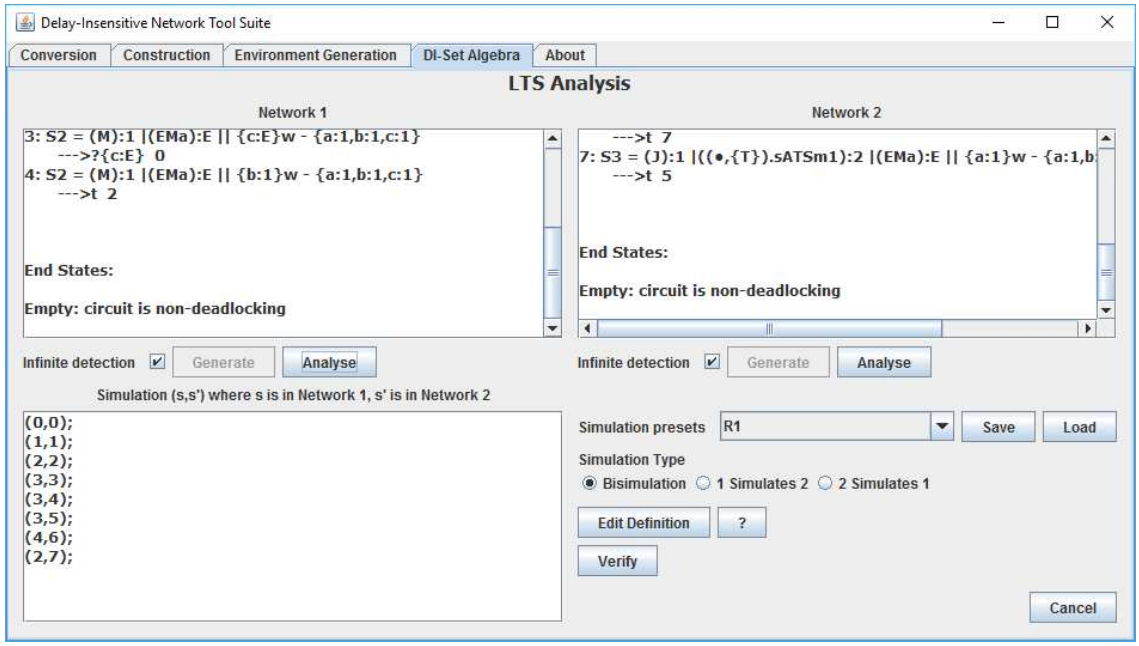


Figure 10.8: Delay-Insensitive Network Tool Suite DI-Set Algebra tab LTS screen.

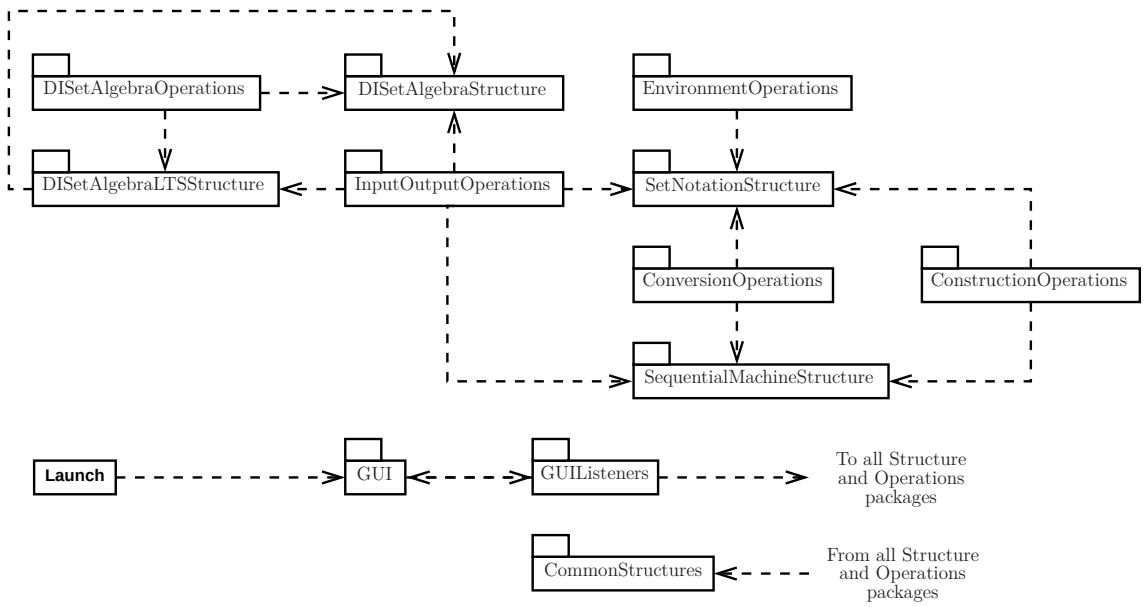


Figure 10.9: High-level architectural diagram of Delay-Insensitive Network Tool Suite, showing the major relationships between the packages which make up the program source. An arrow represents a dependency (import) between packages, where the package at the tail of the arrow is dependent on functionality within the package at the head of the arrow. We also include the Launch class in the diagram, which is not part of any package.

for parsing definitions. For example, DISetAlgebraOperations contains algorithms for inferring transitions in the DI-Set algebra, for generating the LTS of a given network, and verifying simulation/bisimulation of user-defined simulation/bisimulation relations between two LTSs. Each of these packages may also contain data structures which are unique to the functionality provided by that package, but are not utilised across the rest of the software.

A "Structure" package contains classes which are intended to model some data structure or component, along with commonly associated operations. These packages are generally utilised by multiple areas of the software. For example SetNotationStructure contains classes which are used to store all data comprising a single module in the Set Notation model. It is utilised by back-end functionality relating to the Conversion, Environment and Construction tabs (and hence by the ConversionOperations, EnvironmentOperations, and ConstructionOperations packages). It also contains associated operations such as the ability to invert a module definition, or to check properties such as whether a module is arb or b-arb.

The DISetAlgebraStructure contains classes which model the structure of DI-Set algebra. In Figure 10.10 we show the major relationships between the classes in this package. Objects heavily utilise class membership in order to reflect the hierarchical structure of the terms defined in Figure 7.1 in Chapter 7.

In general, each one of the types defined on the left-hand side in Figure 7.1 in Chapter 7 is represented internally by a Java class object: *port* and *name* are implemented directly as instances of the String class (included as part of Java SE); A is implemented using PortSet; *action* is implemented using IOAction; M is implemented using Module; nM is implemented using NamedModule; P is implemented using NamedModuleSet; $nPort$ is implemented using NamedPort; C is implemented using NamedPortSet; *wire* is implemented using WireConnection; $_w$ is implemented using WireFunction; G is implemented using Bus and S is implemented using PartiallyVisibleNetwork. The exception is the type L which does not have a corresponding Java class. Instead, an instance of PartiallyVisibleNetwork, contains membership fields for a NamedModuleSet (type P), a Bus (type G) and a NamedPortSet (type C). Hence the components of L are referenced directly in S, without implementing an intermediate term of type L. This results in simpler code and more efficient manipulation of data during run-time.

