Use of synchronous concurrent algorithms in the development of safety related software.

Tacy, Adam James

USE OF SYNCHRONOUS CONCURRENT ALGORITHMS IN THE DEVELOPMENT OF SAFETY RELATED SOFTWARE

Adam James Tacy

Submitted to the University of Wales in fulfilment of the requirements for the Degree of Doctor of Philosophy.
Swansea University
2005

Abstract

This thesis investigates the use of Synchronous Concurrent Algorithms (SCAs) in the development of safety related software, where a stricter adherence to mathematical correctness is required. The original model of SCAs is extended to produce abstract and concrete dynamic SCAs (dSCAs) that allow dynamic, but predictable, SCAs to be produced whose wiring maybe different at different values of a program counter. A relaxed implementation of the Generalised Railroad Crossing Problem is used to demonstrate each of the SCA models.

SCAs were originally defined by Tucker and Thompson and were restricted to unit-delays between modules. Hobley investigated the introduction of non-unit delay SCAs and how non-unit delay SCAs may be represented as unit delay SCAs. Poole, Tucker and Thompson introduced the concept of hierarchies of Spatially Expanded Systems, of which SCAs are a form. All of these tools are used and expanded upon in this thesis to provide a mechanism enabling an SCA representation of an algorithm to be transformed into an SCA representation of a computing device that implements that algorithm, and to be able to demonstrate correctness.

As each SCA model can be represented algebraically, this thesis provides the transformations as meta-algebras, i.e. algebras that can transfrom one algebra to another algebra.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr Neal Harman for his guidance and patience during my PhD studies. Without his support it would have been difficult to focus for the length of part-time work this thesis has required.

I gratefully acknowledge the support I received from the UK Ministry of Defence during the first few years of this work, particularly my manager at the time, and the opportunities the MoD provided to attend meetings with the "fathers" of safety critical software, and the "opportunity" to spend many silent nights in the Officer's mess in Benbecula, Outer Hebrides, during the long winter of 1995. I am also grateful to the Martyn Hall who allowed me to continue my studies when I joined the commercial world in the guise of CMG (and subsequently for the support of Clive Nicholls when CMG became LogicaCMG).

In addition, my gratitude must go to all those people I have met on planes, trains, ferries, airports, minibusses across Sweden, Finland, Norway, France, USA, Canada, Germany, Greece and the UK for silently putting up with me trying to take more space than usual at the odd table and back of chairs to progress this thesis, and to my friends for their precious time spent either encouraging me or making comments on various drafts of this thesis - there are many, but special thanks must go to Martin Kilb, Melanie Bonnar and Lee Ryder.

Finally, my greatest thanks go to my parents for letting me inherit their wisdom, determination, and intelligence, and crucially for what they gave up to provide me the opportunity to achieve this.

To my dear Mum, Dad and sister.

Contents

List of Tables

List of Figures

Part I

Safety Related Software Development

Chapter 1

Introduction

Software is being used in systems where a high-level of confidence in the correct operation of the system is required. Accidents, such as the radiation overdosing of patients using the Therac-25 cancer treating system ([LT93]), the overshooting of an Airbus A3XX aircraft at Warsaw airport ([Com94]) and the Ariane 5 rocket incident ([Lio96]) demonstrate that care is required in the construction of such systems and that there is perhaps still some way to go to achieve the high level of confidence expected by the general public. Informally these types of systems are referred to as *safety related* systems; and these in turn are one form of a class of systems called *high-integrity* systems.

This thesis will investigate the use of a simple mathematical model that can be used at different levels of abstraction in the development of Safety Related Systems; the aim being to develop processes that have the potential to reduce the cost of safety related software development and minimise the introduction of errors whilst crossing between different mathematical models currently used.

There are many (disparate) approaches proposed in the literature that allow the developer of a high integrity system to understand the (safety) requirements of these system and then to subsequently develop the software to be used in a controlled manner, producing the body of evidence necessary to demonstrate the system's correct

operation in a well defined environment.

Techniques, such as *mathematical correctness* and *refinement*, have been developed by others to increase confidence that an implemented system meets its specification, and confidence in the correctness of a specification can be increased by using *mathematical specification* techniques. All of these techniques have generally been borne out of research into four streams of approaches to high integrity systems: *dependability* (e.g. see the work on Predictably Dependable Computer Systems ([RLKL95], and [ESP94])), *safety engineering* (e.g. the work of Leverson ([Lev86])), *security*(e.g. financial systems) and *real time systems.* Each approach tackles similar problems of integrity demonstration but from different domain perspectives. Rushby provides a useful taxonomy of high integrity systems ([Rus94]), by comparing and drawing together the terminology used in the four approaches above.

The techniques described in the literature generally cover particular aspects in the development lifecycle, e.g. specifications using formal specification techniques or hardware components using hardware description languages. The lack of a single formalism for all phases implies there is additional effort required to translate and maintain correctness across different models if a formal approach is to be adopted from "cradle to grave". This potentially increases both development costs, due to different skill sets per development phase, and the opportunity for error introduction, during the transition between formalisms at the boundary of phases. It should be noted that it may not be necessary, appropriate or even commercially viable to apply formal techniques to all stages of the development, and as is often the case in safety related software development the risks to humans and/or environment needs to be ascertained before appropriate methods are used.

A problem with many of the techniques given in the literature is that they require the developer to become proficient in their specialised symbolism and are often based on mathematical concepts beyond those with a cursory mathematical background.

A driving motivation for this work is for it to be done in a formal notation that is readily accessible to engineers. Our choice of SCA meets this since:

- SCA networks have a graphical representation that allows easy understanding, for example a three module SCA can be represented as:

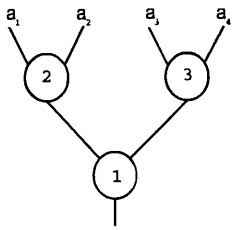

Figure 1.1: Simple SCA

- Values output by modules in an SCA can be specified using simple equations, for example the output of Module 1 can be represented by an equation such as:

$$V_1(t+1, a, x) = add\,(V_2(t, a, x), V_3(t, a, x))$$

- We will show that SCAs are applicable across a hierarchy of models of differing abstraction, reducing the need to be an expert in many different formalisms.

In 1961 McCarthy proposed that one of the goals of computational theory should be

> "....to represent computers as well as computations in a formalism that permits a treatment of the relation between a computation and the computer that carries out the computation." ([McC63]))

In their work on Synchronous Concurrent Algorithms, the mathematical model used in this thesis, Poole, Holden and Tucker presented the Integrative Hierarchy Problem:

> "Develop a mathematical theory that is able to relate and integrate different mathematical models at different levels of abstraction" [PHT98]

Poole, Holden and Tucker show that the construction of a hierarchy of Spatially Expanded Systems (SES) ([PHT98]), of which *Synchronous Concurrent Algorithms* are a form, is possible. They provide a mathematical framework that supports the demonstration of equivalence between SES's in a hierarchy. This thesis investigates whether Synchronous Concurrent Algorithms (SCAs), originally introduced by Tucker and Thompson in [TT85] and Thompson's PhD thesis, [Tho87] (but best described in the 1991 Technical Report from Swansea ([TT91]) which was subsequently updated as the 1994 Technical Report - [TT94]) and further expanded to handle non-unit delays by Hobley [Hob90] can be used in the development of safety related software and thus fulfill McCarthy's goal / the Integrative hierarchy problem.

The author's motivation for the work comes from a) his formulative career years in the UK Ministry of Defence dealing with the practical implementation of safety related systems on a variety of UK only, UK/US and European projects, and b) his undergraduate project that considered the implementation of dataflow architectures as a grid of processing elements, notably the work of Rumbaugh ([Rum77]). Implementations could either be as a grid architecture (e.g. The Manchester Prototype Dataflow Architecture, [GKW85]) or a token based architecture (e.g. Arvind's dataflow architecture with tagged tokens, [AP80]) - more information on dataflow architectures can be found in Sharpe's work, [Sha85]. Indeed, the initial thoughts of the for study after his bachelors degree was to determine how a grid architecture can be implemented as a single processor if all elements in the grid are executed under some form of sequential ordering.

The first part of the author's career, in the UK Ministry of Defence, gave an added aspect to these initial thoughts. During this period, he worked in a section focussed on safety related systems, and together with his knowledge on Synchronous

Concurrent Algorithms (SCAs) - which he was already aware could model hardware, and assumed could implement the dataflow graph in a formal manner - led to the pondering of whether the following development path for high integrity systems was valid:

- Formal specification of a system in a language such as Z, or B;
- Translation of the formal specification into a functional language;
- Animation of that specification to confirm correctness of specification;
- Creation of a dataflow graph of the functional language program;
- Implementation of the dataflow graph as a SCA;
- Implementation of the target architecture as a SCA; and
- Map implementation of the dataflow graph to implementation of the target architecture.

Informally, a SCA consists of a set of modules that calculate and communicate in parallel with respect to some external clock. Data is read into an SCA at a set of input modules, and can be read out of the SCA at a set of output modules. SCAs can be specified *algebraically*, and can therefore be algebraically manipulated. A refinement methodology is provided in this thesis under which a computation represented as a SCA can be transformed into a SCA modelling the computation device carrying out the computation.

The class of systems considered in this thesis is the sub-set of real-time systems known as *reactive systems*. The definition of a reactive system is given in Harel and Pnueli's work "On the development of reactive systems" ([HP85]). To summarise, a reactive system is defined to be a system that controls a set of actuators based on

the values read in from some set of sensors. Figure 1.2 shows an example reactive system.

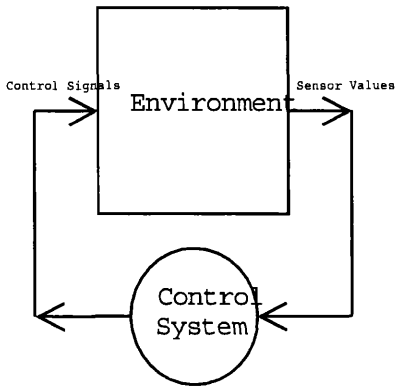

Figure 1.2: Example Reactive System

In such a system, there is a time delay between the reading in of values from the sensors, the performing of some processing on those signals and the resultant sending of control signals to the actuators. If it is stipulated that the reading in, processing, and sending of these signals are co-ordinated by some external clock, then reactive systems map to the notion of SCAs.

This thesis will present the usual model of SCAs and will then discuss a number of "limitations" identified in relation to this work. To address these limitations a number of syntactic extensions are introduced that support the notion of *refinement steps* in safety related software development. SCAs that use these extensions are known as *Dynamic Synchronous Concurrent Algorithms* (dSCAs) and it is useful to distinguish between two types of dSCAs: abstract dSCAs, which allow concepts such as the ability to look back over greater than one time unit; and concrete dSCA, which contain concrete implementations of the abstract concepts of an abstract dSCA, e.g. looking back greater than one time unit can be modelled as a finite tuple of memory values. We acknowledge the work of Hobley ([Hob90]) which first introduced the concept of non-unit delay SCAs, and how non-unit delay SCAs may be represented

as unit-delay SCAs, on which we build.

Each dSCA is given a defining shape, ∇, described in detail later, which is a tuple indicating the number of modules and the number of operations each particular module can perform for a dynamic network. By default, the defining shape of a Dynamic SCA directly representing a SCA with n modules would be $\nabla = (n, 1)$ indicating there are n modules, each capable of performing only 1 operation. Similarly, the defining shape of a simple computing device with one CPU executing a program with n operations would be $\nabla = (1, n)$.

This thesis takes advantage of the property of dSCAs that allow a dSCA with a particular defining shape to be folded into a dSCA with a different defining shape. Consider an algorithm which has 20 separate functions to be implemented; it could be implemented on a dSCA where $\nabla = (20, 1)$ - the usual notion of an SCA - or some other valid defining shape, some of which are $\nabla = (10, 2)$,$\nabla = (5, 4)$, $\nabla = (4, 5)$,$\nabla = (1, 20)$ - the last defining shape perhaps representing a single processor machine. Each dSCA can be algebraically specified (since they are SCAs) and thus it is hypothesised that it is possible to construct algebraic methods to transform between dSCAs of differing defining shapes.

As concrete and abstract dSCAs are SCAs with syntactic extensions, it can be further hypothesised that it is possible to construct algebraic methods to transform between SCAs and both forms of dSCAs. The transformations investigated are shown in Table 1.1.

Transformation	**Result**	**Result Represents**
Start	SCA	SCA representation of Algorithm
1	Abstract dSCA	dSCA representation of SCA
2	Abstract dSCA	Abstract hardware representation of dSCA
3	Concrete dSCA	Hardware implementation

Table 1.1: Transformations

Underlying each algebraic specification of SCAs and transformations is an algebra specification the defines the operations each module can perform, this is referred to as the machine algebra, or M_A.

The input into the transformation process shall be a (source) SCA representation of an algorithm it is wished to implement on a hardware system: the computation. Modules in this SCA must implement a single operation from M_A and the initial transformation (or refinement step) will take this SCA and produces an equivalent dSCA with a defining shape of $\nabla = (n, 1)$. The next refinement step/transformation shapes the resultant dSCA into a shape matching the defining shape of the target architecture, for example, a single processor machine with defining shape of $\nabla = (1, n)$. The final refinement step creates a concrete dSCA from an abstract dSCA. The resultant concrete dSCA represents the computer that carries out the computation.

To summarise, the main tools that this thesis uses from the literature are:

- Synchronous Concurrent Algorithms ([TT94])
- Non unit delay Synchronous Concurrent Algorithms ([Hob90])
- Hierarchy of Spatially Expanded Systems ([HTT89])

and the thesis provides:

- Dynamic Synchronous Concurrent Algorithms (abstract and concrete)
- Methods for the mechanical transformation between SCAs and abstract dSCAs, and further, abstract dSCAs to concrete dSCAs.

It is sensible to divide this thesis into three main sections. This section provides the introduction, and is followed by a section that introduces the original SCA model as

well as the syntactic extensions used to create abstract and concrete dSCAs. The final section introduces the refinement steps/transformations mentioned above as three separate transformations. Throughout this thesis the techniques are exposed through the use of a case study: The Generalised Railroad Crossing Problem, introduced later.

The remainder of this introductory section establishes the context of safety related software development by detailing the mathematical preliminaries and introducing the case study. Chapter 2 introduces a number of issues relating to the development of safety related systems, and includes an explanation of the environment that such developments take place in and highlights the applicable legislation. This chapter also provides a discussion on mathematical specification, correctness and refinement. Chapter 3 discusses the class of reactive systems and introduces the case study. Concluding this section is chapter 4 which presents the thesis statement, the contribution it is making and then describes the structure and organisation of the remainder of the text.

Chapter 2

Safety Related Software Development

The processes used in the development of Safety Related Software mark a return to the basic mathematical methods and techniques of decidability/computability from which computing initially emerged. Over the years development has diversified from a strict mathematical basis driven by the commercial reality of producing software in an environment where the target is the constant reduction in (development) costs. The risk of developing incorrectly functioning software, introduced by a non-mathematical approach, has been addressed, to some extent, by the emergence of the software engineering discipline.

In this chapter, key moments where a mathematical basis has been fundamental to the development of the computer field are discussed, from Church and Turing's 1930's exposition of computability to the work of Spivey and others on the formal mathematical specification of programs. The approach taken is not intended to provide a clearly recognisable path of developments or to single out individual "heroics", but rather to look at where mathematics has been applied to various stages of development and indicate those contributions we see as significant. In summary the class of computable functions, the progression of the computer field to computers and assemblers, and the development of high level languages are considered. The

correctness of programs and specifications is also considered.

2.1 Mathematical Evolution of Software development

Class of Computable Functions

Modern day computing stemmed from the need to address the questions posed by the field of mathematics known as computability theory - a topic that is addressed by Cutland ([Cut89]). Church ([Chu36b, Chu36a]) and Turing ([Tur36]) both identified models that could demonstrate the falseness of the Entscheidungsproblem - one of a number of problems posed by Hillbert and Ackermann in their 1928 work, "Grundzuge der Theoritischen Logik" ([AH28]). Both Church and Turing arrived at their solutions independently, Turing by introducing his logical computing machines (now known as Turing machines) and Church by the application of lambda-calculus. The closeness of each solution was identified by Kleene who stated:

> "So Turing's and Church's thesis are equivalent. We shall usually refer to them both as *Church's thesis*, or ... as the *Church-Turing thesis*" [Kle67].

The modern day understanding of Church's and Turing's work is that whatever can be calculated by a machine can be calculated by a Turing Machine. Since the precise class of problems that are Turing computable are known, there can be confidence that the computational limits of what can be implemented on/by a modern processor are well understood. See "Introduction to Metamathematics" chapters 12 and 13 by Kleene ([Kle52]) for perhaps the fullest summary of Turing computable problems.

Onto Computers and Assemblers

The 1940's work of von Neumann and others on computing machines (Neumann's original internal work has been published in many places, for example [Neu93]) led to the development of devices that could be successfully *programmed* and allowed

to compute on inputs - effectively allowing the implementation of the set of Turing computable problems as *stored programs.* Von Neuman's architecture is suited to the implementation of Turing machines. Later work on functional languages took Church's lambda-calculus forward as the basis of a machine architecture, and led us to the development of dataflow architectures, see 'A dataflow Architecture" by Rumbaugh ([Rum77]), amongst others. Backus had views on dataflow architectures which were given in his paper "Can programming be liberated from the von Neumann Style?" ([Bac78]).

In 1949 Wilkes ([Wil49]) showed that mnemonics codes, which had recently been used to design programs on paper before the process of hand translation into bit-wise machine code used by machines based on von Neuman's architecture, could be "compiled" by the EDSAC computer system he was using. Soon Wilkes added an ability for symbolic addressing ([Wil52, Wil53]) to his mnemonics, creating what is now referred to as assembly languages, and the programs used for translation were to become known as assemblers. By 1954 Backus was directing the implementation of assembler for the IBM 701, the Speedcoding system ([Bac54]), and it wasn't long before the development of high-level languages and compilers was being undertaken, notably Backus and others on FORTRAN ([BBB+57]) - Backus was later to play a major part in the development of mathematical formalisms for languages. In this period Böhm ([B54]) showed that a compiler for a language could be written in its own language thus providing the first seeds of a potential mechanism to demonstrate compiler correctness via a bootstrapping mechanism.

In 1961 McCarthy proposed 5 goals in his work "A Basis for the Mathematical Theory of Computation" ([McC63]); these are paraphrased below:

1. To develop a universal programming language.
2. To define a theory of equivalence of computation processes.

3. To represent algorithms by symbolic expressions.
4. To represent computers as well as computations in a formalism that permits a treatment of the relation between a computation and the computer that carries out the computation.
5. To give a quantitative theory of computation.

It is the fourth of these goals that is specifically considered in this thesis.

The Development of Formalisms for High-Level Languages

A mathematical basis to computing was still being applied as more abstract steps were taken away from bitwise machine code programs. Algol 60 ([Bac59], which was subsequently revised as [BBG+63]) was the first high-level language to have its syntax formally specified using Backus-Naur Form (BNF). BNF introduced the notion of grammars and formal semantics into high level language development and is closely linked to the work on context-free grammars performed by Chomsky in his research into the syntax of natural languages ([Cho56, Cho59]). In 1962 Floyd showed that ALGOL 60 was not a context-free language; and further, that any programming language where all programming variables must be declared before they are used, and where the names of these variables can be arbitrarily long, are not context-free either ([Flo62]). Floyd's result showed that most modern day programming languages are not context-free. The reader is pointed to Stephenson's work on "An Algebraic Approach to Syntax, Semantics and Compilation" ([Ste95]) for a good understanding of where the field has gone since Floyds work.

Correctness of Programs

Providing formalisms for programming languages led to questions being raised on how to demonstrate program correctness. Floyd continued his work on languages looking at the semantics of programs and trying to determine how meanings could be

assigned to programs. In his work, "Assigning Meanings to Programs", Floyd states that the paper:

> "attempts to provide an adequate basis for formal definition of the meanings of a programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs, including proofs of correctness, equivalence, and termination" [Flo67].

Initially Floyd considers correctness, equivalence and termination of a flowchart language by considering verification conditions for each component of the flowchart. Similar techniques were then applied to a subset of the ALGOL language by considering verification conditions for semantic units, i.e. ALGOL statements. The complexities of a high-level language begin to become clear during its exposition, and Floyd states that the introduction of "compound statements with bound local variables..causes some difficulties" [Flo67].

Interestingly, Floyd makes a passing remark on the use of the GOTO statement stating that:

> "transfers out of a block by go-to statements cause local variables to the block to become undefined" [Flo67]

Floyd also notes that his paper:

> "does not say that local variables loose their values upon leaving a block, but that preservation of their variables may not be assumed in proofs of programs." [Flo67]

Dijkstra continued the debate with his paper that identified the GOTO statement as being considered harmful ([Dij68]). It is easy to understand not just from Dijkstra's

viewpoint of creating confusing unmanageable code, but also from Floyds statements on the status of local variables upon leaving blocks that this type of branching is not welcome, and not seen, in Safety Related Software. Indeed, the 2001 ISO technical report on the use of Ada within high integrity systems, of which the author was the co-project editor, states very strongly that the goto statement is not included in high integrity systems since the use of goto:

> "is exceptional because its use is contrary to all principles of structured programming. There are no circumstances in which goto can be used where the use of some other construct is not preferable on grounds of good practice, readability, and aesthetics. Given this, the use of goto within high integrity systems is almost not an issue and the reasons for not using it..... are almost irrelevant." [ISO00] (also published in [Wea99])

An argument may be made that careful use of GOTO can be used for exception handling, however, the ISO guidance discourages the use of exceptions since its use makes verification more difficult, particularly for symbolic and functional analysis.

In this period Hoare's paper "An axiomatic basis for computer programming" ([Hoa69]) argued that a set of axioms and rules of inference can be gained from studying computer programs, and that these axioms and rules of inference can be used in formal proofs of the properties of computer systems. Hoare's work introduced the notion of pre and post conditions, where given a precondition P, a program Q and a description of the result of the programs execution R, then it could be written that:

$$P\,|Q|\,R$$

meaning that

> "if the assertion P is true before initiation of a program Q, then the assertion R is true on its completion." [Hoa69]

Hoare provides axiomatic rules for the majority of procedural language constructs including assignment, consequence, composition and iteration, and agues that:

- "When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics...; and
- ...but the practical advantages of program proving will eventually outweigh the difficulties, in view of the increasing cost of programming error"[Hoa69]

Several years later Dijkstra ([Dij75, Dij76]) introduced the concept of weakest preconditions and guarded commands in order to formally derive proofs of program correctness. Dijkstra's 1982 book "Selected Writings on Computing: A personal perspective" ([Dij82]) challenged the then growing perception that formal proofs are only usable for small toy programs.

The late 1960's saw the emergence of the "software crisis" and this led to the emergence of the field of *Software Engineering* (identified in [NR69]). This software engineering field has undoubtedly added structure and control to the development of software, and techniques such as Rapid Application Development have enabled more complex systems to be developed with a reduced number of errors, but limited guarantees that it is absolutely free from errors. When it comes to safety related software, where absolute reliance on correct operation is required, it is the author's view that formal methods must be applied, albeit with an appropriate amount of pragmatism.

NASA researchers Butler and Finelli, at the Langley research facilities, provide evidence in their report "The Infeasibility of Quantifying the Reliability of Life-Critical

Real-Time Software" ([BF93]) that the use of statistical methods, e.g. testing, is not feasible to ensure the reliability of high integrity systems.

A study by Hetzel ([Het84]) found that the probability of making a correct change is less than 50%. Hetzel identifies two main reasons why changes fail:

- "unforeseen side effects: the change accomplishes what it was supposed to, but also affects something that was working before.
- partial change completion: a change is applied to most parts of a system, but one or more parts are overlooked."[Het84]

Sommerville ([Som95]) indicates that removing X% of software faults does not imply that an X% increase in reliability will be observed. Indeed Sommerville notes that a particular study reported only a 3% increase in reliability after the removal of 60% of software faults. In summation, Sommerville proposes that the emphasis must be upon removing faults with the most serious consequences.

Formal methods offer an opportunity to ensure that a developed system meets its specification and that specification has resolved anomalies and omissions. However, as Sommerville points out, there is a risk that program proofs derived from the specification may be incorrect, or based upon assumptions on the system's environment which are incorrect. Additionally, it is the author's experience that industry in general see formal methods as cumbersome and expensive.

The Arianne rocket incident ([Lio96]) is an apt example of demonstrating that although a component may work correctly in one environment, no assumptions can be made about its operation in another environment. Modern safety practises require a safety case including statements regarding the environment the system has been built for to be produced.

For the sake of cost, and sanity, the use of formal methods should be targeted to those parts of a system where the biggest benefits will be obtained. Modern safety

related development methodologies require a system to be engineered to minimise, and compartmentalise, safety related software aspects.

Dijkstra's pre and post conditions can be seen in current safety related software development techniques where it is not unusual to supplement the chosen development languages with defined annotations associated with program code to enable automatic static analysis to take place. Languages/approaches such as Anna ([LvKBO87]) and SPARK Ada ([CG90, CGM92, Bar97]) are examples of where annotations are used to provide pre and post conditions for use in analysis by a static analysis tool.

Correctness of Specifications

The Software Project Managers Network (SPMN), an organisation established in the US in 1992 by the Assistant Secretary of the Navy to "identify proven industry and government software best practices and convey to .. Department of Defense system acquisition programs" , indicates that:

1. "Rework..." - *the process of having to go back to a previous part of the development process correct an issue and then redevelop* - "...is off the radar screen as a potential killer of cost and schedule.
2. First inspections are informal code walk-throughs despite the fact that metrics consistently show (i) impact of requirements and design defects is much greater than the impact of code defects and (ii) the cost of finding and fixing a defect grows very rapidly with the time between making and finding the defect.
3. The amount of rework done on the project is not tracked." [SPM]

Boehms work, Software Engineering Economics ([Boe81]), defined the relative cost of fixing an error introduced in the requirements phase depending upon where in the development phase it is found in. This work demonstrates that finding an error in the

maintenance phase is typically 100 times more expensive to fix than if it was found in requirements definition phase. In the safety context, any requirements error found in maintenance implies that an accident may already have occurred, and thus the cost would be many many more times higher, both in terms of human issues and potential litigation!

In "Analyzing Software Requirements Errors in Safety Critical, Embedded Systems", Lutz ([Lut93]) analysed the software errors found in the development of the Voyager and Galileo spacecraft software, and placed them within a framework based on Nakajo and Kume's error classification scheme (see [NK91]). Lutz places his work in context with a large number of other studies into software errors, and indicates that his study is the only one to consider complex safety related embedded systems. One of the conclusions of his work is the need to use formal specifications techniques in addition to natural language specifications in order to reduce number of errors. This conclusion is based on the fact that primary cause of functional faults was due to errors in understanding the requirements (62% on Voyager and 79% on Galileo).

Formal development is often thought of by industry as being expensive due to the need to have suitably qualified resources. The ConForm project, an application experiment under the European Commission's European Systems and Software Initiative (Grant 10670), demonstrated that the cost of using formal methods, across the whole lifecycle was close to that of conventional development - noting that additional effort required in the system design phase was recuperated in the reduction of effort for the other phases ([TBL96]).

Lightweight Formal Methods ([JW96, Jon96, DKLM98, ELC+98]) is an approach that brings the benefits of formal methods to the early stages of development, but acknowledges that the cost of completing the development in a formal manner may be disproportionate to the benefits gained. The ConForm project concluded that the

best use of formal methods was to target them at appropriate areas rather than have a blanket use (in the conclusions of [FBGL94]). Other authors have attempted to include formal methods within the traditional approaches such as SSADM, HOOD and Yourdon. For example Draper's work on integrating Z into the SSADM methodology "Practical Experiences of Z and SSADM" ([Dra92]); Giovanni and Iachini's work on HOOD and Z "HOOD and Z for the Development of Complex Systems" ([DI90]); and the work on Yourdon and Z by Semmens and Allen "Using Yourdon and Z: An Approach to Formal Specification" ([SA90]).

A useful survey on the industrial use of formal methods is given in the work conducted on behalf of the Canada's Atomic Energy Control Board, US National Institute of Science and Technology and the US Naval Research Laboratory entitled "International Survey of Industrial Applications of Formal Methods" ([CGR93]).

Leverson led a research group where one of the tasks has been looking at how to make formal methods more accessible to industry. Their paper, entitled "Investigating the Readability of State Based Formal Requirements Specification Languages" ([MZL02]) set out to understand one of the common complaints from industry that formal methods are difficult to read, and therefore require higher levels of training and more intelligent staff. Subjects taken from either a computer science or subject matter background, were shown specifications in a textual, graphical, tabular, and logical expressions of a Traffic alert and Collision Avoidance System (TCAS). The results show that background is an influential factor in understanding specifications - a good background in the specification method being used is better than a good background in the subject matter. Graphical approaches were useful when trying to understand overviews and tabular methods when looking at details. The textual specification provided was found to be not that helpful.

For a list of criteria that can be used when comparing the use of different formal specification methods in reactive systems the reader should consult Ardis et al, "A

Framework for Evaluating Specification Methods for Reactive Systems Experience Report" ([ACJ+96])

The reader is pointed to Jones and McCauley work entitled "Formal Methods - selected historical references" ([JM92]) for a considerable expansion on historical references than has been provided, including references to some of the technical and company reports that were later to lead to significant published efforts. Our intention has been to show the progression of development within a context of a mathematical basis, what is covered next are some of the current techniques used in the development of safety related software.

The result of this discussion could lead to the conclusion that the best return on effort would be from investigating the requirements phase, however this phase is adequately covered by other work, and the author is convinced that reducing the costs for other phases of development is beneficial.

2.2 Current Safety Related Software Development Techniques

Following on from his original comments in his 1969 paper, "An axiomatic basis for computer programming", ([Hoa69]) in which Hoare pointed out his thoughts that the advantages of program proof would eventually outweigh the difficulties, Hoare later reflects in his 1996 paper, "How did software get so reliable without proof?" ([Hoa96]) that the various predictions of doom and gloom given over the last 20 years regarding software safety have not materialised. Hoare's justification of why these problems have not occurred in such a predicted magnitude is interleaved within the previous discussion: the worry of these predictions becoming true have led to the introduction of various engineering techniques which have contributed to the reduction in errors. The author's view is that when it comes to human life, and safety related development, can we afford to take the risk of using "rule of thumb" techniques as opposed to formal

development?

To understand safety, the point at which a system becomes sufficiently important that confidence in its correct development and operation above that for other forms of software must be understand. Under UK law this point comes at the balance of risk: the developer of a system must show that the risk of an accident happening due to the system has been reduced to "as low as reasonably practical". In the UK, the concept of safety is effectively embodied within the Health and Safety At Work Act 1974 ([Gov84]), which requires the risk of danger to be **A**s **L**ow **A**s **R**easonably **P**racticable (the ALARP principle); UK case law, notably Donoghue vv Stevens in 1932, requires that a manufacturer owes a duty of care, not just to those at work, but to all persons to ensure that the systems they produce are safe and do not give rise to injury:

> "You must take reasonable care to avoid acts or omissions which you can reasonably foresee would be likely to injure your neighbour (persons who are so closely and directly affected by my act that I ought reasonably to have them in my contemplation as being so affected when I am directing my mind to the acts or omissions which are called into question)." [Dav93]

An accident occurring within UK jurisdiction could lead to criminal charges being brought under the Health and Safety At Work Act. However, it is increasingly common for civil charges to be brought as well, and there is a lower level of proof required for civil charges. Conviction of a civil charge usually results in the awarding of damages to the injured party and the attraction of bad publicity. For these reasons Davis states that:

> "it is perceived that civil liability is the most important." [Dav93]

Under civil law there are three areas of potential liability:

- Liability under contract law,
- Liability under the law of negligence, and
- Liability under the new product liability legislation

The seller of a safety related system must also ensure that the goods a) comply with their description, b) are of merchantable quality, and c) are fit for purpose.

Davis suggests that a corporation can protect itself from civil claims if they:

1. "ensure as a developer they have the necessary skills and knowledge to develop the system
2. use best practice, e.g. standards
3. Include a reasonable limit of liability in the contract, including
   (a) A requirement to comply with the instructions provided, and
   (b) A description of the operating environment, perhaps with a warning about other environments" [Dav93]

Burnett, ([Bur96]), gives a useful overview of the issues involved in developing safety related software as part of her work on the Rigorously Engineered Decisions (RED) project, a project forming part of the DTI/SERC Safety Critical Systems Research Initiative. Her views come from her position in a firm of solicitors dealing with IT and associated litigation.

Standards

The best defence a developer can use in a criminal/civil claim is the use of best practise. To reduce the need to establish best practise in each case, and to provide support to developers, many industries/professional bodies have captured what they consider to be best practise in standards. The generic standard applicable to all areas is now IEC 61508 ([IEC99]) where part 3 of this standard deals specifically with software. Examples of sector and national specific standards are:

1. European Space Agency: ESA Software Engineering Standards ([ESA91]);
2. US DoD: Military Standard 882B: System Safety Program Requirements ([DoD84]);
3. Nuclear Industry: IEC880 Software for Computers in the Safety Systems of Nuclear Power Stations ([IEC86]);
4. Medical Industry: IEC60196 Medical Electrical Equipment - Part 1: General Requirements for Safety 4: Collateral Standard: Programmable Electrical Medical Systems ([IEC96]);
5. Pharmaceutical: Supplier Guide for Validation of Automated Systems in Pharmaceutical Manufacture ([GAM]);
6. UK MoD: Defence Standard 00-55:The Procurement of Safety Related Software in Defence Equipment and Defence Standard 00-56:Safety Management Requirements for Defence Systems ([MOD89, MOD91] updated by [MOD97, MOD96]);
7. UK Railway Signalling: Safety Related Software for Railway Signalling ([RIA91]);
8. European Rail: Railway Applications: Software for Railway Control and Protection Systems ([CEN97]);

9. Airborne Civil Avionics: DO-178B/ED-12B : Software Considerations in Airborne Systems and Equipment Certification ([RTC92]) Issued in USA by the Requirements and Technical Concepts for Aviation and jointly in Europe by the European Organisation for Civil Aviation Electronics;

10. Motor Industry ([MISRA94]).

All of these standards provide guidance that allow the merging of the commercial reality of making profit and the need to adhere to the ALARP principle (or equivalent in other countries). Many standards introduce the principle of safety integrity levels, where, in brief, functional aspects of a system are graded based on level of risk, probability of accident occuring and severity of that accident. Often a number of techniques are suggested dependent upon the safety integrity level being claimed for functions of the system in a particular environment.

The UK Ministry of Defence was one of the first organisations to generate a standard relating specifically to software safety with Interim Defence Standard 00-55 ([MOD89]), which was also a unique / controversial standard in its mandating of the use of formal methods and formal development. It was quickly realised that it was not viable to have this standard sitting in isolation, as any requirement of its use left an ambiguous notion as to where the boundary of the use of formal methods would lie - in the worst case the whole development becomes safety related, and the cost of development is therefore probably very high. Interim Defence Standard 00-56 ([MOD91]) was introduced to provide project managers and developers a mechanism to identify what elements of a system are related critical. After a number of years of review and practical use (mainly studies) these standards achieved full Defence Standard status (i.e. they are now legally applicable to all contracts) and are referenced as [MOD97] and [MOD96].