LTSs are generated by simply calculating all available transitions for a network, then for each transition, recursing and performing the same procedure for the resulting network. An LTS is represented by a linear list of objects of type LTSState from the DISetAlgebraLTSStructure package. Each LTSState object stores a unique state of the LTS. Each LTSState object also contains a list of outgoing transitions, as well as a corresponding list of LTSState objects, with each object in this list

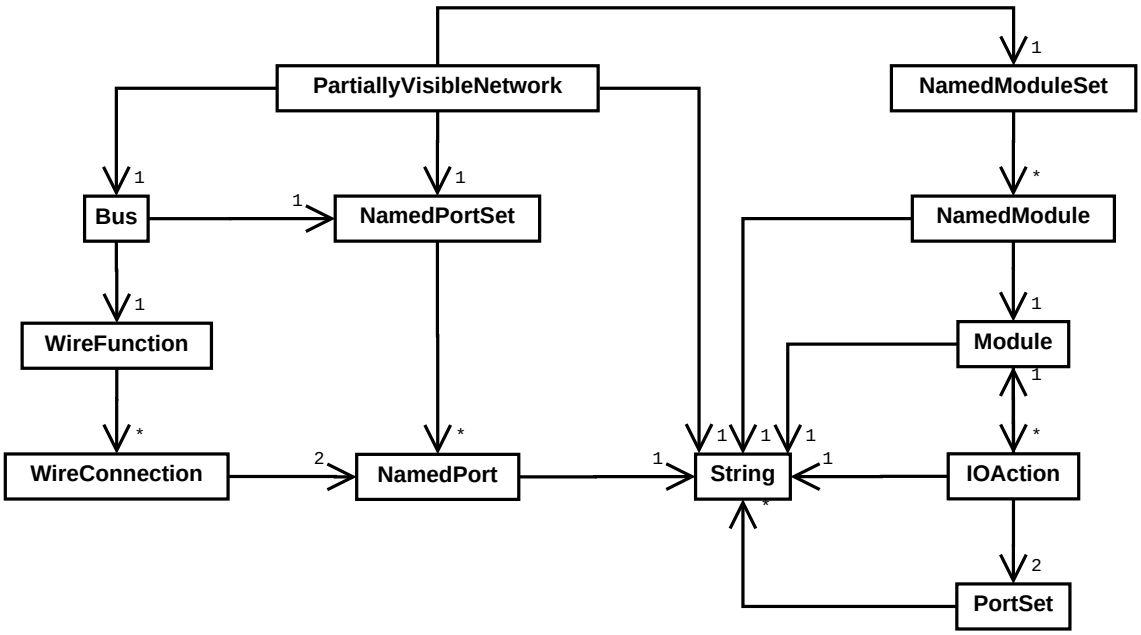


Figure 10.10: Architectural diagram of the DISetAlgebraStructure package, showing the major relationships between the Java classes. An arrow represents class membership, where a single instance of the class at the tail of the arrow contains n instances of the class at the head of the arrow as attributes, where n is the number at the head of the arrow. An asterisk (*) represents an arbitrary number greater than or equal to zero. We also include the String class in the diagram, which is included by default as part of Java SE and is not technically part of the DISetAlgebraStructure package.

being the resulting LTSState for each corresponding outgoing transition. A state of the LTS is checked against the existing overall list of LTSState objects, and a new LTSState object is added if one does not already exist for the given state of the LTS. Non-clashing and safety properties of the LTS are validated by simply iterating through all states in the LTS, and checking that these properties hold for each state as required by Definitions 7.6 and 7.7 in Chapter 7.

Infinite LTS state detection is achieved by examining each new state as it is added, to see if there already exists a state in the LTS such that:

1. the state of each NamedModule in the existing LTS state is the same as in the new LTS state,

2. the bus contents (stored with an instance of Bus) of the new LTS state is a superset of the bus contents of the existing LTS state and,

3. there exists a path of transitions from the existing LTS state to the new LTS state.

If such a state is found, then generation of the LTS halts. Hence this checks whether the LTS exhibits infinite growth behaviour (according to Proposition 7.13 in Chapter 7). The existence of a path between two states is found by recursion. A list of states visited by recursion is maintained to ensure that states are not checked more than once.

Finally, simulation and bisimulation are checked by iterating through each state pair which has been defined by the user. For each pair, it is checked that the requirements of Definition 7.15 or Definition 7.19 in Chapter 7 are satisfied. Recursion is again utilised to check all possible paths of only τ transitions from a particular LTS state. As before, a list of LTS states visited by recursion is maintained to ensure that LTS states are not checked more than once. The software requires that the components within a state pair are written in a fixed order with respect to the two LTSs, regardless of which network is said to be simulating the other by the relation. Hence the left component in a state pair always represents a state from the left LTS on-screen, similarly for the right component and the right LTS.

We briefly describe several other technical aspects of the software.

1. All parsing makes use of Java SE's built-in ability to split strings based on regular expressions. This allows state or constant definitions to be easily broken down into lists of actions, and then further into input/output sets and state names.

2. Set Notation module definitions and (ND) sequential machine definitions are implemented using data structures which are used commonly in the implemen-

tations of the Conversion, Construction and Environment tabs. These structures store lists of strings for state, input, and output names. All transitions are then stored as tuples of integers, where the values refer to the indexes of the appropriate names in their respective lists. Calculating properties of these modules (e.g. where they are non-arb, 1-step consistent, auto-clashing etc.) is done in the expected way by comparing transitions.

3. The Construction tab utilises algorithms which generate non-arb and eq-arb module constructions according to the methods in Chapter 6. These algorithms are not included in this thesis due to their size. However the code is extensively commented and these algorithms heavily utilise the console output to detail step-by-step behaviour for any construction during run-time. The interested reader may consult these sources if they wish to view an imperative implementation of the construction methods in Chapter 6.

10.2 STCA Simulator

We now describe the simulator which was developed to aid in the research and study of the direction-reversible STCA in Chapter 9. As with the previous software, usage instructions can be found in the included README file which accompanies the program. As before, the software source code is also bundled [51] and extensively commented, so the interested reader may explore the following features in more depth. The various constructions given in Chapter 9 are also included as examples.

10.2.1 Brief overview of features

We now briefly list the main features of the simulator.