In some particular industries, e.g. the UK Nuclear Power Industry, values for

acceptable accidents are laid down in an Act of Parliament. Two objectives are achieved by this: a) reassurances to the public that the executive of government believes that these values are correct, and b) provision of protection to the developers and safety auditors in what would be a very high profile case if something were to go wrong.

Some industries take other approaches, for example the UK railway industry has a formal licensing scheme introduced following the recommendations in the report into the UK Clapham Rail Disaster. See [WWG96] for an informal introduction to this scheme.

Language Choice

The most significant language in relation to safety related software development is Ada. Although the original language (Ada '83, [ISO87]) was not designed specifically for safety related development it's mandated use for defence systems in UK and US until the year 2000 means it has been subject to the most study. Its most recent version (Ada '95, [ISO95]) specifically addresses issues relating to high integrity in an annex (Annex H). Ada originated from the US Department of Defense's desire to standardise on one High Order Language for its software development programs. On the 28th January 1975 the Director of Defense Research and Engineering (DDRE) issued a memorandum requiring:

> "Military Departments to immediately formulate a program to assure maximum DoD software compatibility...the advantages in...training, instrumentation, module reutilization, program transportability, etc. are obvious" [Cur75]

Subsequently the Higher Order Language Working Group (HOLWG) was formed.

In the 1976 DDRE Memo covering the WOODENMAN version of the language requirements ([Cur76]) there is visibility of some of the issues relating to conflicting

requirements that are now considered good principles in developing safety related software . Examples of these "conflicting requirements" are a)Programming Ease vs. Safety from Programming Errors, and b)Object Efficiency vs. Program Clarity and Correctness. WOODENMAN concludes these issues by stating:

> "this tradeoff should be resolved in favor of error avoidance and against programming ease" and "...the major criteria in selecting a programming language should be clarity and correctness of programs within the constraint of allowing generation of extremely efficient object code when necessary". [Cur76]

An extensive evaluation took place of several languages and it was determined that no existing language met all the requirements the HOLWG was looking for, as recorded in [AWM+77]. A subsequent competition was held for a new language, and in May 1979 the US Secretary of Defense announced the winner of the competition to design and develop the new language ([Dun79]). Throughout the process the French, German and UK governments were involved, the UK providing substantial advice on language consolidation after previously performing its own similar exercise.

The reader is directed to Col. Whitaker USAF(Rtd)'s report in ACM SIGPLAN on the HOLWG for a personal view on the development of Ada and the details of the memos referred to above ([Whi93]). This work also indicates that Bell Laboratories, upon invitation to submit the C programming language ([ISO90, KR78]) for the evaluation, indicated that:

> "there was no chance of C meeting the requirements for readability, safety, etc." [Whi93]

Work in the UK, on behalf of the Motor Industry Research Association has tried to address the use of C in safety related systems. The use of C is wide spread throughout the commercial software development world, and therefore the cost of resource is

cheaper. Hatten's book, Safer C ([Hat95]), shows that there are a large number of constraints that must be placed on language constructs and a number of additional tools required before C can be used. In Germany, the TÜV has issued a set of "Rules for Programming in 'C"' ([FKPW96]). The related programming language C++ has also been considered in some areas, Binkley provides a technical report on the use of C++ from the US Government's National Institute for Standards and Technology Software, High Integrity Software Systems Assurance department ([Bin97]). In recent years other programming paradigms have been considered including functional languages under the UK Department of Trade and Industry (DTI) study SADLI ([CBB+96]).

In the 2004 British Computer Society Intelligent Catalogue (available on-line to members of the British Computer Society), of which the author of this thesis authored the Safety Engineering and Safety Assessment chapters, it is suggested that:

> "Language Choice - subsets of Ada, for example SPARK Ada, are the prime candidates for development of safety systems for good engineering reasons; however, this does not preclude the use of other languages. There are systems written in C, and functional languages, such as ML, have been used in research projects. However, the choice of language is guided by requirements given in the standard relevant for the field, the amount of tool support available, and the ability to demonstrate to peers why the developed system is safe. It is for these reasons subsets of Ada are usually used." [Tac04]

Even though it is often used, the programming language Ada itself is not fully suited to safety related software development, e.g. it includes the GOTO command. To address concerns, the safety field created subsets of Ada and several of these implementations include the ability to use annotations within a program to enable

static code examination. Two examples of this approach are Anna - A Language for Annotating Ada Programs ([LvKBO87]) - and SPARK - a safety related subset ([CG90, CGM92, Bar97]).

The applicability of Ada in the development of safety related software increased when the original Ada standard ([ISO87]) was updated, under ISO rules, and became Ada'95 ([ISO95]). This new standard became the first ISO language standard to specifically address issues relating to safety; detailed in Annex H of the standard. Despite this, the high integrity community decided that whilst Annex H addressed a number of issues, a clarification and re-emphasis document was required, and subsequently an ISO technical report was produced. The report is entitled "Guide for the Use of the Ada Programming Language in High Integrity Systems" ([ISO00],[ISO00]) [also published in draft as "The Use of Ada in High Integrity Systems" ([Wea99])]. The technical report identified the techniques shown Table 2.1 as being in current use in the development of high integrity software development to understand program correctness.

The technical report then goes on to consider all of Ada's language features, and applies one of the following tags to each feature against the techniques above:

- "**Included**: A feature is included if it is directly amenable to the designated verification technique....Included features enable the analysis to be undertaken and directly support the production of high integrity code.
- **Allowed**: A feature is allowed if the designated verification step is not straightforward, but is still achievable; or if the use of the feature is necessary and the use of a problematic verification technique can be effectively circumvented.
- **Excluded**: A feature is excluded if there is no current cost effective

Approach	Group Name	Technique
Analysis	Symbolic Analysis	Formal Code Verification Symbolic Execution
	Flow Analysis	Control Flow Data Flow Information Flow
	Stack Usage	Stack Usage
	Timing Analysis	Timing Analysis
	Range Checking	Range Checking
	Other Memory Usage	Other Memory Usage
	Object Code Analysis	Object Code Analysis
Testing	Structure-based Testing	MCDC
		Branch Coverage Structure Coverage
	Requirement-based Testing	Equivalence Class Boundary Value

Table 2.1: Techniques

way of undertaking the designated verification technique. Assurance of exclusion requires some form of verification" [ISO00]

The ISO technical report clearly describes how it's approach should be used, i.e.:

1. "the set of verification techniques should be determined from standards and guidelines the development is to take place under,
2. identify and understand the objectives to be satisfied by those techniques,
3. use the tables provided to determine what language features the technical report includes, allows or excludes and then finally
4. confirm the resulting subset and additional verification steps for any allowed features can actually satisfy the programming and verification requirements." [ISO00]

It is the author's view that any language considered for use in high integrity environment should go through a similar exercise.

Towards Safe Systems

The author's experience shows that Safety Related Software developments can use a disjoint set of tools, specialised to particular design phases, that means a consistent view of safety may be distorted when transitioning from one tool to another. In Heisel's Six Steps Towards Provably Safe Software, shown in Table 2.2 (([Hei95]), it can be seen that a major boundary comes when crossing from step 4 to step 5, and that these steps miss out the final part of a safety system, that of the hardware used to execute the software.

No	Step	Proof Obligations
1	Define the legal states of the system.	Show that the initial state is legal.
2	Define the actions the system can perform.	Analyze the conditions under which the actions transform legal states into legal states.
3	Define the interfaces of the system to the outside world.	Show that the internal system operations are only involved if their preconditions are satisfied. Show that for each combination of sensor values exactly only one internal operation is invoked. Show that - if the sensors work correctly - the system faithfully represents its environment.
4	Refine the data and operations of the specification until data and control structures of the target programming language can be used.	Show the correctness of the refinements.
5	Transform the specification in Step 4 into a form suitable for a program synthesis system.	Show the correctness of the algorithm performing this task.
6	Use the synthesis system to obtain a proven correct implementation of the system.	Proof obligations are generated by the synthesis system.

Table 2.2: Heisel's Six Steps Towards Provably Safe Software

Adding an additional seventh step, shown in Table 2.3, to Hiesel's 6 steps provides a path of development that would meet McCarthy's vision.

No	Step	Proof Obligations
7	Map the provably correct system from step 6 to the mathematical model of the hardware under consideration	Proof obligations are generated by the system that maps software implementation to hardware implementation

Table 2.3: Additional step to Heisel's Six Steps

2.3 Other Development Techniques

Standards, such as Defence Standard 00-56 ([MOD96]), require a developer to justify the safety integrity level particular functions in a system acquire. Once a particular level of integrity is identified for a function, the hardware and software elements that implement that function inherit that same level of integrity. It is possible, by design, to implement lower level integrity systems together to produce a complete system of higher integrity.

As a simplified example, consider a pipe in which molten metal will pass when required, but when it is not required to flow, people are likely to be standing underneath the pipe performing maintenance or other functions whilst the molten metal is held back (thus if the metal was to flow whist people were maintaining there could be casualties). It would be easy to understand that the system that prevents the molten metal flowing would have the highest integrity possible. Suppose this system is a simple valve, then there would have to be very stringent requirements on its construction. Perhaps it is even impossible to create a valve with such integrity. These requirements can be reduced by proposing two valves in series. Now both valve must fail before an possible accident could happen, and so it could be argued that each valve can now have less integrity because the two in series meet the integrity

requirement of the system.

For software compnets, one of a number of fault tolerance techniques could be applied - these techniques are amply addressed by the reports from the Predictably Dependable Computer Systems (PDCS) project, [RLKL95] and [ESP94]). As an example of a technique, consider n-version programming, introduced by Chen and Avizienis in [CA78]. In this technique multiple versions of the software are created independently, and are subsequently executed with their outputs being collected and examined by an external entity. This external entity then chooses which result it will use - perhaps on a majority voting algorithm or similar functionality. Leverson and Knight performed a much discussed experiment in n-version programming ([KL86]) in which they concluded that whilst a valuable technique, the assumption of independence of errors did not hold in their experiment and that the levels of improvement in reliability given by models was not achieved. An updated paper, ([KL90]), refutes many allegations made against this experiment - particularly those by Avizienis and his students. The Ariane 5 incident demonstrates that simply using copies of the same software/hardware running in parallel is not sufficient.

The use of fault tolerant techniques and sound engineering principles will continue to provide a reduction in failure rates below what could be expected. However, as systems get more complicated and is it less easy to partition off safety related aspects into distinct bounded parts of a system, the need to push forward with formal techniques increases. Hoare concluded [Hoa96] by suggesting that it was the push for formal methods that led to some of their principles being adopted by industry and that there is still much research that has not yet crossed into the commercial world. The work in this thesis is intended to be another step on the road of reducing complexity of mathematical models for developers and reducing costs. Isaksen, Bowen, and Nissanke ([IBN96]) provide a very comprehensive overview and bibliography of the techniques used in developing safety related software.

Chapter 3

Reactive Systems

3.1 Introduction

The system investigated in this thesis is one of a type of systems collectively known as *reactive systems* (Harel and Pnueli are credited as identifying this class of systems in their 1985 work "On the development of reactive systems" ([HP85])). In "Models for Reactivity" ([MP93]), Pnueli and Manna put reactive systems in context with more commonly talked about *real-time systems* by defining a hierarchy of models (where each subsequent model builds upon the previous model):

- "A reactive systems model that captures the *qualitative* (non quantitative) temporal precedence aspect of time. This model can only identify that one event precedes another but not by how much.
- A real-time systems model that captures the *metric* aspect of time in a reactive system. This model can measure the time elapsing between two events.
- A hybrid systems model that allows the inclusion of *continuous* components in a reactive real-time system. Such continuous components may cause continuous change in the values of some state variables according to some physical or control law". [MP93]

Correctness of Pnueli's reactive systems is addressed in Ketsen, Manno and Pnueli's work "Verification of Clocked and Hybrid Systems" ([KMP98]) where reactive systems are described as a clocked transition system.

This thesis will consider the class of reactive systems, and so the example is restricted to one where only the fact that one event precedes another can be identified, but not by how much.

It is useful to place this restriction so the thesis can concentrate on understanding transformations in the simplest form - one where there is no state information required to be handled by the system, and leave as the task for future work the application and potential modification of our techniques to real-time and hybrid systems.

The example selected for this thesis is the Generalised Railroad Crossing problem introduced in "A Benchmark for comparing different approaches for specifying and verifying real-time systems" ([HJL93]). Choosing such an example is aimed at showing that the processes defined in this thesis are applicable to real-life examples.

The common example found during literature surveys on safety related systems is that of a gas burner. The seminal definition of this problem is found in Ravl, Rischel and Hansen's work, "Specifying and Verifying Requirements of Real-Time Systems" ([RRH93]) delivered as part of the Provably Correct Systems I project. It was adopted by the Provably Correct Systems II (ProCos II) project as its case study, (see [HLOR93] for an overview of ProCos II and [BHL+96] for the ProCos II Final Report), and has been studied, amongst others by, Lamport in "Hybrid Systems in TLA+" ([Lam92]), using Temporal Logic of Actions (TLA+) (see [Lam91] for more on TLA+). Lano et al introduce a similar gas burner example in their paper "Design of Real-Time Control Systems for Event Driven Operations" ([LS97]). Both Bowen, in his work "Hardware Compilation of the ProCoS Gas Burner Case Study using Logic Programming" ([Bow96]), and Müller-Olm, in "Compiling the Gas Burner Case Study" ([MO95]) have looked at the compilation of the gas burner problem.

In contrast to the work on implementations, when considering a comparison of the specification, design and analysis of different formal verification techniques for real-time systems, Hietmeyer's Generalised Railroad Crossing (GRC) problem is the benchmark problem found in the literature. This problem is defined in "A Benchmark for comparing different approaches for specifying and verifying real-time systems" ([HJL93]). Hietmeyer and Lynch study this example in their MIT technical memo "The Generalized Railroad Crossing: A Case study in Formal Verification of Real-Time Systems" ([HL94a], which is also summarised in [HL94b]). Pnueli has studied the Generalized Railroad Problem, with details available in "Deductive Verification of Real-Time system using STeP" ([BMSU98]). Puchol has provided an ESTEREL solution described in "A Solution to the Generalized Railroad Crossing Problem in ESTEREL" ([Puc95]).Piveropoulos and Wellings cover the Requirements Engineering aspects of the GRC problem using the Σ-notation in "Requirements Engineering for Hard Real-Time Systems: the Σ Notation and a Case Study" ([PW99]).

Thus, both examples are well established in their relevant fields. The choice to progress with Heitmeyer and Lynch's GRC problem as the GRC problem was made since it is a reactive system that provides a sensibly sized example. This allows the proposed transformations to be exposed rather than worrying about a large state space (that the real-time gas burner problem introduces) and an overly complicated specification. Steggles has considered such a problem and demonstrated that (Second-order) algebraic approaches can be used to describe he problem and also discussed a form of functional refinement within his algebraic method (see either [Ste00a] and [Ste00b]).

3.2 Case Study - The Generalised Railroad Crossing Problem

The generalized railroad crossing (GRC) was introduced by Heitmeyer and Lynch to allow the comparison of the increasing number of formal methods being invented to specify, design and the analysis real-time systems. The problem is produced in its entirety below:

> "The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R, i.e. $I \subseteq R$. A set of trains travel through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R. To describe the system formally, a gate function is defined as $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up. Additionally, a set $\{\lambda_i\}$ of occupancy intervals is defined, where each occupancy interval is a time interval during which one or more of the trains are in I. The i^{th} occupancy interval is represented as $\lambda_i = [\tau_i, \upsilon_i]$, where τ_i is the i^{th} time of entry of a train into the crossing when no other train is in the crossing and υ_i is the first time since τ_i that no trains are in the crossing (i.e. the train that entered at τ_i has exited as have any trains that entered the crossing since τ_i)
>
> Given two constants ξ_1 and ξ_2, where $\xi_1 > 0$ and $\xi_2 > 0$, the problem is to develop a system to operate the gate that satisfies the following two properties:
>
> **Safety Property**: $t \in \bigcup_i \lambda_i \implies g(t) = 0$ (The gate is down during all occupancy intervals)

Utility Property:$t \notin \bigcup_i [\tau_i - \xi_1, \upsilon_i + \xi_2] \Longrightarrow g(t) = 90$ (The gate is up when no train is in the crossing)" [HJL93]

The system allows for there to be multiple trains in R at the same time.

Implementation of the GRC Problem

We will now consider an implementation of the GRC Problem, though it should be noted that we are not overly concerned with this being a complete and formally correct implementation. Rather, we wish to propose a solution that we can semantically reason is correct in order to focus on demonstrating our methods and techniques.

It is proposed to implement a solution to the sensor system described above for a region of interest, R, in which there are 2 tracks, tk_1 and tk_2. Each track will have two sensors sub-systems on it, one to the left of the gates and the other to the right of the gates, and each sensor sub-system is constructed from two sensors, each capable of counting how many trains have passed in a particular direction, with the intention being that one sensor captures trains moving into R and the other trains moving out of R (that is to say that they have cleared I).

There are two assumptions made about the property of trains travelling through R:

1. a train must continue through R in the same direction in which it entered, and
2. a train cannot cross between tracks whilst in R.

and there are two assumptions made about the system overall:

1. the length of a train is no greater than the distance between the boundaries of R and I, and
2. the time it takes a train to pass from the boundary of R to the boundary of I is such that the barriers can be fully lowered or raised.

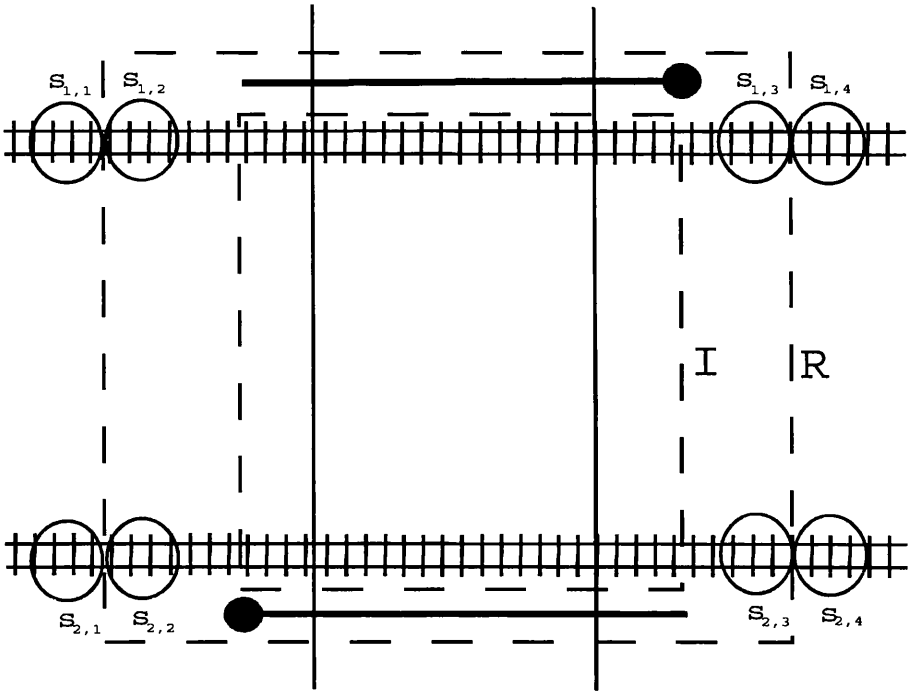

Figure 3.1: Simple Crossing System

Note that if one of the sensors fails then the system fails into a safe situation of the barriers down.

In order to identify whether there is a train in R for a particular track, the difference between the values held by the sensors is calculated. If this difference is zero, then there is no train in R, for that track.

Consider that track Tk_1 has sensors s_1 and s_2 in the sensor system on the left hand side of R, and sensors s_3 and s_4 in the sensor system to the right hand side of R. A train going left to right will first trip sensor s_1 on entering R and then s_3 on leaving; similarly a train going right to left will trip sensor s_4 on entering R and then s_2 on leaving R. To identify if a train is in R the following logical test is performed:

$$inR(t) = (s_1(t) - s_3(t) > 0) \vee (s_4(t) - s_2(t) > 0)$$

Where there is more than one track, sensor numbers will be annotated with the track number, thus a two track system will identify if a train is in R by performing the following logical test:

$$inR(t) = \begin{pmatrix} ((s_{1,1}(t) - s_{1,3}(t) > 0) \vee (s_{1,4}(t) - s_{1,2}(t) > 0)) \vee \\ ((s_{2,1}(t) - s_{2,3}(t) > 0) \vee (s_{2,4}(t) - s_{2,2}(t) > 0)) \end{pmatrix}$$

and generically, for tracks $Tk_1, \ldots, Tk_n$ the logic will be:

$$inR(t) = \begin{pmatrix} ((s_{1,1}(t) - s_{1,3}(t) > 0) \vee (s_{1,4}(t) - s_{1,2}(t) > 0)) \vee \\ \vdots \\ \vee((s_{n,1}(t) - s_{n,3}(t) > 0) \vee (s_{n,4}(t) - s_{n,2}(t) > 0)) \end{pmatrix}$$

For a practical implementation, the counter value will keep increasing to large values, however for our discussions this is not really an issue.

Let us assume that a train takes 2 time units to cross between sensors in R and that one enters at time 2. Tables 3.1 and 3.2 show the values of sensors and the logic above in this situation.

t	$s_{1,1}(t)$	$s_{1,2}(t)$	$s_{1,3}(t)$	$s_{1,4}(t)$	(a) $s_{1,1}(t) - s_{1,3}(t)$	(b) $s_{1,4}(t) - s_{1,2}(t)$
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	1	0	0	0	1	0
3	1	0	0	0	1	0
4	1	0	1	0	0	0
5	0	0	0	0	0	0

Table 3.1: One Train Passing Through R (Sensors)

Similarly, if the same train passes on Tk_1 entering at time 2 and another train passes on Tk_2 entering at time 3 then sensors s_1 and s_3 will trip for the train going left to right and sensors s_8 and s_6 for the train from right to left. Table 3.3 shows the values of sensors in this situation, and Tables 3.4 and 3.5 show the logic determining whether a train is in R.

t	(c) $a > 0$	(d) $b > 0$	(e) $c \vee d$
0	False	False	False
1	False	False	False
2	True	False	True
3	True	False	True
4	False	False	False
5	False	False	False

Table 3.2: One Train Passing Through R (Logic)

t	$s_{1,1}(t)$	$s_{1,2}(t)$	$s_{1,3}(t)$	$s_{1,4}(t)$	$s_{2,1}(t)$	$s_{2,2}(t)$	$s_{2,3}(t)$	$s_{2,4}(t)$
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	1
4	1	0	1	0	0	0	0	1
5	1	0	1	0	0	1	0	1
6	1	0	1	0	0	1	0	1

Table 3.3: Two Trains Passing Through R On Different Tracks

To meet the **Safety property** of the problem the sensors identifying the arrival of a train in I ($s_{n,1}$ and $s_{n,4}$) must be placed at the boundary of R in order to give the gate time to close. Similarly the sensors identifying the departure from I ($s_{n,2}$ and $s_{n.3}$) must be placed on the boundary of R (assuming no train is longer than the distance between boundaries of R and I). The values of ξ_1 and ξ_2 identified in the **Utility Property** are therefore directly related to the distance between boundaries of R and I, the maximum speed of trains travelling through R and the speed of the gate movement.

Determining whether a train is in R is not the final step in our implementation; the gates need to be implement. Recall that the definition of the problem introduced the gate function, $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means it is up. This is implemented as a sensor on the gate that provides an output in the required range.

t	(a) $s_{1,1}(t) - s_{1,3}(t)$	(b) $s_{1,4}(t) - s_{1,2}(t)$	(c) $s_{2,1}(t) - s_{2,3}(t)$	(d) $s_{2,4}(t) - s_{2,2}(t)$
0	0	0	0	0
1	0	0	0	0
2	1	0	0	0
3	1	0	0	1
4	0	0	0	1
5	0	0	0	0
6	0	0	0	0

Table 3.4: Two Trains Passing Through R On Different Tracks

t	(e) $a > 0$	(f) $b > 0$	(g) $c > 0$	(h) $d > 0$	(i) $((e \vee f) \vee (g \vee h))$
0	False	False	False	False	False
1	False	False	False	False	False
2	True	False	False	False	True
3	True	False	False	True	True
4	False	False	False	True	True
5	False	False	False	False	False
6	False	False	False	False	False

Table 3.5: Two Trains Passing Through R On Different Tracks

In addition a gate operation function is provided that will take the values from $g(t)$ and $inR(t)$ to determine if the gate motor should be sent the command *up*, *down* or *stay* according to the logic shown in Table 3.6.

$inR(t)$	$g(t)$	*Command*
False	$= 90$	stay
True	> 0	down
True	$= 0$	stay
False	< 90	up

Table 3.6: Gate Control Logic

We define $motor(t)$ as:

$$motor(t) = \left(\begin{array}{ll} if & \left(\begin{array}{l} ((inR(t) = False \wedge g(t) = 90) \vee \\ (inR(t) = True \wedge g(t) = 0) \end{array} \right) then\ stay \\ else & if\ (inR(t) = True \wedge g(t) > 0) \\ & then\ down \\ & else\ up \end{array} \right)$$

The correctness of this solution is discussed in chapter 8.2.

Chapter 4

Thesis Overview

In this chapter the scope of the remainder of this thesis is proposed by identifying the key elements discussed in the previous chapters and indicating the motivation for the technical work contained in the following chapters.

So far the notion of high integrity software/systems has been discussed, as have some details on the legal and professional pressures that drive the need for certain approaches to be taken when building such a high integrity system. It has been indicated that the author's view is that a formal approach is required, rather than reliance on extensive testing and that even in the US, where the author has some experience, the reliance on testing is being questioned by researchers in key institutions, such as NASA.

From these early chapters it can be seen that there are a number of differing techniques that can be used at differing life-cycle phases, but there is not really one technique that can traverse all levels. The discussion on language choice, sets the scene for future work on investigations on whether functional languages are suitable, and demonstrates that some research has been performed on translating Z to ML, and the use of functional languages in safety related systems, namely the SALDI project.

By introducing the GRC Problem as a case study a problem, and a solution, have been presented that will be returned to throughout this thesis to demonstrate the

necessary extensions to the existing Synchronous Concurrent Algorithms model, and how an implementation in one of the extended models may be transformed into an implementation in another of the extension models.

4.1 Thesis Statement

4.1.1 Scope

Within this thesis we shall focus on developing a method to support a number of transformations of an algorithm described as a Synchronous Concurrent Algorithm to its implementation on a piece of hardware, also described as a SCA. A formal basis is given to the methods by performing the transformations on the algebraic specification of SCAs, and having the transformations themselves defined as algebra specifications. For ease of discussion a pseudo algebraic format, that can trivially be converted to descriptions that may be used an algebraic specification tool, is used.

4.1.2 Contribution

The core contribution of this thesis are the:

- introduction of syntactic extensions to SCA theory ([TT94] and [HTT90]), named as dynamic SCAs (dSCAs), to solve the problem of being able to represent both the computation and the computing device that implements the computation in the same notation. Two forms of dSCAs are introduced:
  - Abstract dSCA;
  - Concrete dSCA.
- algebraic methods necessary to translate between a number of SCA models:
  - SCA to Abstract dSCA;

- Abstract dSCA to Abstract dSCA (with differing defining shapes); and
- Abstract dSCA to Concrete dSCA.

4.2 Thesis Structure

The work in this thesis falls naturally into four areas:

1. Introduction of SCAs and our extensions to this model (Chapters 5-8);
2. Transformations between various SCA models defined (Chapter 9-12); and
3. Summary and suggestions for future work (Chapter 13).

This chapter concludes Part I.

Part II commences with Chapter 5 which gives an overview and definition of Tucker and Thompson's SCAs and a short comparison of other applicable models. As well as an informal and formal definition of SCAs, it is discussed what it meant by saying a SCA is correct and how an SCA is specified algebraically. The final part of Chapter 5 looks at the use of SCAs in the literature and discusses some limitations of the original definition relating to the purposes of this thesis.

Next the extensions to SCA theory are introduced. These extensions address the limitations identified in Chapter 5. In Chapters 6 and 7 abstract and concrete dSCAs are introduced, respectively, in a similar style to that of Chapter 5. A return is then made to the case study and it is demonstrated in Chapter 8 how each SCA model can be used to provide an implementation.

In part III of this thesis three transformations used in the transformation (or refinement) of an SCA to a concrete dSCA are introduced. Firstly, Chapter 9 will introduce the concept of correct transformation, and discusses a number of fundamental specifications necessary for transformations. The first transformations, SCA to abstract dSCA, is defined in Chapter 10, with chapter 11 covering the abstract to

abstract dSCA (a process that allows the "reshaping" of the dSCA structure), and 12 covers the abstract dSCA to concrete dSCA transformations.

This thesis concludes in Chapter 13 with a discussion on proposed further work.

Part II

Synchronous Concurrent Algorithms

Chapter 5

Synchronous Concurrent Algorithms

5.1 Introduction

Synchronous Concurrent Algorithms are defined by Tucker and Thompson in their work "Equational Specifications of Synchronous Concurrent Algorithms" [TT91] and [TT94]. SCAs were introduced as a means of modelling the behaviour of a number of discrete processing elements, that communicate and process in parallel with respect to a single clock. This chapter introduces Tucker and Thompson's Synchronous Concurrent Algorithms in an informal and formal manner. Following the exposition of SCAs some other models from the literature are discussed, followed by how SCAs are currently used. The chapter concludes by a) demonstrating how SCAs can be specified algebraically, b) how the GRC problem can be implemented as an SCA and finally c) what limitations have been found with the SCA original model during the investigations for this thesis.

5.2 Informal Definition of SCAs

A Synchronous Concurrent Algorithm is a parallel algorithm consisting of a network of M modules connected by channels. The network communicates and computes in parallel over a data set A, with the communication and computation synchronised with respect to a clock, $T = \{0, 1, 2, \ldots\}$ which measures discrete time. Input and output to the network occurs at modules that are connected to sources and sinks, respectively. A representative network is shown in Figure 5.1 consisting of 3 modules, 4 sources - $a_1, \ldots, a_4$ - and one sink - the output of module 1.

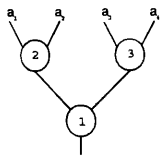

Figure 5.1: A Generalised SCA Network

Throughout this thesis it is implied that in the diagrams representing SCAs communication between modules will always travel in a downwards direction. For example in Figure 5.1 module 1 will receive its inputs from modules 2 and 3. This notion is clearly obvious when looking at an SCAs associated wiring function definition.

5.2.1 SCA Components

Data and Time: The algorithm processes data from a set A, at times t from a clock $T = \{0, 1, 2, \ldots\}$.

Channels: Modules within a SCA communicate via the channels of the network. Each channel has unit bandwidth, with respect to the carrier set A and each channel is uni-directionary, that is to say, a channel may only transmit a single datum $a \in A$ at any one time, in one direction. A channel may branch infinitely, with the intention

that the datum being transmitted along the channel is "copied" and transmitted along each of the new branches, but channels may never merge, since it would be difficult to determine which datum would be used on the merged channel.

Modules: Each module in the network is capable of subsuming and processing its inputs and producing one output. Consider the module $m_i \in M$, which has $n(i) \in N$ input channels, the processing performed by this module is defined by the total function $f_i : A^{n(i)} \rightarrow A$. The intent is that if the values $b_1, \ldots b_{n(i)} \in A$ arrive on $m_i's$ input channels, then m_i computes $f_i(b_1, \ldots, b_{n(i)})$.

Source Modules (Input): Data is read into the network at sources, also known as input modules. Sources have no input channels and a single output channel, which as with other channels, may branch. A network with n sources will process n input streams, $a_1, \ldots, a_n \in [T \rightarrow A]$ with the convention that $a_i(t)$ is supplied as input by source $i \in I_{in}$ at time $t \in T$.

Sink Modules (Output): Data is read out of the network by sink modules; by definition, sink modules have a single input channel and no output channels.

5.3 Formal Definition of SCA

Let N be a synchronous network over data set A with clock T. If N has $n > 0$ sources then the input to N is represented as a stream $a_1, \ldots, a_n \in [T \rightarrow A]$. It is also assumed that N has $k > 0$ modules, $m_1, \ldots m_k$ (where $\mathbb{N}_k = \{1, 2, \ldots, k\}$). Further, any vector $x_1, \ldots x_k \in A^k$ serves to specify the networks initial values, with the intention that module m_i initially holds the value x_i. The **termination assumption** is defined as follows:

> "At each time $t \in T$ a value is produced from each module. This value can always by determined uniquely from the time t, the set of inputs a,

and the set of initial values x." [TT91]

To support the termination assumption the value held by module m_i, at time t, can be determined by introducing functions $V_1, \ldots, V_k$ where for $i \in \mathbb{N}_k$ the following definition holds:

$$V_i : T \times [T \rightarrow A]^n \times A^k \rightarrow A$$

These functions are called the network's Value Functions. By exploiting the termination assumption and the synchronous nature of the network, the output of every module in the network is either specified initially, or is specified in terms of the values held at previous time cycles. Value Functions for a module are defined in two phases - the Initial State phase, where $t = 0$, and the State Transition phase, where $t > 0$, i.e. for module m_i the following is defined:

$$V_i(0, a, x)$$
$$V_i(t+1, a, x)$$

To complete the definition of Value Functions, it is necessary to define how the modules are wired together, and what length of delay is required when selecting the appropriate value from previous time cycles. This is achieved by the introduction of wiring functions, $\gamma(i,j)$ and $\beta(i,j)$, and a delay function $\delta_{i,j}(t, a, x)$.

Wiring Functions

Given the network N with $k > 0$ modules, $n > 0$ sources and modules $m_1, \ldots, m_k$, then m_i (where $i \in \mathbb{N}_k$) will have an associated function, f_i, that requires $n(i) > 0$ arguments and is defined as $f_i : A^{n(i)} \rightarrow A$. Each argument will arrive on $m_i's$ input channels and will be filled with data from the set A from either a source, or an adjacent module. Two operations, $\gamma(i,j)$ and $\beta(i,j)$ are introduced to identify whether module m_i's j^{th} input is from a source or an adjacent module, and the index of that source or module (where $j = 1, \ldots n(i)$).

The operation $\gamma(i,j)$, which indicates whether the input is from a source or a module, is defined as follows:

$$\gamma : \mathbb{N}_k \times \mathbb{N} \rightarrow \{S, M\}$$

where S indicates a source module and M indicates a module. The operation $\beta(i,j)$, which identifies the index of the source or module, is defined as:

$$\beta : \mathbb{N}_k \times \mathbb{N} \rightarrow \mathbb{N}_k$$

For wiring functions the following three conditions always hold for $i \in \mathbb{N}_k$ and $1 \leq j \leq n(i)$:

1. $\beta(i,j) \downarrow \wedge \gamma(i,j) \downarrow$ i.e. for all inputs $j = 1, \ldots, n(i)$ of all modules $i \in \mathbb{N}_k$ the wiring functions $\beta(i,j)$ and $\gamma(i,j)$ are defined.

2. $\gamma(i,j) = S \Rightarrow 1 \leq \beta(i,j) \leq n)$ with the intended meaning that if the j^{th} input channel of module m_i comes from a source, then the index of that source, provided by the β-wiring function, must be within the valid source indices $1, \ldots, n$.

3. $\gamma(i,j) = M \Rightarrow 1 \leq \beta(i,j) \leq k$ with the intended meaning that if the j^{th} input channel of module m_i comes from a module, then the index of that source, provided by the β-wiring function, must be within the valid module indices $1, \ldots, k$.