1. Experiment with the behaviour resulting from the transition rules of the STCA found in [32, 36, 37], as well as all STCA introduced in this thesis (*RS*, *S*, *NANBP* and *NAP*) and inverses of *RS* and *NANBP*.

2. Modify states of any subcells in the cell grid interactively by clicking on the subcell which the user wishes to modify (Figure 10.11).

3. Save/load configurations defined in any of the implemented STCA. All constructions defined in Chapter 9 are included.

4. Perform stochastic "path verification" for any pair of configurations (Figure 10.13). This executes (randomly) a given STCA starting with a given configuration. Upon reaching some other given configuration the execution stops.

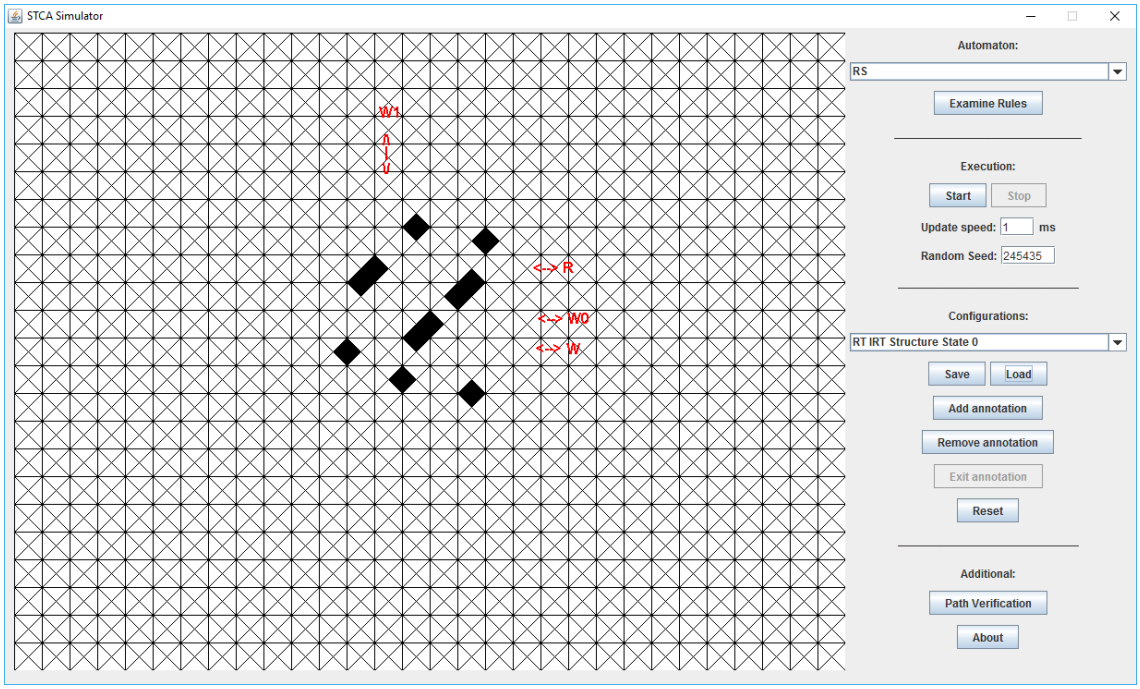


Figure 10.11: STCA Simulator main screen.

It also allows automatic repeating of this procedure, to check many times in rapid succession whether a given configuration is likely to reach some other given configuration.

5. Analyse local determinism and local reversibility of an STCA's transition rules (Figure 10.12).

6. Annotate the cell grid (as seen by the text on top of the cells in Figure 10.11). This helps the user to see which areas of a construction correspond to input or output lines of modules and networks. Annotations are stored when saving a configuration.

10.2.2 Implementation details

In Figure 10.14 we show the major relationships between the classes which make up the software. Note that the JFrame, JPanel and Thread classes are included as part of Java SE.

We briefly describe several key technical aspects of the software.

1. The state of the cell grid is represented internally using a finite 2D array of Cell objects within an instance of Cellspace. A Cell object is a simple data structure that contains four integer values, each one representing the state of a subcell. All STCA implemented contain only two states per subcell, which are

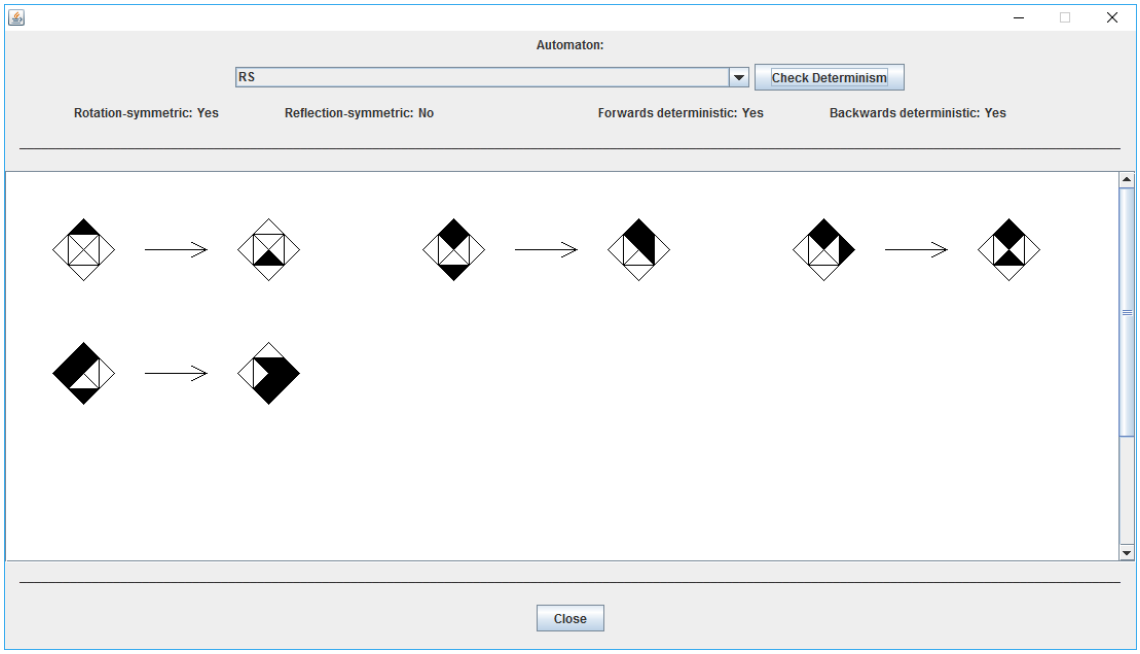


Figure 10.12: STCA Simulator Rule Examination screen.