Delay Functions

For each input channel j of module m_i data, calculated at some previous time cycle, is selected. The delay function $\delta_{i,j}(t,a,x)$ identifies how many clock cycles ago the input was calculated, or was available at the source. The delay function is defined as:

$$\delta_{i,j} : T \times [T \rightarrow A]^n \times A^k \rightarrow T$$

The value of the delay is deliberately set so that it takes account of the current time, the current values of the input streams and the initial values of the network. To preclude the construction of predictive circuits, i.e. where the value of $\delta_{i,j}$ is such that it looks forward in time, a temporal condition is introduced such that for any time $t \in T$, inputs $a \in [T \rightarrow A]^n$, and initial values $x \in A^k$, the delay must be less that t:

$$\delta_{i,j}(t, a, x) < t$$

Early work on SCAs introduced the Unit Delay Assumption ([TT91]), which said that all delays would be of unit length. Hobley ([Hob90]) showed that this restriction was not necessary, and that a SCA with non-unit delay could be represented as a unit delay SCA if constructed, for example, using buffering in channels.

Value Function Initial State Phase

The Initial State phase for Value Functions defines the state of modules in N at time $t = 0$. Since x_i, the i^{th} element of the set of initial values x, is intentionally the value held by module m_i at time $t = 0$, it is appropriate to define, for $i \in \mathbb{N}_k$:

$$V_i(0, a, x) = x_i$$

Value Function State Transition Phase

The intention behind the module specification $f_i : A^{n(i)} \rightarrow A$ of module m_i is that if $b_1, \ldots, b_{n(i)}$ are the values selected by means of its delay functions $\delta_{i,1}, \ldots, \delta_{i,n(i)}$ from past data along its input channels, then $f_i(b_1, \ldots, b_{n(i)})$ is the value held at time t. However, for $j = 1, \ldots, n(i)$, the j^{th} input is either supplied by some source at some previous time, in which case, $b_j = a_q(\delta_{i,j}(t, a, x))$, or it is supplied by some other module in the network at some previous time, in which case $b_j = V_q(\delta_{i,j}(t, a, x), a, x)$. Accordingly, $V_i(t, a, x)$ is defined as:

$$V_i(t, a, x) = f_i(b_1, \ldots, b_{n(i)})$$

where for $j = 1, \ldots, n(i)$:

$$b_j = \begin{cases} a_q(\delta_{i,j}(t, a, x)) & \text{if } \gamma(i,j) = S \wedge \beta(i,j) = q \\ V_q(\delta_{i,j}(t, a, x), a, x) & \text{if } \gamma(i,j) = M \wedge \beta(i,j) = q \end{cases}$$

Network Output

V_{out} is defined as the vector representing the output from network N, consider that N has $m > 0$ sinks, then V_{out} would be constructed as $V_{out} = (V_{s_1}, \ldots, V_{s_m})$, where $s_1, \ldots, s_m \in \mathbb{N}_k$.

Note that, in terms of specification, V_{out} may be reformulated as the stream transformer $\widehat{V}_{out}$, such that the initial values can be considered as system parameters. In this case the stream transformer $\widehat{V}_{out}$ is defined as:

$$\widehat{V}_{out} : [T \rightarrow A]^n \times A^k \rightarrow [T \rightarrow A]^m$$

where:

$$\widehat{V}_{out}(a, x)(t) = V_{out}(t, a, x)$$

for any time $t \in T$, set of inputs $a \in [T \rightarrow A]^n$, and initial values $x \in A^k$.

However, it should also be noted that whilst $\widehat{V}_{out}$ may be a useful alternative form of specification to the Cartesian form originally given, there is a subtle problem associated with the implicit λ-abstraction on V_{out} used in the definition of $\widehat{V}_{out}$. Tucker and Thompson state the following theorem:

> "For any SCA over a set A and module functions $f_1, \ldots, f_k$, the local state functions $V_1, \ldots, V_k$, global state function, V_N, and output function V_{out} are primitive recursive over A_N. However, if A contains two or more elements, then $\widehat{V}_{out} \notin PR(A_N)$." [TT94]

Thus, although the definition of a Synchronous Concurrent Algorithm can be classed as primitive recursive, if the definition of $\widehat{V}_{out}$ contains more than one element,

then it itself is not (the reader is referred to [TT94] for a proof of this). The result has no impact on the proof of correctness of Synchronous Concurrent Algorithms, rather Tucker and Thompson imply it is better to deal with SCAs using the Cartesian form.

5.3.1 Example SCA

The following small SCA is introduced as a running example that will be referred to during the next few chapters before we consider the GRCP in full. Consider the 3 module SCA network N shown in Figure 5.2.

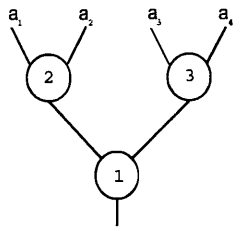

Figure 5.2: Example SCA

It is simple to define the delay functions for $i = \{1, 2, 3\}$ and $j = \{1, 2\}$ as follows:

$$\delta_{i,j}(t, a, x) = t - 1$$

with the wiring functions for N being defined as

$$\begin{array}{ll} \gamma(1,1) = M & \beta(1,1) = 1 \\ \gamma(1,2) = M & \beta(1,2) = 2 \\ \gamma(2,1) = S & \beta(2,1) = 1 \\ \gamma(2,2) = S & \beta(2,2) = 2 \\ \gamma(3,1) = S & \beta(3,1) = 3 \\ \gamma(3,2) = S & \beta(3,2) = 4 \end{array}$$

with the initial state vector being defined as $x = (1, 2, 3)$ the Value Functions can be defined in their two phases as:

$$
\begin{aligned}
&V_1(0,a,x) = x_1 \\
&V_2(0,a,x) = x_2 \\
&V_3(0,a,x) = x_3 \\
&V_1(t+1,a,x) = add\left(\begin{array}{l} V_{\beta(1,1)}(\delta_{1,1}\,(t+1,a,x),a,x), \\ V_{\beta(1,2)}(\delta_{1,1}\,(t+1,a,x),a,x) \end{array}\right) \\
&V_2(t+1,a,x) = sub\left(\begin{array}{l} a_{\beta(2,1)}(\delta_{1,1}\,(t+1,a,x)), \\ a_{\beta(2,2)}(\delta_{2,2}\,(t+1,a,x)) \end{array}\right) \\
&V_3(t+1,a,x) = sub\left(\begin{array}{l} a_{\beta(3,1)}(\delta_{3,1}\,(t+1,a,x)), \\ a_{\beta(3,2)}(\delta_{3,2}\,(t+1,a,x)) \end{array}\right)
\end{aligned}
$$

However, we will always write the definition of Value Functions out as the simplified equations:

$$
\begin{aligned}
&V_1(0,a,x) = 1 \\
&V_2(0,a,x) = 2 \\
&V_3(0,a,x) = 3 \\
&V_1(t+1,a,x) = add\,(V_2(t,a,x),V_3(t,a,x)) \\
&V_2(t+1,a,x) = sub\,(a_1(t),a_2(t)) \\
&V_3(t+1,a,x) = sub\,(a_3(t),a_4(t))
\end{aligned}
$$

5.4 Correctness

Thomson and Tucker introduce two types of correctness, Type I and Type II. Type I correctness focuses on the behaviour of a network's modules at particular times of the system clock T. The second type of correctness, Type II, focuses on the loading of input data and recovery of output data from an external environment. Type I correctness is analogous to traditional glass box testing, whereas Type II correctness is analogous to black box testing. Type II correctness is now considered in more detail as future correctness discussions in this thesis will be based on the ideas presented.

In Type II correctness, it is assumed that the user specification is based on another external clock $C = \{0, 1, 2, \ldots\}$ that is running slower than the system clock T of the network under consideration. If this is the case, then it is necessary to provide some form of scheduling of input streams - and possibly output streams.

It is assumed that the relation:

$$U \subseteq [C \rightarrow A]^n \times [C \rightarrow A]^m$$

specifies the computational task, or behaviour, such that for any set of inputs, $a \in [C \rightarrow A]^n$, and set of outputs, $b \in [C \rightarrow A]^m$:

$$U(a, b)$$

means that b is an acceptable stream of outputs for the stream of inputs a. We call U a type II system task relation. To meet this specification there must be a function that maps inputs to outputs. Let this function be Φ, and be defined as:

$$\Phi : [C \rightarrow A]^n \rightarrow [C \rightarrow A]^m$$

such that for any inputs $a \in [C \rightarrow A]^n$, it can be said that:

$$U(a, \Phi(a))$$

The design of a network that meets this specification is given by choosing a new clock T, with modules $m_1, \ldots, m_k$ and functional specifications $f_1, \ldots, f_k$, so that a n-source, k-module, m-sink network N can be constructed . This network is generally running with respect to a faster clock than the specification, and thus some scheduling of the inputs and outputs is needed to make the relation U still make sense. Scheduling can be modelled by the introduction of stream transformers θ_1 and θ_2, which are defined as:

$$\theta_1 : [C \rightarrow A]^m \rightarrow [T \rightarrow A]^m$$
$$\theta_2 : [T \rightarrow A]^n \rightarrow [C \rightarrow A]^n$$

The following should be noted:

- θ_1 and θ_2 need not be related
- θ_1 and θ_2 are part of the design that implements Φ.

There are no restrictions on the definition of θ_1 and θ_2, apart from insisting that they are primitive recursive over the appropriate algebra, and that they can perform copying of data and other useful tasks.

A special case is when θ_1 and θ_2 are determined by the clocks T and C - this arises when there is a deterministic relationship between the two clocks. This special case is also known as **retiming**, not to be confused with the retiming of Leiserson and Saxe ([LS91]), and is further discussed in Chapter 9.

To show correctness between the specification and design consideration needs to be given to θ_1, θ_2, and $\widehat{V}_{out}$. There is also a need to load the initial values into the specification (since they do not exist). For convenience, this can be incorporated within some initialisation operation within θ_1 by defining for a given initial state $x \in A^k$:

$$\theta_1^{init} : [C \rightarrow A]^m \times A^k \rightarrow [T \rightarrow A]^m \times A^k$$

by:

$$\theta_1^{init}(a, x) = (\theta_1(a), x)$$

To further increase convenience of the notation, θ_1^{init}, θ_2, and $\widehat{V}_{out}$ can be combined into one stream transformer specification, Ψ, defined as:

$$\Psi : [C \rightarrow A]^m \times A^k \rightarrow [C \rightarrow A]^n$$

defined for each $x \in A^k$ as:

$$\Psi = \theta_2(\widehat{V}_{out}(\theta_1^{init}))$$

that is for each $a \in [C \rightarrow A]^n$ and $x \in A^k$:

$$\Psi(a, x) = \theta_2(\widehat{V}_{out}(\theta_1^{init}(a), x))$$

Ψ is known as the external specification.

5.5 Use in Literature

SCAs have been used widely in the literature, but mainly stemming from the research carried out from the research groups originally in Leeds, and subsequently, Swansea.

Stephens provides a wide ranging summary of the use of Stream Transformers in the literature in the paper "Survey of Stream Transformers" ([Ste97]), which covers the use of SCAs. It is not intended to repeat this already accomplished work in this thesis, and will restrict this section to a short summary to demonstrate that SCAs are in actual use and not just a limited use tool.

Six useful papers are:

- Synchronous Concurrent Algorithms [TT85];
- Non-unit delay SCAs [Hob90];
- Algebraic specification of Synchronous Concurrent Algorithms and Architectures [TT91];
- Equational Specifications of Synchronous Concurrent Algorithms [TT94];
- Scope and limits of synchronous concurrent computations [MT88]; and
- Clocks, retimings, and the transformation of synchronous concurrent algorithms [HT94].

Hardware

There are a number of papers in the published literature relating to the application of SCAs to hardware specification and design issues:

- Formal specification and the design of verifiable computers[HT88];
- Specification and verification of synchronous concurrent algorithms: a case study of the Pixel Planes architecture[ET89a];
- Formal specification of a digital correlation ([HT90]);

- Specification and verification of synchronous concurrent algorithms: a case study of a convolution algorithm [HTT89];
- Clocks, retimings, and the formal specification of a UART ([HT89]);
- Consistent refinements of specifications for digital systems [HT91];
- Infinite synchronous concurrent algorithms: the specification and verification of a hardware stack [MT93].

Language

The literature contains several articles relating to synchronous languages and reactive systems, the following is a list of those specific to SCAs:

- A parallel deterministic language and its application to synchronous concurrent algorithms[TT88].

Biological

The SCA model has successfully been applied to entities outside the direct field of computing:

- Computational structure of neural systems [HTT90]; and
- An algorithmic model of the mammalian heart: propagation, vulnerability, reentry and fibrillation [HPT96].

Other

- Specification, derivation and verification of concurrent line drawing algorithms and architectures ([ET88]);
- Tools for the development of a rasteration algorithm[ET89b];

- Theoretical Considerations in algorithm design ([TT85]);
- Concurrent Assignment representation of synchronous systems ([MT87]) revised in ([MT89]); and
- Verification of synchronous concurrent algorithms using OBJ3. A case study of the Pixel Planes architecture ([EST91]).

The author has not been able to identify any source in the literature referring to the use of SCAs in Safety Related Software development.

5.6 Other Relevant Models

SCAs are not the only mathematical approach that could have been the basis for investigation in this thesis. The literature has many models that could have applied, a comprehensive overview of these models can be found written by Astesiano, Broy and Reggio (contained in Chapter 13 of [ABR99]). In this work, algebraic specification techniques are divided into (at least) four different approaches, and a simple case study is used to examine how the techniques are used. Techniques identified in that work and some others techniques are:

- Milner's Calculus of Communicating Systems (CCS) [Mil80];
- Hoare's Communicating Sequential Processes (CSP) [Hoa85];
- Baeten and Weijland's Algebra of Communicating Processes (ACP) - which is built up from a Basic Process Algebra [BW90];
- The International Organisation for Standardisation's Language of Temporal Ordering Systems (LOTOS) [ISO89];

- Process Specification Formalism (PSF) [MV89] and [MV90], which incorporates the Algebraic Specification Formalism (ASF) of Bergstra, Heering and Klint ([BHK89]);
- Petri Nets (original model defined in [Pet81]. Reisig ([Rei91]) considers Petri nets and algebraic specifications and in [Rei98] their use in specifying concurrent systems is considered; and
- Iterated Maps (see [FH98] for an example)

The author has chosen to use SCAs for their simplistic and readily accessible mathematical notation.

5.7 Algebraic Specification of SCAs

Synchronous Concurrent Algorithms will be specified in an algebraic style, based on the work of Ehrig and Mahr's "Fundamentals of Algebraic Specification" ([EM92]) and Wirsing's "Algebraic Specification" ([Wir90]). To understand the mathematical background to algebraic specifications the following elements need to be introduced:

1. Signatures;
2. Algebras;
3. Terms;
4. Equations; and
5. Specifications.

5.7.1 Mathematical Entities

Signatures

Informally by an algebra we mean a collection of sets $A_1, \ldots, A_n$, a collection of constants, $c_i \in A$, for $i \in I$, and a collection of operations (or functions):

$$f_i : A_{i_1} \times \cdots \times A_{i_n} \rightarrow A_{i_n+1}$$

To describe, compare and reason about such algebras syntactic names are given to each of these three kinds of objects. These names are collected together and organised into a many sorted signature.

For any non empty set $s \in S$ of sort names, an S-Sorted signature (Σ) is the $S^* \times S$ indexed collection of sets:

$$\Sigma = \{\Sigma_{w,s} | w = s_1, \ldots, s_n \in S^*, s \in S\}$$

where

- for the empty word $\lambda \in S^*$, and each $s \in S$, the element $c \in \Sigma_{\lambda,s}$ is called a constant symbols of sort name S.
- for each non-empty word $w = s_1, \ldots, s_n \in S^+$, and each $s \in S$, the element $f \in \Sigma_{w,s}$ is called a function name of domain type w, range type s, and arity n.

By a signature we mean a pair (S, Σ) consisting of the sort name set S, and the S-sorted signature Σ. We can write a signature in a more human readable way, such as:

Begin		
	Signature	A
	Sorts	$\ldots, A_i, \ldots$
	Constant Symbols	$\ldots, c_i, \ldots$
	Function Names	$\ldots, f_i : A_{s(1)} \times \ldots \times A_{s(n)} \rightarrow A_{s(i)}, \ldots$
End		

Algebras

Where a signature defines the syntactic objects, an algebra provides the semantics. If Σ is an S-Sorted signature, then an S-Sorted $\Sigma - Algebra$ is the ordered pair (A, Σ^A) where $A = \{A_s | s \in S\}$ is an s-indexed collection of sets.

For each sort name $s \in S$

- the set A_s is termed the carrier set of the algebra of sort names s
- Σ^A is an $S^* \times S$-indexed collection of sets:

$$\Sigma^A = \{\Sigma^A_{w,s} | w = s_1, \ldots, s_n \in S^*, s \in S\}$$

where

- for each sort name $s \in S$ and the empty word λ the following is defined:

$$\Sigma^A_{\lambda,s} = \{c_A | c \in \Sigma_{\lambda,s}\}$$

  where $c_A \in A$ is termed a constant of sort name $s \in S$ which interprets the constant name $c \in \Sigma_{\lambda,s}$ in the algebra.

- for each non-empty word $w = s_1, \ldots, s_n \in S^+$, and each $s \in S$, the following is defined:

$$\Sigma^A_{w,s} = \{f_A | f \in \Sigma_{w,s}\}$$

  where $f_A : A^w \rightarrow A_s$ is termed an operation with domain $A_{s_1} \times \ldots \times A_{s_n}$, range A_s and arity n which interprets the function name $f \in \Sigma_{w,s}$ in the algebra.