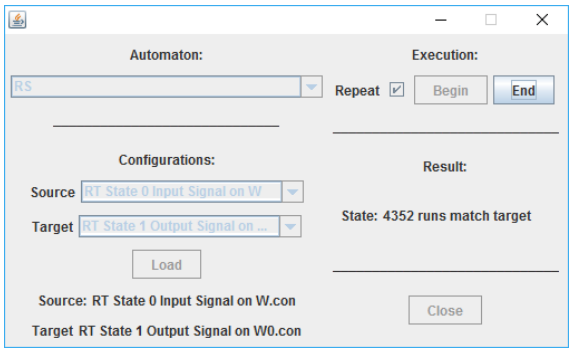


Figure 10.13: STCA Simulator Path Verification screen.

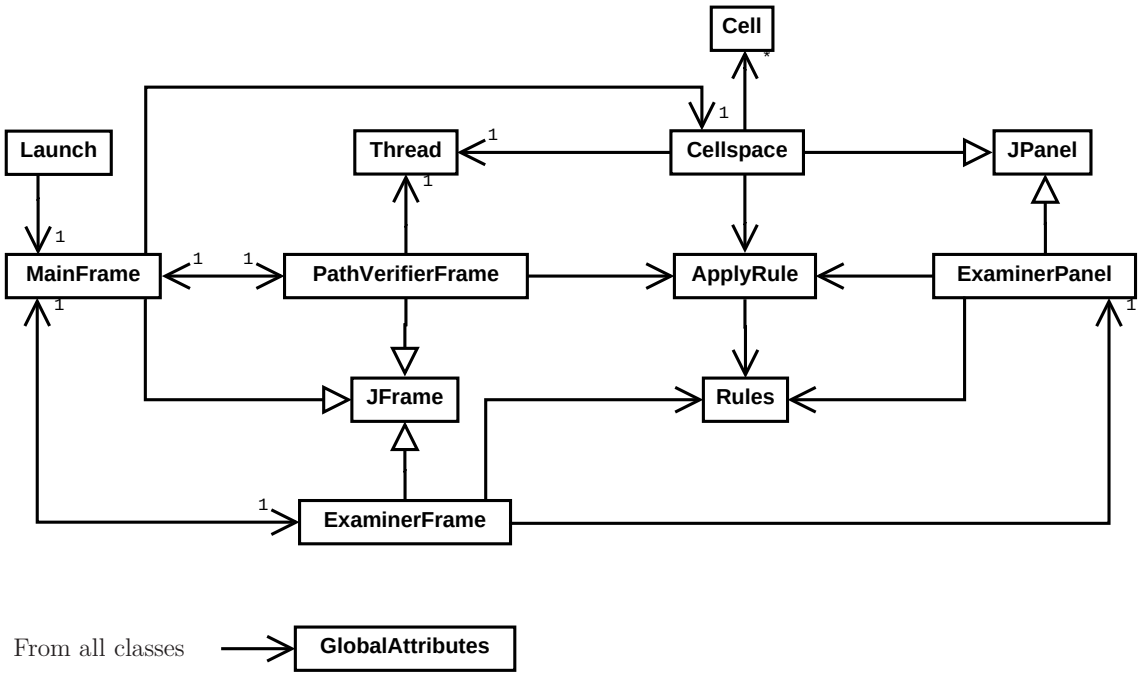


Figure 10.14: Architectural diagram of STCA Simulator, showing the major relationships between the Java classes. An arrow with a solid white triangle as its head represents inheritance, where the class at the tail of the arrow is a subclass of the class at the head of the arrow. An arrow with 1 (*) by its head represents class membership, where a single instance of the class at the tail of the arrow contains a single (multiple) instance of the class at the head of the arrow as a field. The remaining arrows represent basic dependency of non-instantiated classes, where a single instance of the class at the tail of the arrow calls upon static fields or methods within the class at the head of the arrow. Note also that the ApplyRule, Rules and GlobalAttributes classes contain only static methods and fields, and are never instantiated.

represented with the integer values of 0 and 1. These are depicted graphically as white (for 0, the default or "quiescent" state) and black (for 1, which is used in practice to model signals and structures).

2. Execution is handled by randomly selecting a Cell object, then passing the values of its four subcells and four adjacent neighbouring subcells to a method in ApplyRule which checks the states of subcells against hard-coded rules stored in Rules. Rotations and reflections of the states of the subcells are also considered on-the-fly. This removes the need to hard-code rotations and reflections of STCA rules. If a match is found then the new states of the subcells are applied to the given Cell object. This happens repeatedly until execution is stopped by the user.

3. As implied above, the execution logic occurs on a single thread, stored in Cellspace. Cell objects are selected one at a time. A new Cell is not considered until the current Cell is checked against the current STCA's transition rules, and the states have been updated if applicable. This however does not affect correctness of the implementation, as the underlying theoretical model does not allow adjacent cells to update at the same time (see Chapter 8). As a result, the overall behaviour of an STCA is not affected by forcing cells to update in a serial manner.

4. The hard-coded rules are stored as 1D arrays of integers in Rules, with each array representing a set of rules. Every set of 16 integers in the array represents a single rule. The first 8 integers in a set represent the states of the 8 subcells of the source of the rule, and the last 8 integers in a set represent the states of the 8 subcells of the target of the rule.

5. Local determinism and local reversibility of rules is determined by comparing the values of the subcells in the expected way. Rotations and reflections are also considered if appropriate, and these are applied on-the-fly similarly to when executing the STCA.

6. The Java2D API (part of Java SE) is used for rendering the cell grid, and this is handled by the Cellspace object. Each cell is rendered as a square containing 4 triangles representing the 4 subcells. When depicting the STCA from [37], for simplicity, subcells are also depicted using triangles rather than with circles as in [37].

10.3 Conclusion

In this chapter we detailed the two pieces of software developed in support of this thesis. The first, called *Delay-Insensitive Network Tool Suite* contains implementations of the maximal environment generation algorithms in Chapter 4, conversion algorithms in Chapter 5 and algorithms which follow the non-arb and eq-arb construction methods in Chapter 6. It also implements a version of the DI-Set algebra in Chapter 7 with interactive execution and LTS generation with property and bisimulation/simulation checking. The second piece of software, called *STCA Simulator*, implements the four direction-reversible STCA in Chapter 9 with an interactive graphical interface. It also contains the constructions from that chapter. A brief description of how important features are implemented was given for each piece of software.

Chapter 11

Conclusion and future research

In this chapter we summarise the work achieved in this thesis and outline possible future directions of research.

11.1 Achievement of objectives

We set out to define a new, more robust model for abstract delay-insensitive systems, which exhibits truly concurrent behaviour at the fundamental level. This model would allow us to define reversibility of concurrent modules, as well as a clear distinction between deterministic and non-deterministic behaviour. We wished to also formalise the notion of an environment for a module or network. This would allow us to study and define clear notions of universality and implementation. We have achieved this objective, in the form of the Set Notation model for delay-insensitive modules and networks. The main description of the Set Notation model is found in Chapter 3, and notions of environments, implementation and universality in Set Notation are defined in Chapter 4. Universality results for the Set Notation model are found in Chapter 6.