We can write an algebra out in a more human readable manner, such as:

Begin		
	Algerba	A
	Carriers	$\ldots, A_i, \ldots$
	Constant	$\ldots, c_i^A \rightarrow A_{s(i)}, \ldots$
	Operations	$\ldots, f_i^A : A_{s(1)} \times \ldots \times A_{s(n)} \rightarrow A_{s(i)}, \ldots$
End		

Terms

Let Σ be a non void S-Sorted signature and $X = \{x_s | s \in S\}$ be a family of sets of variables, then the term algebra is defined as:

$$T(\Sigma, X) = (T(\Sigma, X), \Sigma^{T(\Sigma,X)})$$

to be the $\Sigma - algebra$ with S-indexed family of carrier sets:

$$T(\Sigma, X) = \{\Sigma_{w,s}^{T(\Sigma,X)} | w = s_1, , s_n \in S^*, s \in S\}$$

as follows:

1. for any sort $s \in S$ and any constant symbol $c \in \Sigma_{\lambda,s}$, the following term can be defined:

$$c_{T(\Sigma,X)} = c_s$$

2. for any sort $s \in S$, any non-empty word $w = s_1, \ldots, s_n \in S^+$, functional name $f \in \Sigma_{w,s}$, and any terms $t_i \in T(\Sigma, X)_s$, for $1 \leq i \leq n$, the following term can be defined:

$$f_{T(\Sigma,X)}(t_1, \ldots, t_n) = f_s(t_1, \ldots, t_n)$$

This definition of terms allows the complex definition of new terms, e.g.

$$f_a\left(f_b\left(f_c\left(a\right), b\right), d, e, f\right)$$

Equations

Let Σ be a non void S-Sorted signature and $X = \{x_s | s \in S\}$ be a family of sets of variables and let $s \in S$ be any sort name. We define an equation , of sort name s, to be an expression of the form:

$$t_1 = t_2$$

where $t_1, t_2 \in T(\Sigma, X)_s$ are terms of sort s.

Specifications

Finally we can introduce the notion of an algebraic specification. An algebraic specification is the pair (Σ, E) where Σ is a signature and E a set of equations defining the exact behaviour of the function names within the signature. We define that the specification:

$$A = \left(\begin{array}{l} \ldots, A_i, \ldots; \ldots, c_i^A, \ldots; \ldots, \\ f_i^A : A_{s(1)} \times \ldots \times A_{s(n)} \rightarrow A_{s(i)}, \ldots, \\ f_i^A(a_{s(1)}, \ldots, a_{s(n)} = \ldots \end{array} \right)$$

can be written in a more human readable way as

Begin		
	Specification	A
	Sorts	$\ldots, A_i, \ldots$
	Constant Symbols	$\ldots, c_i, \ldots$
	Function Names	$\ldots, f_i : A_{s(1)} \times \ldots \times A_{s(n)} \rightarrow A_{s(i)}, \ldots$
	Equations	$\ldots, f_i^A(a_{s(1)}, \ldots, a_{s(n)} = \ldots$
End		

To further aid readability the notion of importing one specification into another is introduced; by importing specifications the ability to define a specification A in terms of another specification B is possible. Consider the specification B which is defined as:

Begin		
	Specification	B
	Sorts	$\ldots, B_j, \ldots$
	Constant Symbols	$\ldots, c_j, \ldots$
	Function Names	$\ldots, f_j : B_{s(1)} \times \ldots \times B_{s(n)} \rightarrow B_{s(i)}, \ldots$
	Equations	$\ldots, f_j(a_{s(1)}, \ldots, a_{s(n)} = \ldots$
End		

then the specification given as:

Begin		
	Specification	A
	Import	B
	Sorts	$\ldots, A_i, \ldots$
	Constant Symbols	$\ldots, c_i, \ldots$
	Function Names	$\ldots, f_i : A_{s(1)} \times \ldots \times A_{s(n)} \rightarrow A_{s(i)}, \ldots,$
		$\ldots, f_j : B_{s(1)} \times \ldots \times B_{s(n)} \rightarrow B_{s(i)}, \ldots,$
	Equations	$\ldots, f_i(a_{s(1)}, \ldots, a_{s(n)} = \ldots$
		$\ldots, f_j(a_{s(1)}, \ldots, a_{s(n)} = \ldots$
End		

is an informal way of writing:

$$A = \left(\begin{array}{l} \ldots, A_i, \ldots, B_j, \ldots; \\ \ldots, c_i, \ldots, c_j, \ldots; \\ \ldots, f_i : A_{s(1)} \times \ldots \times A_{s(n)} \rightarrow A_{s(i)}, \ldots; \\ \ldots, f_j : B_{s(1)} \times \ldots \times B_{s(m)} \rightarrow B_{s(j)}, \ldots; \\ \ldots, f_i(b_{s(1)}, \ldots, b_{s(n)}) = \ldots, \ldots\ldots, f_j(a_{s(1)}, \ldots, a_{s(m)} = \ldots \end{array} \right)$$

5.7.2 Algebraic Specification of SCAs

SCAs will be described in this thesis using a specific form of the algebraic specification just introduced. Firstly all equations used in the specification of an SCA are special cases of equations in that they are explicit definitions. Secondly, we introduce into our specification notation additional divisions of the signature and equation components. Our algebraic specification of an SCA will be:

Begin		
	Specification	SCA
	Import	T, A
	Sorts	
	Constant Symbols	
	VF Function Names	$V_i : T \times [T \to A]^n \times A^k \to A$
	γ **Function Names**	$\gamma : \mathbb{N}_k \times N \to \{S, M\}$
	β **Function Names**	$\beta : \mathbb{N}_k \times N \to N$
	δ **Function Names**	$\delta_{i,j} : T \times [T \to A]^n \times A^k \to T$
	γ **Equations**	$\gamma(i, j) = \ldots$
	β **Equations**	$\beta(i, j) = \ldots$
	δ **Equations**	$\delta_{i,j}(t, a, x) = \ldots$
	IV Equations	$V_i(0, a, x) = \ldots$
	ST Equations	$V_i(t + 1, a, x) = \ldots$
End		

Where we import 2 specifications, a clock specification (T) and a specification, A, that defines the data that goes over the SCA channels and the operations that Value Functions can implement.

It is a trivial task to turn the above algebra specification into an algebra specification able to be executed in an algebraic specification tool, e.g. Maude ([CDE+99]) by removing elements such as the InitialValueEqs section, and collapsing the individual operation and equation sections. Both of these are in the definition to aid construction and decomposition of SCA algebras. An example algebraic specification produced by the removal of the proposed syntax is as follows:

Begin		
	Specification	SCA
	Import	T, A
	Sorts	
	Constant Symbols	
	Function Names	$V_i : T \times [T \to A]^n \times A^k \to A$
		$\gamma : \mathbb{N}_k \times N \to \{S, M\}$
		$\beta : \mathbb{N}_k \times N \to N$
		$\delta_{i,j} : T \times [T \to A]^n \times A^k \to T$
	Equations	$\gamma(i, j) = \ldots$
		$\beta(i, j) = \ldots$
		$\delta_{i,j}(t, a, x) = \ldots$
		$V_i(0, a, x) = \ldots$
		$V_i(t + 1, a, x) = \ldots$
End		

Algebraic Specification of the Example SCA

Recall the simple SCA that was introduced in Chapter 5.3.1, as an algebraic specification in our style it would become (with appropriate specifications for A and T):

Begin

Specification	SCA
Import	T, A
Sorts	
Constant Symbols	
VF Function Names	$V_i : T \times [T \rightarrow A]^n \times A^k \rightarrow A$
γ **Function Names**	$\gamma : \mathbb{N}_k \times N \rightarrow \{S, M\}$
β **Function Names**	$\beta : \mathbb{N}_k \times N \rightarrow N$
δ **Function Names**	$\delta_{i,j} : T \times [T \rightarrow A]^n \times A^k \rightarrow T$
γ **Equations**	
	$\gamma(1,1) = M$
	$\gamma(1,2) = M$
	$\gamma(2,1) = S$
	$\gamma(2,2) = S$
	$\gamma(3,1) = S$
	$\gamma(3,2) = S$
β **Equations**	
	$\beta(1,1) = 1$
	$\beta(1,2) = 2$
	$\beta(2,1) = 1$
	$\beta(2,2) = 2$
	$\beta(3,1) = 3$
	$\beta(3,2) = 4$
δ **Equations**	
	$\delta_{1,1}(t,a,x) = t-1$
	$\delta_{1,2}(t,a,x) = t-1$
	$\delta_{2,1}(t,a,x) = t-1$
	$\delta_{2,2}(t,a,x) = t-1$
	$\delta_{3,1}(t,a,x) = t-1$
	$\delta_{3,2}(t,a,x) = t-1$
IV Equations	
	$V_1(0,a,x) = 1$
	$V_2(0,a,x) = 2$
	$V_3(0,a,x) = 3$
ST Equations	
	$V_1(t+1,a,x) = add\,(V_2(t,a,x), V_3(t,a,x))$
	$V_2(t+1,a,x) = sub\,(a_1(t), a_2(t))$
	$V_3(t+1,a,x) = sub\,(a_3(t), a_4(t))$

End

In the work performed to generate this thesis the author had no consistent access to an algebraic specification tool, the use of which would be useful in demonstrating the ease of modelling systems and the implementation of the transformations defined

later in this thesis. It is proposed that these definitions and transformations are placed into an algebraic specification tool as the first part of future work.

For the reader interested in a real life example, Appendix B contains an algebraic SCA specification of the Generalised Railroad Crossing Problem that is used as the case study.

5.8 Limitations of the Standard SCA Model

The simple translation of an SCA to hardware would require one "processor" per module. This is achievable if it were to be implemented using a technology such as Field Programmable Gate Arrays (FPGA), however this would not sit comfortably with safety related software standards. This is because development of a new FPGA is new hardware and would therefore introduce an untested and untried element to the solution. Additionally, use of a FPGA potentially drives a large through life cost as any upgrades/alterations would require new hardware. Ideally the solution would be implemented on a generic microprocessor that is in wide spread use so that a) known issues can be avoided, e.g. floating point problem with the Intel pentium processor ([Int94]) and b) there is a reduced cost of upgrade and ownership. Note that in recent years steps have been made to apply formal methods to hardware, e.g. Hunt's work on formally verifying the FM8501 microprocessor ([Hun94]), and Harman and Tucker's application of algebraic methods to correctness and verification of microprocessors ([HT93], [HT96] and [HT97]) and also the work of UK Ministry of Defence research establishment on the VIPER microprocessor ([Coh88] and [Coh89]).

The challenge addressed in this thesis is how to create an SCA that executes on one processor from an SCA that requires a processor for each module, or put another way, how is a single module SCA created that can alter its computation, wiring, and delay operations depending upon the current execution time? It is contended that this can be done using the existing model of SCAs, however the introduction of some

syntactic sugar makes the process cleaner and easier to understand. Since an SCA can be specified algebraically, then it is possible to create algebraic transformations that can be applied to manipulate a multi-module SCA into a single module SCA, through a hierarchy of such models. This hierarchy allows correctness proofs between each layer of abstraction in a style resembling refinement.

Consider again our simple example SCA, shown in Figure 5.3, representing a computation. It is intended to execute this computation on a single SCA module (which is how the computing device will later be represented).

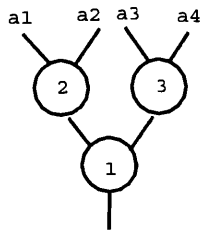

Figure 5.3: Sample SCA Network

In the simple example SCA network in Figure 5.3 each module performs a simple mathematical operation. If this network is implemented by a single module then there is the implication that each original network module is executed in sequence, e.g. at time $t = x$ the function from module 1 is executed, at time $t = x + 1$ the function from module 2 is executed and then at time $t = x + 2$ the function for the third module, module 3, is executed; to make semantic sense the computation would wrap around to execute module 1 at time $t = x + 3$ etc. To summarise, module 1 should execute at $t \ mod \ 3 = 0$, module 2 at $t \ mod \ 3 = 1$ and module 3 at $t \ mod \ 3 = 2$. Since the output of module 1 relies on the outputs of modules 2 and 3 in the original network, it must be possible to access data produced at times greater than the unit delay associated with the initial definition of SCAs, in this case at times $t - 1$ and

$t-2$. To summarise, what is required is an SCA that allows:

- Execution different functions at different times;
- Introduction of delays greater than the unit delay; and
- Alteration of the wiring of a module depending upon the time.

A new module is introduced into the network, a program counter, that starts at 0 when $t = 0$ and is incremented at each clock cycle. This program counter can be implemented either at each module or centrally, supplying values to all modules in the network. Forms of SCAs that act in such a way are referred to as *dynamic SCAs* (dSCAs).

Two forms of dynamic SCAs will soon be introduced - abstract and concrete. Abstract dSCAs are a simple syntactical extension to Tucker and Thompson's original definition, which will support:

1. a functional specification for each module in N, except the program counter, that contains a number of specific component operations, executing only one at a time, dependent upon the value of a counter;
2. variations in the delay function between modules of N dependent upon the value of a counter; and
3. variations in the wiring functions between modules in N dependent upon the value of a counter.

Concrete dSCAs implement the abstract principles modelled by an abstract dSCA. To achieve this requires the ability to store previously calculated values in tuples as outputs on the SCA channels - in a manner similar to that proposed in Hobley ([Hob90]). In a concrete dSCA functions to manage tuples are provided that allow

a consistent method for adding and retrieving values from the tuple. For this thesis, two of the more interesting tuple management techniques, queues and indexed arrays, will be considered.

5.9 Concluding Comments

Synchronous Concurrent Algorithms have been introduced, both informally and formally, and it has been shown that they currently have practical uses. In addition, it has been shown how a SCA can be specified algebraically.

To conclude, a number of limitations have been identified that need to be overcome in order for SCAs to be used in a refinement structure for taking an SCA representing a computation and producing an SCA that represents the computing device.

5.10 Sources

The initial work on Synchronous Concurrent Algorithms is an exposition of Tucker and Thompson's original work on SCAs, including some reference to the work of Hobley who showed that SCAs did not need to be restricted to unit delay. The author's first ideas for this thesis were inspired by his work for his undergraduate degree on dataflow architectures, and the subsequent desire to understand how SCAs could represent dataflow machines. It was further inspired by the author's initial career in the Ministry of Defence related to safety related systems.

Chapter 6

Abstract Dynamic Synchronous Concurrent Algorithms

Abstract dynamic SCA

To provide a model in which transformation of algorithm shape can take place with an understanding on impact of time, and, input and output streams.

6.1 Introduction

Abstract Dynamic Synchronous Concurrent Algorithms (referred to as abstract dSCAs) are introduced to overcome those limitations that have been identified with SCAs and to support transformations of the shape of a SCA. Next the elements of an abstract dSCA are informally described and then progression to a formal description is made. The chapter concludes with an introduction to the concepts of Defining Shape and Defining Size of an abstract dSCA and how to specify an abstract dSCA algebraically.

6.2 Informal Definition of Abstract dSCAs

Informally, an abstract dSCA module will execute a component specification where the component executed is selected based on the value of a program counter supplied to that module. Inputs to the component specification will be selected from the dSCA channels and inputs as indicated by the wiring functions, and values will be selected from previous calculations based on the delay function. Both the wiring and delay functions will also be bound to the value of the program counter, enabling a predictable but dynamically shaped SCA to be defined.

Before examining abstract dSCA components this thesis will consider the three (syntactic) differences introduced for abstract dSCAs that address the limitations identified with SCAs, namely:

1. Increasing number of functional specifications per module.
2. Relaxing unit delay assumption.
3. Variable wiring functions.

6.2.1 Increasing Number of Functional Specifications per Module

A SCA computes values using on a single functional specification per module. In the dSCA framework it is intended to choose a functional specification based upon a particular value of a program counter, allowing one module to implement many functions. To achieve this using simple syntactic extensions, a module's specification will be constructed from a number of component functional specifications with the correct component selected by means of referring to an externally provided program counter. This program counter value will be supplied to modules as the first argument.

Each module has the same finite number of component functional specifications,

$f_{i,0}, \ldots, f_{i,Max_N-1}$. This finite number is defined as Max_N, and this fact has three implications:

1. At some values of the program counter the module may be performing some null calculation.

2. At some values of the program counter the functional specification may not use all the arguments in the overall module specification.

3. In the traditional SCA model there is a single equation defining the Value Functions Initial State phase and a single equation defining the State Transition phase of the Value Function. In the dSCA model, a Value Functions initial state phase must be provided for each component functional specification, i.e. there are Max_N initial state equations representing the value for program counter values $pc = 0, 1, 2, \ldots, Max_N - 1$ at times $t = 0, 1, 2, \ldots, Max_N - 1$.

The program counter value could be generated in three different ways:

1. a program counter per module (our investigations have shown that this approach presents clumsy manual transformations and required additional proofs that all program counters are set to zero at the start of execution and that they all increase in step)

2. include the program counter value as part of the output such that the module has 2 outputs, one containing the functional output and the other the updated clock value. Again, proofs need to be shown that all values are set properly and are suitably incremented.

3. a single globalised program counter that provides its output to all modules.

Since this thesis will look at manual transformations the author has decided to minimise the burden of transformation by generating the program counter using the third option - the global program counter.

Allowing multiple functions on a module introduces a problem of cycle consistency, where results required were calculated more then Max_N cycles ago. The notion of cycle consistency is discussed in detail in chapter 6.2.3, since to make a sensible discussion required the introduction of non-unit delays.

6.2.2 Relaxing Unit Delay Assumption

The delay function in an SCA allowed the retrieval of data calculated at previous times - the original convention was the SCAs adhered to the unit delay assumption. In a SCA with unit time delay, the definition of one Initial State phase and one State Transition phase for Value Functions prevents the lookup of values where $t < 0$; since at time $t = 0$ the Initial State phase determines the exact value output by the module and at times $t > 0$ the State Transition phase dictates the restriction by using inputs calculated at most $t - 1$ time units ago).