We set out to introduce a new process algebra for the modelling and formal representation of systems defined using such a newly developed model, and further formalise behaviour and important properties (such as implementation) in the context of this algebra. Such systems would then be able to have their behaviours inferred based on the rules of this algebra. Formalising properties and behaviour in this algebra would also help prevent the ambiguity of properties and conditions which is commonly found in existing DI literature. The DI-Set algebra in Chapter 7 further formalises the behaviour and conditions of the Set Notation model and its main concepts such as implementation, safety and non-clashing.

We also wanted to investigate how to implement concurrent delay-insensitive networks in cellular automata, and see if any newly developed notions of reversibility could be exploited in order to minimise the complexity of a cellular automaton's

transition rules, or constructions in such cellular automata. The four Self-Timed Cellular Automata given in Chapter 9 exploit the inherent ability for networks of non-arb non-b-arb modules in Set Notation to be inverted, in order to construct networks which operate in both the forwards and backwards direction. The first of these STCA was argued to be an improvement over existing STCA intended for modelling DI networks, in terms of the number of transition rules which are defined.

The above concepts were intended to be further explored and studied via the development of dedicated software tools. The two tools successfully developed in support of this work are Delay-Insensitive Network Tool Suite and STCA Simulator. Both of these were described in detail in Chapter 10.

11.2 Summary of results by chapter

We now list the main results in this thesis, collected in order of chapter.

- We introduced our own reversible memory modules in the sequential machine model, and we used these modules to infer some simple universality results.

- We introduced a new model for describing the behaviour of delay-insensitive modules, called *Set Notation* which more naturally models concurrency and notions of reversibility. We defined important classes of modules in the Set Notation model, such as the non-arb, eq-arb and arb classes. We defined networks of modules in the Set Notation model, along with the execution behaviour of such networks. We defined properties of such networks, such as the non-clashing and safety properties. We investigated several further properties of modules in the Set Notation model which limit the behaviour of the environment in unexpected ways. Examples included the auto-firing or 1-step consistency properties of modules.

- We formalised the notion of an environment for a module in the Set Notation model. We gave an algorithm for calculating what is referred to as a *maximal environment* of any non-arb module. An algorithm for calculating a maximal environment of any module was then given. Finally, we used this notion to define implementation of a module using a network of modules in Set Notation.

- We introduced an extension to the sequential machine model called the *ND sequential machine* model. We established limited correspondences between three models; the sequential machine model, the ND sequential machine model, and the new Set Notation model. We proved universality results for the ND sequential machine model. We gave algorithms for converting modules which satisfy certain conditions (formalised as E1 and E2) in the sequential machine

model or the ND sequential machine model, to corresponding definitions in the Set Notation model, and vice versa.

- We investigated inversion of networks of modules. We gave some universality results for serial modules in the Set Notation model. We then gave a series of detailed construction methods and universal sets for multiple classes of modules, including the non-arb and eq-arb classes. We also proved a universal set for all Set Notation modules, by utilising a correspondence between the ND sequential machine and Set Notation models defined in the previous chapter. We demonstrated an interesting property inherent to networks of concurrent DI modules, which is that bijectivity of all modules' transition maps can still result in useful irreversible behaviour at the global level. We compared this with a similar but less general result in the literature.

- We introduced a new process algebra, known as *DI-Set algebra*, which is intended to model formally the behaviour of networks from the Set Notation model. We gave examples of encoding modules (such as *Merge*) and networks (such as a network that implements *Merge*) in DI-Set algebra, and defined properties of networks discussed in previous chapters (such as safety and non-clashing) more formally in the context of DI-Set algebra. We investigated the use of bisimulation and simulation, and used these together with the aforementioned properties to define more formally the concept of implementation of a module using a network of modules.

- We defined new versions of *global reversibility* and *global determinism* in the context of Self-Timed Cellular Automata. These are related to similar properties in the literature, but are more flexible properties which are based on notions of convergence, and are appropriate for STCAs which simulate concurrent DI networks.

- We introduced four novel STCAs for implementing DI networks, including two STCAs for reversible serial and non-arb non-b-arb networks. The two main STCAs have several very useful properties, they are locally deterministic, locally reversible and support what we call direction-reversibility. This allows us to operate a network in reverse by changing the direction of signals and utilising its output lines as input lines (and vice versa). This removes the need for separate constructions to implement the inverse of a network. The new notions of global determinism and global reversibility were proven to hold for these two STCAs. We also introduced two further extensions to the STCAs which simulate irreversible serial and non-arb b-arb networks. These two additional STCAs were shown to be locally deterministic and globally

deterministic. Finally, we proved that the third and fourth STCAs can be used to implement any module in either Set Notation or the ND sequential machine model.

- We detailed the two pieces of software developed in support of this thesis. The first, called *Delay-Insensitive Network Tool Suite* contains implementations of the maximal environment generation algorithms in Chapter 4, conversion algorithms in Chapter 5 and algorithms which follow the non-arb and eq-arb construction methods in Chapter 6. It also implements a version of the DI-Set algebra in Chapter 7 with interactive execution and LTS generation with property and bisimulation/simulation checking. The second piece of software, called *STCA Simulator*, implements the four direction-reversible STCA in Chapter 9 with an interactive graphical interface. It also contains the constructions from that chapter. A brief description of how important features are implemented was given for both pieces of software.

11.3 Future work

We finish by briefly outlining some areas of research from this thesis which could be further developed.

- Currently, the only method for generating a construction of some arb module (which is not eq-arb) is to utilise the correspondence between Set Notation and (ND) sequential machines, and then follow the construction method for an arbitrary (ND) sequential machine (Section 5.1.1 in Chapter 5). As discussed in Chapter 6, this results in a "sequential" style construction, which lacks any useful level of concurrency. Research into such "parallel" constructions was conducted, but no meaningful methods have yet to be discovered. Given the existence of "parallel" constructions for non-arb and eq-arb modules, a method for constructing arb modules in such a way would allow the full set of Set Notation modules to be constructed in a way that utilises concurrency.

- Verification of the constructions in Chapter 9 was performed both by hand and by testing several thousand execution sequences using the STCA Simulator. However a more formal means of verification has yet to be developed. Full enumeration of all possible resulting configurations of a STCA, starting from some given configuration, is an obvious future direction. However we note that, given an STCA which allows 2 states per subcell and a finite grid of m cells, there are 16 possible states per cell, and hence up to 16^m possible configurations in total for the given STCA and grid. Therefore any algorithm

which enumerates all possible configurations may have a very large upper-bound time-complexity.

- Verifying whether a network satisfies the notion of implementation in DI-Set algebra (Definition 7.24 in Chapter 7) is dependent on enumeration of the LTS of the given network. However highly parallel networks contain very large LTSs due to a state explosion arising from concurrent signals. This has proven impractical to calculate in many cases. We attempted to calculate the LTS of the network resulting from combining Figures 6.5 and 6.6 with a maximal environment of P (see Chapter 6), using the Delay-Insensitive Network Tool Suite described in Chapter 10. This resulted in crashing the system due to excessive RAM usage. Attempting to manually define a simulation relation for such a large LTS would also be impractical. Hence a more feasible notion of implementation which does not require the full enumeration of an LTS is ideal.

- The algorithms in this thesis have not had their time-complexity studied in great depth. This is a difficult task and beyond the scope of this work. However there are some suggestions of a high-cost for many of the algorithms.

    1. Figure 4.4: The algorithm duplicates the input/output sets and states of the module for the environment. For each action of the original module, it creates two actions and one new state for the environment. We can deduce that this algorithm has linear time complexity.

    2. Figure 4.5: The algorithm creates an environment state for each possible uncertainty which may be reached during execution of the network containing the module and its environment. An uncertainty contains any possible set of configurations. Hence, in general there are 2^n possible uncertainties, where n is the number of possible configurations. The number of possible configurations is dependent on all possible combinations of input lines, output lines, and states of a module. Hence there are up to 2^m possible configurations, where $m = |I| + |O| + |Q|$ for some given $N = (Q, I, O, T)$. This implies that there are up to 2^{2^m} possible uncertainties for a given module. Furthermore, the creation of each environment state involves a non-fixed number of steps, as all possible prefixes of variable length autopaths must be examined. However it is not clear whether there exists some module such that all possible uncertainties must be calculated for a corresponding environment.

    3. Figure 5.4: for all q and for all $S \in A(q)$ and for all $A \subseteq S$, the algorithm produces at least $n!$ actions (one for each permutation of the set A),

where $n = |A|$. The algorithm produces even more actions if the original module is an ND sequential machine. This suggests that the algorithm's upper-bound time complexity is at least factorial time.

4. Figure 5.5: for all actions $(A, B).q'$ in the Set Notation module, the algorithm produces up to $n!$ actions in the resulting (ND) sequential machine (a sequence of actions for each permutation of A), where $n = |A|$. This suggests that the algorithm's upper-bound complexity is at least factorial time. Furthermore, it also tracks which sequences are "available" to implement a particular action, and therefore contains additional overhead regarding the ruling out of sequences.

- Modelling the behaviour of DI networks defined in the Set Notation model using an existing formalism is a possible future direction. The advantage of this is that it would allow the utilisation of well-developed verification techniques and software tools. It may occur to the reader that the behaviour of individual modules in the Set Notation model (the "firing" of a transition once certain prerequisites are satisfied) seems a good match for *Petri Nets* [73]. A deliberate decision was made to instead model networks using the domain-specific DI-Set algebra given in this thesis. This had the advantage of allowing CCS-like definitions of modules to be encoded directly, and allows a human to quickly analyse the state of the network and bus by simple visual inspection of the syntax itself. Utilising Petri Nets would require the conversion of modules from their more intuitive CCS-like form to a Petri Net that exhibits the same behaviour, as well as the converse. The current state of the module would need to be explicitly tracked, due to the static structure of Petri Nets. Furthermore, while the property of non-clashing appears easy to verify in the context of Petri Nets, the property of safety does not appear to be as simple due to this static nature. Checking that the currently signalled input lines are a subset of those required to fire some transition, such that the module is also in the correct state to fire the transition, is a more complex procedure. Compare this with the DI-Set algebra, where we simply check that the set of signals with a given label is a subset of the input set of some action, currently visible in the syntax, for the module with the same label. Therefore the current state of each module does not need to be explicitly tracked or factored into the verification of safety. However the abundance of research into Petri Nets as well as wide availability of software tools means that Petri Nets should not be ruled out as a way of modelling these types of networks. We note [43] which investigates modelling "DI" circuits (where they are referred to as speed-independent) of logic gates (Section 2.4.1 in Chapter 2) using Petri Nets. Similarly, [77] shows how to

model the behaviour of multiple classes of asynchronous logic circuits using a formal framework based on Petri Nets. We also note [57, 58] which detail how to realise Petri Nets using asynchronous circuits of logic gates.

- We also note that DI-Set algebra does not include notions of reversibility or backtracking. As a result, it is not possible to invert a network of non-arb non-b-arb modules in the same way as described in Section 6.1 in Chapter 6, and utilised in the direction-reversible STCA given in Chapter 9. This would involve sacrificing the dynamic nature of the syntax of DI-Set algebra, and the full definition of a module, including the definitions of previous, now possibly unreachable, states would need to be retained in the top-level term. The simplicity of encoding the CCS-like definition of a state of a module would be lost, and a syntactic way of "tracking" the current state would need to be implemented, further complicating the algebra. However a process algebra that allows inversion analogously to the STCA in Chapter 9 would be useful in order to allow the modelling of the bidirectional networks found in these STCA using a single network definition. We note [4, 7] which detail *RCCS* (for *reversible* CCS) process algebra. RCCS allows the backtracking of events. Similarly, we note [70] which describes a process algebra which can reverse the direction of computation. Finally, we also note [69] which shows how to convert standard dynamic process algebra operators (such as "+") to static equivalents, permitting notions of reversibility.

- Recall the Seq/Set model outlined in Chapter 5 which was used to prove correspondences between the sequential machine, ND sequential machine and Set Notation models. DI-Set algebra appears capable of also modelling the Seq/Set model. As a result, it should be possible to encode (ND) sequential machine definitions in the algebra, and also to further formalise the notion of sequential implementation by defining a corresponding notion in the algebra. This was not attempted due to space, as well as the fact that the motivation for DI-Set algebra was to facilitate the further development and formalisation of the Set Notation model. However such notions could be further explored, and existing notions of bisimulation and simulation in DI-Set algebra could be utilised in this possible future study.

Bibliography

[1] *java.com*. `http://www.java.com/en/`. Accessed: 2016-09-08.

[2] R. J. Baker. *CMOS Circuit Design, Layout, and Simulation*. 3rd edition. Wiley-IEEE Press, 2010.

[3] C. H. Bennett. Logical reversibility of computation. In *IBM Journal of Research and Development*, 17(6):525–532. IBM, 1973.

[4] L. Cardelli and C. Laneve. Reversible structures. In *Proceedings of the 9th International Conference on Computational Methods in Systems Biology (CMSB '11)*, pages 131–140. ACM, 2011.