Hobley ([Hob90]) identified that restricting SCAs to unit delay was not necessary, and further that a non-unit SCA could be represented as a unit delay if an appropriate buffering of data was instigated (either in the channels or in the modules). Our work, for the relaxing of the unit delay assumptions, draws heavily from Hobley's work, but notably:

- constrains the implementation to meet the needs of future transformations;
- introduces the notion of cycle consistency and cycle inconsistency; and
- generalises Hobley's implementation, to produce a more flexible management of buffering and a simplified syntax.

If the unit delay assumption is relaxed then it cannot be guaranteed that values from times $t \leq 0$ are not requested if there are only definitions for $t = 0$ and $t + 1$. In chapter 6.2.1 the value Max_N was introduced to indicate the number of components in the functional specification. It is therefore defined that if there are Max_N components in the specification then there must be Max_N initial values, for times $0, 1, \ldots, Max_N - 1$.

It is immediately tempting to bound the delays allowed to be no greater than Max_N, however the concept of cyclic consistency needs to be considered. This concept is discussed in the next section, but has the following implication: let *pc_now* be the current value of the program counter, and $pc\_res_1, \ldots, pc\_res_{n(i)}$ represent the values of the program counter when the functions that produce results that m_i uses, then:

1. if $pc\_now < pc\_res_1, \ldots, pc\_res_{n(i)} < Max_N$ then the delay required is within the range:

$$1, \ldots, Max_N$$

2. Otherwise the delay required is in the range:

$$Max_N + 1, \ldots, 2 \times 2 * Max_N$$

6.2.3 Cycle Consistency

Allowing multiple functions per module implies that there is an execution order for those functions. The investigations carried out in the author's work has ascertained that certain execution orders can introduce potential temporal issues.

Consider a module that executes 5 modules, thus $Max_N = 5$, then a cycle can be defined to be any consecutive time period $[t_1, t_2, t_3, t_4, t_5]$ where $\frac{t_1}{5} = 0$ and $t_5 = t_4 + 1, t_4 = t_3 + 1, t_3 = t_2 + 1, t_2 = t_1 + 1$.

If any time *pc_now* within the range $[t_1, \ldots, t_5]$ is chosen then the component functional specification executing at module m_i at that point in time will select its

inputs from the channels (or inputs) of the network calculated at some previous time, as indicated by the delay functions. The following lemma is introduced:

Lemma 6.2.1. *The execution order chosen for the modules will have a direct affect on the period of delay functions required.*

In any execution order the component specifications providing results required for the functional specification operating at *pc_now* will be executed on some module at values of the program counter corresponding to $pc\_res_1, \ldots, pc\_res_{n(i)}$.

Let us consider the case where:

$$pc\_res_1, \ldots, pc\_res_{n(i)} > pc\_now$$

then there are three cases that need considering:

1. If *pc_now* is at the start of a cycle, for example when $t = Max_N$, then since the functions that will create values for its inputs will not have executed yet, then it is the case that the values it requires (from the initial state) are within the range $0 < t < Max_N$;

2. If *pc_now* is at the last point in the cycle it can be for its inputs to be calculated after it, e.g. $t = Max_N + (Max_N - 2)$ for a functional specification with 2 inputs, then its inputs must be found at times $t = Max_N - 2$ and $t = Max_N - 1$; and

3. If *pc_now* is after the last point in the cycle it can be for its inputs to be calculated after it then the system includes a loop - which is not allowed.

If for all values of the program counter it can be shown that functions that calculate inputs happen at program counter values higher than the one under consideration, then the abstract is defined to be a *cycle consistent* abstract dSCA.

A *cycle inconsistent* abstract dSCA is one where for some values of the program counter on some modules, the component specification for any result is executed

earlier in the execution order. In such a case, the delay will always be greater than Max_N for that particular input.

Further, a *totally cycle inconsistent* abstract dSCA is one where for all values of the program counter, and all modules, the component specification for any result is executed earlier in the execution order.

The effort required to show that a cycle inconsistent abstract dSCA is not totally cycle inconsistent may be too great and thus any cycle inconsistent abstract dSCA can be treated as totally cycle inconsistent. The implication of this is immaterial for abstract dSCAs but has space implications for concrete dSCAs, this is discussed later.

Demonstration Consider the 3 module SCA shown in Figure 6.1 which will be implemented as a one module abstract dSCA.

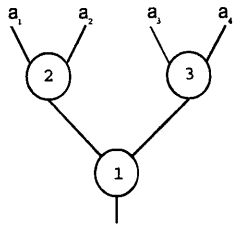

Figure 6.1: Execution Order Example SCA

The Value Functions for the SCA are given as:

$$
\begin{aligned}
V_1(0, a, x) &= 1 \\
V_2(0, a, x) &= 2 \\
V_3(0, a, x) &= 3 \\
V_1(t+1, a, x) &= add\,(V_2(t, a, x), V_3(t, a, x)) \\
V_2(t+1, a, x) &= sub\,(a_1(t), a_2(t)) \\
V_3(t+1, a, x) &= sub\,(a_3(t), a_4(t))
\end{aligned}
$$

The detail of the wiring and delay functions for the SCA are not given here as they are not necessary for understanding (if the reader so wishes they may be

easily constructed by examining the network structure of the Value Functions). Let $V_{out} = V_1$ and given the input streams below:

$$a_1 = (7, 9, 5, 4, 6, 8, 3, 5, 6)$$
$$a_2 = (1, 8, 3, 0, 5, 8, 1, 1, 2)$$
$$a_3 = (5, 12, 5, 7, 8, 9, 5, 8, 12)$$
$$a_4 = (4, 4, 4, 4, 4, 4, 4, 4, 4)$$

then the execution of the SCA can be traced as shown in Table 6.1, where the value of V_{out} at every clock cycle is given in the last row of the table.

Time	0	1	2	3	4	5	6	7	8	9
Mod 1 Val	1	5	7	9	3	7	5	5	3	8
Mod 2 Val	2	6	1	2	4	1	0	2	4	4
Mod 3 Val	3	1	8	1	3	4	5	1	4	8
$V_{out}(t, a, x)$	1	5	7	9	3	7	5	5	3	8

Table 6.1: SCA Execution Trace

Cycle Consistent abstract dSCA

The creation of a cycle consistent abstract dSCA will produce an abstract dSCA where the functional specification that produces the results for any other functional specification occurs later in the execution order. This is easily achieved in this example by implementing the execution order of 1-2-3 (another viable alternative is the order 1-3-2).

In order to provide consistent inputs to the network the input stream must be delayed by Max_N clock cycles, to take account of an initialisation period, and then each input value must be held for Max_N clock cycles i.e. for the length of a cycle. For this example, the input streams would be rescheduled as:

$$a_1 = (u, u, u, 7, 7, 7, 9, 9, 9, 5)$$
$$a_2 = (u, u, u, 1, 1, 1, 8, 8, 8, 3)$$
$$a_3 = (u, u, u, 5, 5, 5, 12, 12, 12, 5)$$
$$a_4 = (u, u, u, 4, 4, 4, 4, 4, 4, 4)$$

The value functions for the one module cycle consistent dSCA will be constructed as:

$$V_1(0,a,x) = (1,0,0)$$
$$V_1(1,a,x) = (1,2,0)$$
$$V_1(2,a,x) = (1,2,3)$$
$$V_1(t+1,a,x) = \begin{cases} add\,(V_1(t-1,a,x), V_1(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\ sub\,(a_1(t), a_2(t)) & \text{if } V_{pc}(t,a,x) = 1 \\ sub\,(a_3(t), a_4(t)) & \text{if } V_{pc}(t,a,x) = 2 \end{cases}$$

The program counter is defined as:

$$V_{pc}(0,a,x) = 0$$
$$V_{pc}(t+1,a,x) = mod\,(add\,(V_{pc}(t,a,x),1)\,.Max_N)$$

By examining the value function definition for V_1 it can be seen that the first component of the specification represents module 1 in the SCA, i.e. when the program counter value is 0, and the second and third components represent modules 2 and 3 of the SCA respectively.

If $V_{out} = V_1$ then it is possible to trace the values output as time progresses. The results of this tracing can be seen in Table 6.2.

Time	0	1	2	3	4	5	6	7	8	9
PC Val	0	1	2	0	1	2	0	1	2	0
Module Val	1	2	3	5	6	1	7	1	8	9
$V_{out}(t,a,x)$	**1**	2	3	**5**	6	1	**7**	1	8	**9**

Table 6.2: 1-2-3-Execution Trace

As before, the results of V_{out} are shown on the last row of the table, but in this case only the results produced on every 3^{rd} clock cycle from time $t = 0$ are of interest (the others results are intermediate results). The results that are of interest are shown in bold.

Comparing Tables 6.2 with 6.1 it can be observed that the correct set of results are computed every 3rd clock cycle.

Cycle Inconsistent abstract dSCA

For a cycle inconsistent abstract dSCA the execution order of 2-3-1 is considered. The naive, and incorrect, implementation would produce something akin to the following dSCA:

$$V_1(0,a,x) = 2$$
$$V_1(1,a,x) = 3$$
$$V_1(2,a,x) = 1$$
$$V_1(t+1,a,x) = \begin{cases} sub\,(a_1(t), a_2(t)) & \text{if } V_{pc}(t,a,x) = 0 \\ sub\,(a_3(t), a_4(t)) & \text{if } V_{pc}(t,a,x) = 1 \\ add\,(V_1(t-2,a,x), V_1(t-1,a,x)) & \text{if } V_{pc}(t,a,x) = 2 \end{cases}$$

In this example the component specification representing module 1 in the SCA is executed last in the execution order, and the implementation simply tries to match the delay function for its inputs to be in the same cycle as it. The result trace for such an execution (assuming a similar rescheduling of inputs) is shown in Table 6.3.

Time	0	1	2	3	4	5	6	7	8	9
PC Val	0	1	2	0	1	2	0	1	2	0
Mod Val	2	3	1	6	1	7	1	8	9	2
$V_{out}(t,a,x)$	2	3	**1**	6	1	**7**	1	8	**9**	2

Table 6.3: Wrong 2-3-1-Execution Trace

Again the important results are highlighted in bold whereas the other results are intermediatory results. The first results is produced at time $t = 2$ which is fine, however the second result is 7 and not 5 as expected. Looking at the intermediate results it can be seen that the result required for the calculation are actually computed at times $t-5$ and $t-4$ (or put another way at $t-2*Max_N-1$ and $t-2*Max_N-2$).

If the definition of the Value Function is given to reflect these timings, then the following abstract dSCA being defined:

$$V_1(0,a,x) = 2$$
$$V_1(1,a,x) = 3$$
$$V_1(2,a,x) = 1$$
$$V_1(t+1,a,x) = \begin{cases} sub\,(a_1(t), a_2(t)) & \text{if } V_{pc}(t,a,x) = 0 \\ sub\,(a_3(t), a_4(t)) & \text{if } V_{pc}(t,a,x) = 1 \\ add\,(V_1(t-5,a,x), V_1(t-4,a,x)) & \text{if } V_{pc}(t,a,x) = 2 \end{cases}$$

This abstract dSCA results in the trace shown in Table 6.4.

Time	0	1	2	3	4	5	6	7	8	9	10	11
PC Val	0	1	2	0	1	2	0	1	2	0	1	2
Mod Val	2	3	1	6	1	5	1	8	7	2	1	9
$V_{out}(t,a,x)$	2	3	**1**	6	1	**5**	1	8	**7**	2	1	**9**

Table 6.4: Correct 2-3-1-Execution Trace

It can now be seen that the correct set of results (shown in bold) are obtained every 3rd cycle after an initial delay of 3 clock cycles.

This difference in start time of correct results is easily managed using retimings if a formal syntactic proof of correctness were to be performed.

6.2.4 Variable Wiring Functions

The introduction of a component based functional specification means that not all inputs must be wired to the same modules for all values of the program counter (it is also not necessarily true that all the inputs to a module are wired up for that particular value of the program counter). The SCA definition of wiring functions are extended to include the program counter as an index - $\gamma_{pc}(i,j)$ and $\beta_{pc}(i,j)$. Where the component of the functional specification at a particular value of the program counter does not wire up all inputs, a special value will be introduced to indicate this.

6.2.5 Abstract dSCA Components

Data and Time: As with SCAs, the algorithm will process data from a set A at times t from a global clock $T = \{0, 1, 2, \ldots\}$.

Channels: The modules in a abstract dynamic Synchronous Concurrent Algorithm communicate via the channels of a network. Each channel has unit bandwidth, with respect to the carrier set A, and, each channel is uni-directionary. Thus, a channel may only transmit a single datum $a \in A$ at any one time, in one direction. Channels also have the properties that they may branch infinitely, but they may never merge. When a channel branches, the intent is that the datum being transmitted along the channel is "copied" and transmitted along each of the new branches.

Modules: Each module is capable of processing its inputs and producing one output. Processing occurs according to the functional specification. Consider module $m_i \in M$, where $i > 0$ and which has $n(i) + 1 \in \mathbb{N}$ input channels. The processing performed by this module for the functional specification is defined by the total function $F_i : A \times A^{n(i)+1} \rightarrow A$. The intent being that if $b_0, b_1, \ldots b_{n(i)} \in A$ were to arrive on $m_i's$ input channels, then m_i computes $F_i(b_0, b_1, \ldots, b_{n(i)})$, where F_i is made of Max_N component specifications $f_{i,0}, \ldots, f_{i,Max_N-1}$ and the appropriate component is selected based on the value of the program counter, b_0, and for $0 \leq pc \leq Max_N - 1$, $f_{i,pc}$ will select the appropriate arguments from $b_1, \ldots b_{n(i)} \in A$ for its calculation.

The notation that m_0 will be referred to as the program counter module will be adopted, and for ease of description this thesis will often refer to m_0 as m_{pc}. The program counter module is similarly specified to have Max_N component specifications, however each component will be the operation of adding 1 to its previous value modulo Max_N. Additionally, the program counter module will output 1 mod Max_N

at time $t = 0$ through to 0 at time $Max_N - 1$. This slightly counter-intuitive definition is necessary as the rest of the network will always consider the value of the program counter at $t - 1$. Thus, in the simplest case where the first state transition value function is computed, the current time will be $t = Max_N$ and it is intended to execute the 0^{th} component which requires $V_{pc}(t-1, a, x)$ to be 0, which is guaranteed by the above rule.

In an SCA the inputs to a module m_i, where $i > 0$ were denoted as $b_1, \ldots, b_{n_{(i)}}$. For an abstract dSCA the inputs will be denoted as $b_0, b_1, \ldots, b_{n_{(i)}}$ where $b_1, \ldots, b_{n_{(i)}}$ correspond to the inputs in an SCA and b_0 is reserved for the value of the program counter.

Source Modules (Input): Data is read into the network at sources, also known as input modules. Sources have no input channels and a single output channel, which as with other channels, may branch. A network with n sources will process n input streams, $a_1, \ldots, a_n \in [T \rightarrow A]$ with the convention that $a_i(t)$ is supplied as input by source $i \in I_{in}$ at time $t \in T$.

Sink Modules (Output): Data is read out of the network by sink modules; by definition, sink modules have a single input channel and no output channels. Data is read out of the network as values from A.

6.3 Formal Definition of Abstract dSCAs

Let N be a synchronous network over data set A with clock T. If N has $n > 0$ sources then the input to N is represented as the streams $a_1, \ldots, a_n \in [T \rightarrow A]$. It is also assumed that N has $k > 0$ modules, $m_0, \ldots m_k$ and that each module m_i , where $i \in \mathbb{N}_k$, has a maximum number, Max_N, of components in its functional specification, and that m_{pc} is the usual denotation for the programme counter module m_0. Further, it is assumed that for any vector $x_0, \ldots x_k \in A^k$, where $i \in \mathbb{N}_k$, x_i is a

tuple of values, $x_i = (x_{i,0}, \ldots, x_{i,Max_N-1}) \in A^{Max_N}$ will serve to specify the networks initial Max_N values, with the intention that module m_i holds the value $x_{i,pc}$ at time $t = pc$ and $0 \leq pc \leq Max_N - 1$. The initial value for module m_{pc} is specified as the tuple $x_{pc} = (1, 2, \ldots, Max_N - 1, 0)$. (the reason for the offset index of program counter values, rather than starting from 0, is that a module at time t will consider the program counter at time $t - 1$, e.g. at time $t = 36$, the value from the program counter at time $t = 35$ should be 0 otherwise the wrong component specification will be selected!).

Further, the **termination assumption** from the definition of SCA applies such that:

"*We assume that at each time $t \in T$ there is a value output from each module, and that this value can always by determined uniquely from the time t, the set of inputs a, and the set of initial values x*"([TT94])

The value held by module m_i at time t can be determined as required by using the termination assumption and the introduction of functions $V_0, \ldots, V_k$ where for $i \in \mathbb{N}_k$ the following is defined:

$$V_i : T \times [T \rightarrow A]^n \times A^k \rightarrow A$$

These functions are called the network's Value Functions.

The output of every module in the network can be determined by exploiting the termination assumption and the synchronous nature of the network. Every module's output is either specified initially or is specified in terms of the values held at previous time cycles. Value Functions can be defined in two phases, in a similar manner to the definition of SCAs given earlier. For modules $m_0, \ldots, m_k$, there are Max_N Initial

State phases (where $t = 0, 1, \ldots, Max_N - 1$), and a single State Transition phase:

$$\begin{aligned} &V_i\,(0, a, x)\,, \\ &\vdots \\ &V_i\,(Max_N - 1, a, x)\,, \\ &V_i\,(t + 1, a, x)\,. \end{aligned}$$

To provide the complete definition of Value Functions, it is necessary to introduce wiring functions, $\gamma_{pc\_val}(i, j)$ and $\beta_{pc\_val}(i, j)$, and delay functions, δ_{i,j,pc_val}.