[5] F. Cheng. Synthesizing iterative functions into delay-insensitive tree circuits. In *Proceedings of the 1997 International Conference on Computer Design: VLSI in Computers and Processors (ICCD '97)*, pages 301–306. IEEE, 1997.

[6] E. F. Codd. *Cellular Automata*. Academic Press, Inc., 1968.

[7] V. Danos and J. Krivine. Reversible Communicating Systems. In *CONCUR 2004 - Concurrency Theory*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004.

[8] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. Carnegie Mellon University, 1987.

[9] J. C. Ebergen. *Translating Programs into Delay-insensitive Circuits*. Centrum voor Wiskunde en Informatica, 1989.

[10] J. C. Ebergen. A formal approach to designing delay-insensitive circuits. In *Distributed Computing*, 5(3):107–119. Springer, 1991.

[11] N. Fatès. A guided tour of asynchronous cellular automata. In *Journal of Cellular Automata*, 9(5–6):387–416. Old City Publishing, 2014.

[12] E. F. Fredkin and T. Toffoli. Conservative logic. In *International Journal of Theoretical Physics*, 21(3):219–253. Springer, 1982.

[13] G. Gopalakrishnan and V. Akella. High-level optimizations in compiling process descriptions to asynchronous circuits. In *Journal of VLSI signal processing systems for signal, image and video technology*, 7(1):33–45. Springer, 1994.

[14] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. 1st edition. Addison-Wesley Professional, 2014.

[15] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. 1st edition. Addison-Wesley Professional, 2014.

[16] S. Hauck. Asynchronous design methodologies: an overview. In *Proceedings of the IEEE*, 83(1):69–93. IEEE, 1995.

[17] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[18] C. R. Jesshope, I. M. Nedelchev, and C. G. Huang. Compilation of process algebra expressions into delay-insensitive circuits. In *IEE Proceedings E - Computers and Digital Techniques*, 140(5):261–268. IEEE, 1993.

[19] M. B. Josephs and D. Furey. Delay-insensitive interface specification and synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 169–173. ACM, 2000.

[20] M. B. Josephs and S. M. Nowick. Scanning the technology: Applications of asynchronous circuits. In *Proceedings of the IEEE*, pages 223–233. IEEE, 1999.

[21] M. B. Josephs and J.T. Udding. An overview of D-I algebra. In *Proceedings of the 26th Hawaii International Conference on System Sciences (HICSS '93)*, pages 329–338. IEEE, 1993.

[22] J. Kari. Reversible Cellular Automata. In *Developments in Language Theory*, volume 3572 of *LNCS*, pages 57–68. Springer, 2005.

[23] R. M. Keller. Towards a theory of universal speed-independent modules. In *IEEE Transactions on Computers*, 23(1):21–33. IEEE, 1974.

[24] R. M. Keller. A fundamental theorem of asynchronous parallel computation. In *Parallel Processing: Proceedings of the Sagamore Computer Conference*, pages 102–112. Springer, 1975

[25] M. Z. Kwiatkowska. Defining process fairness for non-interleaving concurrency. In *Foundations of Software Technology and Theoretical Computer Science*, volume 472 of *LNCS*, pages 286–300. Springer, 1990.

[26] R. Landauer. Irreversibility and Heat Generation in the Computing Process. In *IBM Journal of Research and Development*, 5(3):183–191. IBM, 1961.

[27] J. Lee, S. Adachi, and F. Peper. A partitioned cellular automaton approach for efficient implementation of asynchronous circuits. In *The Computer Journal*, 54(7):1211–1220. Oxford University Press, 2011.

[28] J. Lee, S. Adachi, F. Peper, and S. Mashiko. Delay-insensitive computation in asynchronous cellular automata. In *Journal of Computer and System Sciences*, 70(2):201–220. Elsevier, 2005.

[29] J. Lee, S. Adachi, F. Peper, and K. Morita. Embedding universal delay-insensitive circuits in asynchronous cellular spaces. In *Fundamenta Informaticae*, 58(3-4):295–320. IOS Press, 2003.

[30] J. Lee, S. Adachi, Y. Xia, and Q. Zhu. Emergence of universal global behavior from reversible local transitions in asynchronous systems. In *Information Sciences*, 282:38–56. Elsevier, 2014.

[31] J. Lee, X. Huang, and Q. Zhu. Decomposing Fredkin Gate into simple reversible elements with memory. In *International Journal of Digital Content Technology and its Applications*, 4(5):153–158. Advanced Institute of Convergence Information Technology, 2010.

[32] J. Lee, X. Huang, and Q. Zhu. Embedding simple reversed-twin elements into self-timed reversible cellular automata. In *Journal of Convergence Information Technology*, 6(1):49–54. Advanced Institute of Convergence Information Technology, 2011.

[33] J. Lee, F. Peper, S. Adachi, and S. Mashiko. On Reversible Computation in Asynchronous Systems, In *Quantum Information and Complexity*, pages 296–320. World Scientific, 2004.

[34] J. Lee, F. Peper, S. Adachi, and S. Mashiko. Universal delay-insensitive systems with buffering lines. In *IEEE Transactions on Circuits and Systems I: Regular Papers*, 52(4):742–754. IEEE, 2005.

[35] J. Lee, F. Peper, S. Adachi, and K. Morita. Universal delay-insensitive circuits with bidirectional and buffering lines. In *IEEE Transactions on Computers*, 53(8):1034–1046. IEEE, 2004.

[36] J. Lee, F. Peper, S. Adachi, and K. Morita. An asynchronous cellular automaton implementing 2-state 2-input 2-output reversed-twin reversible elements. In *Cellular Automata*, volume 5191 of *LNCS*, pages 67–76. Springer, 2008.

[37] J. Lee, F. Peper, S. Adachi, K. Morita, and S. Mashiko. Reversible computation in asynchronous cellular automata. In *Unconventional Models of Computation*, volume 2509 of *LNCS*, pages 220–229. Springer, 2002.

[38] R. Manohar and A. J. Martin. *Quasi-delay-insensitive circuits are turing-complete.* Technical report. California Institute of Technology, 1995.

[39] A. J. Martin. *Asynchronous logic: Results and prospects.* `http://www.async.caltech.edu/general07.ppt`. California Institute of Technology, Accessed 2016-09-08.

[40] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the 6th MIT Conference on Advanced Research in VLSI (AUSCRIPT '90)*, pages 263–278. MIT Press, 1990.

[41] D. D. McCracken and E. D. Reilly. Backus-naur form (BNF). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley & Sons, 2003

[42] R. Milner. *A Calculus of Communicating Systems.* Springer, 1980.

[43] D. Misunas. Petri nets and speed independent design. In *Communications of the ACM*, 16(8):474–481. ACM, 1973.

[44] K. Morita. Universality of a reversible two-counter machine. In *Theoretical Computer Science*, 168(2):303–320. Elsevier, 1996.

[45] K. Morita. A simple universal logic element and cellular automata for reversible computing. In *Machines, Computations, and Universality*, volume 2055 of *LNCS*, pages 102–113. Springer, 2001.

[46] K. Morita. Reversible computing and cellular automata - a survey. In *Theoretical Computer Science*, 395(1):101–131. Elsevier, 2008.

[47] K. Morita. Reversible computing systems, logic circuits, and cellular automata. In *2012 3rd International Conference on Networking and Computing (ICNC '12)*, pages 1–8. IEEE, 2012.

[48] K. Morita. Reversible logic elements with memory and their universality. In *Proceedings Machines, Computations and Universality 2013 (MCU 2013)*, pages 3–14. Electronic Proceedings in Theoretical Computer Science, 2013.

[49] K. Morita, T. Ogiro, K. Tanaka, and H. Kato. Classification and universality of reversible logic elements with one-bit memory. In *Machines, Computations, and Universality*, volume 3354 of *LNCS 3354*, pages 245–256. Springer, 2004.

[50] T. Ogiro, A. Kanno, K. Tanaka, H. Kato, and K. Morita Nondegenerate 2-State 3-Symbol Reversible Logic Elements Are All Universal. In *International Journal of Unconventional Computing*, 1(1):47–67. Old City Publishing, 2005.

[51] D. Morrison. *Delay-Insensitive Network Tool Suite* and *STCA Simulator*, `http://www.cs.le.ac.uk/people/dm181/PhDTools.zip`. Department of Informatics, University of Leicster. Accessed 2016-09-19.

[52] D. Morrison and I. Ulidowski. Reversible delay-insensitive distributed memory modules. In *Reversible Computation*, volume 7948 of *LNCS 7948*, pages 11–24. Springer, 2013.

[53] D. Morrison and I. Ulidowski. Arbitration and reversibility of parallel delay-insensitive modules. In *Reversible Computation*, volume 8507 of *LNCS*, pages 67–81. Springer, 2014.

[54] D. Morrison and I. Ulidowski. Direction-reversible self-timed cellular automata for delay-insensitive circuits. In *Cellular Automata*, volume 8751 of *LNCS*, pages 367–377. Springer, 2014.

[55] D. Morrison and I. Ulidowski. Direction-reversible self-timed cellular automata for delay-insensitive circuits. In *Journal of Cellular Automata*, 12(1-2):101–120. Old City Publishing, 2016.

[56] T. Ogiro, A. Alhazov, T. Tanizawa, and K. Morita. Universality of 2-State 3-Symbol Reversible Logic Elements — A Direct Simulation Method of a Rotary Element. In *Natural Computing*, volume 2 of *Proceedings in Information and Communications Technology*, pages 252–259. Springer, 2010.

[57] S. S. Patil *Coordination of asynchronous events.* ScD Thesis, Department of Electrical Engineering. MIT, Project MAC, 1970.

[58] S. S. Patil. *Circuit Implementation of Petri Nets, Computation Structures Group Memo 73.* MIT, Project MAC, 1972.

[59] P. Patra and D. S. Fussell. *Building-blocks for designing DI circuits.* Technical report. Department of Computer Sciences, University of Texas at Austin, 1993.

[60] P. Patra and D. S. Fussell. Efficient building blocks for delay insensitive circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async '94)*, pages 196–205. IEEE, 1994.

[61] P. Patra and D. S. Fussell. Fully asynchronous, robust, high-throughput arithmetic structures. In *Proceedings of the 8th International Conference on VLSI Design*, pages 141–145. IEEE, 1995.

[62] P. Patra and D. S. Fussell. Conservative delay-insensitive circuits, In *Proceedings of the 4th Workshop on Physics and Computation (PhysComp96)*, pages 248–259. New England Complex Systems Institute, 1996.

[63] P. Patra and D. S. Fussell. Efficient delay-insensitive RSFQ circuits. In *Proceedings of the 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '96)*, pages 413–418. IEEE, 1996.

[64] F. Peper. Simplifying brownian cellular automata: Two states and an average of two rules per cell. In *2012 3rd International Conference on Networking and Computing (ICNC '12)*, pages 367–370. IEEE, 2012.

[65] F. Peper, T. Isokawa, N. Kouda, and N. Matsui. Self-timed cellular automata and their computational ability. In *Future Generation Computer Systems*, 18(7):893–904. Elsevier, 2002.

[66] F. Peper, J. Lee, S. Adachi, and S. Mashiko. Laying out circuits on asynchronous cellular arrays: a step towards feasible nanocomputers? In *Nanotechnology*, 14(4):469–485. IOP Publishing, 2003.

[67] F. Peper, J. Lee, J. Carmona, J. Cortadella, and K. Morita. Brownian circuits: Fundamentals. In *Journal on Emerging Technologies in Computing Systems*, 9(1):3:1–3:24. ACM, 2013.

[68] I.C.C. Phillips and I. Ulidowski. Reversibility and models for concurrency. In *Electronic Notes in Theoretical Computer Science*, 192(1):93–108. Elsevier, 2007.

[69] I.C.C. Phillips and I. Ulidowski. Reversing algebraic process calculi. In *Journal of Algebraic and Logic Programming*, 73(1-2):70–96. Elsevier, 2007.

[70] I.C.C. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In *Reversible Computation*, volume 7581 of *LNCS*, pages 218–232. Springer, 2013.

[71] G. D. Plotkin. *A structural approach to operational semantics*. Technical report. Computer Science Department, Aarhus University, 1981.

[72] G D. Plotkin. The origins of structural operational semantics. In *Journal of Logic and Algebraic Programming*, 60-61:17–139. Elsevier, 2004.

[73] W. Reisig. *Petri Nets: An Introduction*. Springer, 1985.

[74] O. Schneider and T. Worsch. A 3-state asynchronous CA for the simulation of delay-insensitive circuits. In *Cellular Automata*, volume 7495 of *LNCS*, pages 565–574. Springer, 2012.

[75] M. Shams, J. C. Ebergen, and M. I. Elmasry. *Asynchronous Circuits*. John Wiley & Sons, 2001.

[76] J. Spars and S. Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective*. 1st edition. Springer, 2010.

[77] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. In *Formal Methods in System Design*, 9(3):139–188. Springer, 1996.

[78] A. Yakovlev, P. Vivet, and M. Renaudin. Advances in Asynchronous logic: from Principles to GALS & NoC, Recent Industry Applications, and Commercial CAD tools In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '13)*, pages 1715–1724. IEEE, 2013.