Wiring Functions

Modules in a dSCA are wired together in different ways depending on the value of the program counter. Consider the network N which has $k > 0$ modules and $n > 0$ sources, then a module m_i, where $0 \leq i \leq k$, will have an associated function specification, F_i, that requires $n(i) + 1 > 0$ arguments. These arguments will arrive on the input channels for m_i and will be filled with data from the set A from either a source or an adjacent module. Two operations, $\gamma_{pc\_val}(i, j)$ and $\beta_{pc\_val}(i, j)$ are introduced that determine whether the j^{th} input for module m_i, at the program counter value *pc_val*, is from a source, an adjacent module or the undefined module, and what the index of that source or module is.

For a module m_i with inputs $j = 0, \ldots n(i)$, the operation $\gamma_{pc\_val}(i, j)$ is defined as follows:

$$\gamma_{pc\_val} : \mathbb{N}_k \times \mathbb{N} \rightarrow \{S, M, U\}$$

where S indicates a source module and M indicates a module, and U indicates the input is not connected/needed for the pc^{th} component functional specification of F_i. The value of $\gamma_{pc\_val}(i, 0)$ will always be M since it is always connected to the program counter.

We similarly define $\beta_{pc\_val}(i, j)$ as:

$$\beta_{pc\_val} : \mathbb{N}_k \times \mathbb{N} \rightarrow \mathbb{N}_k \cup \{\omega\}$$

where ω represents a special value for an unconnected connection. The value of $\beta_{pc\_val}(i,0)$ will always be pc, indicating that it is always connected to the program counter.

We require the following five conditions to always hold (where $i \in \mathbb{N}_k$, $j \in \{0,\ldots,n(i)\}$ and $pc\_val \in \{0,\ldots,Max_N - 1\}$:

1. $\beta_{pc\_val}(i,j) \downarrow \wedge\gamma_{pc\_va;}(i,j) \downarrow$ with the intended meaning that for all values of the counter $pc\_val = 0,\ldots Max_N - 1$ and inputs $j = 0,\ldots,n(i)$ of all modules $i = 1,\ldots,k$, the wiring functions $\beta_{pc\_val}(i,j)$ and $\gamma_{pc\_val}(i,j)$ are defined;

2. $\gamma_{pc\_val}(i,j) = S \Rightarrow 1 \leq \beta_{pc\_val}(i,j) \leq n$ with the intended meaning that if the j^{th} input channel of module m_i at counter value pc_val comes from a source, then the index of that source, provided by the β-wiring function, must be within the valid source indices $1,\ldots,n$;

3. $\gamma_{pc\_val}(i,j) = M \Rightarrow 0 \leq \beta_{pc\_val}(i,j) \leq k$ with the intended meaning that if the j^{th} input channel of module m_i at counter value pc_val comes from a module, then the index of that source, provided by the β-wiring function, must be within the valid module indices $0,\ldots,k$ (recall that the program counter module is m_0); and

4. $\gamma_{pc\_val}(i,j) = U \Leftrightarrow \beta_{pc}(i,j) = \omega$ with the intended meaning that if the j^{th} input channel of module m_i at counter value pc_val is not connected, then the value of the β-wiring function, must indicate the special non connected index ω.

5. $\gamma_{pc\_val}(i,0) = M \wedge \beta_{pc\_val}(i,0) = pc$ with the intended meaning that the $zero^{th}$ input to a module would always be from a module, whose index is the program counter (and this includes the program counter module).

A consequence of the first condition is the need to define values of $\gamma_{pc\_val}(i,j)$ and

$\beta_{pc\_val}(i,j)$ when the component of the functional specification under execution uses only a subset of the module's inputs.

Delay Functions

For each input channel j of module m_i at program counter value *pc_val* a delay is associated, δ_{i,j,pc_val}, that indicates from what time cycle the input was produced. It is defined as:

$$\delta_{i,j,pc\_val} : T \times [T \rightarrow A]^n \times A_{tup}^k \rightarrow T$$

The delay function in a dSCA is deliberately set so that it can take account of the current time, the current values of the input streams and the initial values of the network, as per the SCA definition. It is additionally indexed to reflect the different values it may take at different values of the program counter the dSCA module is currently executing at. To preclude the construction of predictive circuits, i.e. where the value of $\delta_{i,j,pc\_val}(t,a,x)$ is such that it looks forward in time, the temporal condition is introduced that for any time $t \in T$, inputs $a \in [T \rightarrow A]^n$, initial values $x \in A^{MAX_N \times K}$, and values of the counter $pc\_val \in \{0,1,2,\ldots Max_N - 1\}$ the value of delay must be less than t, i.e.: $\delta_{i,j,pc}(t,a,x) < t$.

For a cycle inconsistent dSCA the definition of the delay function is further bound so it cannot look at data calculated at times greater than $2 \times Max_N - 1$ clock cycles before the current time, i.e.:

$$t - (2 \times Max_N) < \delta_{i,j,pc\_val}(t,a,x) < t$$

and that it must never be allowed to refer to times less that 0:

$$\delta_{i,j,pc\_val}(t,a,x) > 0$$

The general rules of the delay function are that for $i \in \mathbb{N}_k$, $j \in \{0,\ldots,n(i)\}$ and $pc\_val \in \{0,\ldots,Max_N - 1\}$:

1. $\delta_{i,j,pc\_val}(t,a,x)\downarrow$ with the intended meaning that for all inputs $j = 0,\ldots,n(i)$ of all modules $i \in \mathbb{N}_k$, the delay function $\delta_{i,j,pc\_val}(t,a,x)$ is defined;

2. $t - 2 \times Max_N < \delta_{i,j,pc\_val}(t,a,x) < t$ with the intended meaning that for all inputs $j = 0,\ldots,n(i)$ of all modules $i \in \mathbb{N}_k$ and at all values of the program counter $pc\_val = 0,\ldots,Max_N$, the delay function $\delta_{i,j,pc\_val}(t,a,x)$ must be as a minimum the unit delay and as a maximum $2 \times Max_N$ (for a cycle consistent dSCA this constraint would be appropriately amended to cope with tuple of length Max_N);

and specifically relating to the program counter:

1. $\delta_{i,0,pc\_val}(t,a,x) = t - 1$ with the intended meaning that all program counter inputs to all modules are subject to a unit delay; and

2. $\delta_{pc,0,pc\_val}(t,a,x) = t - 1$ with the intended meaning that there is unit delay on the input of the program counter module to itself.

There will be times where the values of $\delta_{i,j,pc\_val}(t,a,x)$ are meaningless for the calculation; however, since the rules require a value to be provided for all modules at all values of the program counter the unit delay will be used in these cases.

Value Function Initial State Phase

The initial state phase for Value Functions defines the state of modules in N at times $t = 0,1,\ldots,Max_N - 1$ for modules $0,\ldots,k$. For modules $0,\ldots,k$ and program counter values $0,\ldots,Max_N - 1$ since $x_{i,pc}$, the pc^{th} element of the i^{th} vector of the set of initial values x, is intentionally the data value held by module m_i at time $t = pc$ then it is appropriate to define:

$$V_i(pc\_val,a,x) = x_{i,pc}$$

for $i \in \mathbb{N}_k$ and $pc\_val = 0, \ldots, Max_N - 1$.

For the program counter module, the values are specifically defined as:

$$(1, 2, \ldots, Max_N - 1, 0)$$

with the intended meaning that:

$$\begin{aligned} &V_{pc}(0, a, x) = 1 \\ &V_{pc}(1, a, x) = 2 \\ &\vdots \\ &V_{pc}(Max_N - 2, a, x) = Max_N - 1 \\ &V_{pc}(Max_N - 1, a, x) = 0 \end{aligned}$$

Value Functions State Transition Phase

For modules $0, \ldots, k$ the intention behind the module specification $F_i : A \times A^{n(i)+1} \rightarrow A$ is that if $b_0, b_1, \ldots, b_{n(i)}$ are the values selected by means of its delay functions $\delta_{i,0,pc\_val}, \delta_{i,1,pc\_val}, \ldots, \delta_{i,n(i),pc\_val}$ from past data along its input channels, then:

$$F_i\left(b_0, b_1, \ldots, b_{n(i)}\right)$$

is the value held at time t.

The definition of F_i for modules $1, \ldots, k$ consist of Max_N component specifications, one for each value of pc, and may include producing the "undefined" constant; where the intention is that the module performs no processing at that value of the counter and simply outputs, u. The introduction of u is necessary to ensure that each module has a value to output for each of its Max_N component functional specifications.

Max_N component specifications are defined for F_i as follows:

$$F_i\begin{pmatrix} b_0, \\ b_1, \\ \ldots, \\ b_{n(i)} \end{pmatrix} = \begin{cases} f_{i,1}\begin{pmatrix} b_{1,0}, \\ \ldots, \\ b_{n(i,0),0} \end{pmatrix} & \text{if } b_0 = 0 \\ f_{i,2}\begin{pmatrix} b_{1,1}, \\ \ldots, \\ b_{n(i,1),1} \end{pmatrix} & \text{if } b_0 = 1 \\ \vdots & \vdots \\ f_{i,Max_N}\begin{pmatrix} b_{1,Max_N-1}, \\ \ldots, \\ b_{n(i,Max_N-1),Max_N-1} \end{pmatrix} & \text{if } b_0 = Max_N - 1 \end{cases}$$

where $\left(b_{1,0},\ldots,b_{n(i,1),0},b_{2,1},\ldots,b_{n(i,Max_N)-1},Max_N\right) \in \left\{b_1,\ldots,b_{n(i)}\right\}$, i.e. arguments in the component specification are selected from arguments in the functional definition of the SCA.

For $j = 0,\ldots,n(i)$, the j^{th} input is either supplied by some source at some previous time, in which case, $b_j = a_q\left(\delta_{i,j,pc\_val}(t,a,x)\right)$, or it is supplied by some other module in the network at some previous time, $b_j = V_q\left(\delta_{i,j,pc\_val}(t,a,x),a,x\right)$. Accordingly, $V_i(t,a,x)$ is defined as:

$$V_i(t,a,x) = F_i\left(b_0,b_1,\ldots,b_{n(i)}\right)$$

where for $j = 0,\ldots,n(i)$:

$$b_j = \begin{cases} a_q\left(\delta_{i,j,b_0}(t,a,x)\right) & \text{if } \gamma_{b_0}(i,j) = S\wedge \\ & \quad \beta_{b_0}(i,j) = q \\ V_q\left(\delta_{i,j,b_0}(t,a,x),a,x\right) & \text{if } \gamma_{b_0}(i,j) = M\wedge \\ & \quad \beta_{b_0}(i,j) = q \\ u & \text{if } \gamma_{b_0}(i,j) = U\wedge \\ & \quad \beta_{b_0}(i,j) = \omega \end{cases}$$

it will always be the case that $b_0 = V_{pc}(t, a, x)$.

For the program counter module, m_{pc}, the definition of component specification, F_i, is always:

$$F_i(b_0) = \begin{cases} mod\,(add\,(b_0, 1)\,Max_N) & \text{if } b_0 = 0 \\ \vdots & \\ mod\,(add\,(b_0, 1)\,Max_N) & \text{if } b_0 = Max_N - 1 \end{cases}$$

Network Output

V_{out} is defined as the vector representing the output from network N. Consider that N has $m > 0$ sinks, then V_{out} would be constructed as $V_{out} = (V_{s_1}, \ldots, V_{s_m})$, where $s_1, \ldots, s_m \in \{1, \ldots, k\}$, i.e.:

$$V_{out} = (V_{s_1}, \ldots, V_{s_m})$$

V_{out} is not allowed to change with the values of the program counter to ensure that only one set of outputs is considered. However, it should be noted that if $Max_N > 1$ then a retiming is probably required on the values produced by V_{out} to ensure only relevant values are observed. The relevant values will be produced every Max_N clock cycles after an initial delay dependent upon the execution order chosen.

6.3.1 Defining Shape of an abstract dSCA

It is possible to implement the same algorithm on several dSCAs, differing only by the number of modules and values of Max_N. To distinguish between such dSCAs the pair $\nabla = (k, Max_N)$ is defined as the **Defining Shape of a dSCA**, where $k > 0$ is the number of modules in the dSCA network and Max_N is the number of component specifications each module in the dSCA network has.

An algorithm which has 20 separate functions to be implemented, can therefore be implemented on a dSCA where $\nabla = (20, 1)$ - the usual notion of an SCA - or some other valid combination, some examples of which are $\nabla = (10, 2)$, $\nabla = (5, 4)$, $\nabla =$

$(4,5)$, $\nabla = (1,20)$ - the last defining shape perhaps representing a single processor machine.

It is possible for a dSCA to be defined where ∇ can support more functions than are available, e.g. in the example we are using the value could be given as $\nabla = (5,5)$. The only restriction placed on the defining shape is that there must be a sufficient number of modules/computations allowed (value of Max_N) to handle all functions in the computation. To determine this, the defining size of the dSCA is introduced.

6.3.2 Defining Size of an abstract dSCA

The **Defining Size of a dSCA** with a defining shape of $\nabla = (k, Max_N)$ is defined to be $\Delta = (k \times Max_N)$. The defining size provides a metric to understand if a particular algorithm will fit onto a particular dSCA defining shape. The example used in the defining shape section, section 6.2.1, with 20 operations would clearly need a dSCA where $\Delta \geq 20$ for it to be implemented on a dSCA.

6.4 Correctness

By inspection it can be seen that a dSCA is simply an SCA with some syntactic sugar around the modules functional definition. Therefore the same correctness approaches used for SCAs can be applied to abstract dSCAs.

Care must be taken to ensure that the same type of consistency is applied to all modules, i.e. an abstract dSCA should not have modules where some are cycle consistent and some are cycle inconsistent.

6.5 Algebraic Specification of abstract dSCAs

Since an abstract dSCA is really an SCA with some syntactic sugar, it can be specified algebraically in the same way as an SCA. An example algebraic specification of an

abstract dSCA is given in Appendix C.

As with the SCA example given in chapter 5, this form can be readily translated into a format suitable for use within one of the algebraic tools available by collapsing the additional operation and equation definitions in the notation, as shown for SCAs.

6.6 Concluding Comments

The introduction of abstract dSCAs is fundamental to this thesis, as it supports the notion of transforming the defining shape of a dSCA, as will be demonstrated in Chapter 11. The author has specifically ensured that abstract dSCAs are just simple syntactic extensions of SCAs in order to provide a solid foundation for mathematical analysis. This is known since it is possible to construct an abstract dSCA using just the syntax of the normal definition of SCAs. In doing so, the indexing of the wiring and delay functions would have to be removed and codified into the definition of the function.

6.7 Sources

The work in this chapter on extending SCAs syntactically to cover the requirements for dynamic SCAs is all my own work except for that that deals with the introduction of non-unit delay SCAs which is based on the work of Hobley; however this thesis enhances the understanding of non-unit delay SCAs when used in a hierarchy, in particular the identification of cycle consistency and the need to provide equations to represent the Initial State for times $t = 0, 1, \ldots, Max_N - 1$. The initial work investigated SCAs and what have now become concrete dSCAs - the author is grateful to his supervisor, Dr. N. Harman, who suggested introducing the abstract dSCA concept enabling the transformation of SCAs to be studied in a cleaner manner.

Chapter 7

Concrete Dynamic Synchronous Concurrent Algorithms

Concrete dynamic SCA

To approximate a model of physical hardware implementation with memory and a program counter.

7.1 Introduction

The previous chapter introduced abstract dSCAs allowing the limitations of SCAs, in the context of this thesis, to be addressed. Data was passed around the network as single datum from an underlying algebra A and the delay function was responsible for selecting the correct data from previous time cycles, $t-1,\ldots,t-(2\times Max_N-1)$ (or $t-1,\ldots,t-Max_N-1$ for cycle consistent abstract dSCAs). Current technology does not support a machine with these temporal look-ups without the look-ups being encoded in a more concrete manner. Concrete dSCAs are introduced to support the encoding of results by storing these results in a finite length tuple per module.

As discussed in Chapter 6 a cycle inconsistent abstract dSCA can be either partially or completely cycle inconsistent. It was also mentioned that for abstract dSCAs

this distinction was merely a classification but that there were implications for concrete dSCAs. These implications relate to the size of the tuple needed for results.

Hobley, [Hob90], showed how non-unit delay SCAs may be represented as unit delay SCAs by the introduction of buffered channels or internal storage. Whereas Hobley's work considers these buffers/storage as shift registers, we extend this and generalise with the introduction of tuple management functions allowing, if we wish, buffers to act as indexed arrays. Additionally, the concrete dSCAs that this thesis introduces implement the other attributes of dynamic SCAs given in the previous chapter.

7.2 Informal Definition of Concrete dSCAs

Recall that modules within an abstract dSCA network communicate via channels, and each channels is of unit bandwidth with respect to the underlying algebra A, and uni-directional. Concrete dSCAs, like abstract dSCAs, may be cycle consistent or inconsistent, and this thesis will consider both types of concrete dSCAs since both have differing requirements for storage.

In a concrete dSCA the size of the storage depends on the type of concrete dSCA under examination:

- Cycle consistent dSCA needs a storage size of only Max_N.
- A totally cycle inconsistent dSCA needs a storage size of $2 \times Max_N$.
- All other cycle inconsistent dSCA needs, as a minimum, a storage size of Max_N plus a storage element for each result that is not cycle consistent.

Thus in the worst case it is necessary to store up to the last $2 \times Max_N$ values calculated by the module so they are available to other modules.

This storage will be implemented in concrete dSCAs using tuples, thus the carrier set A_{tup} is constructed in such a way so that it includes the algebra A and tuples made from A of the necessary length. Using A_{tup} it is now possible to maintain the unit bandwidth notion of SCAs.

There are many conceivable ways of placing data into the tuple and subsequently retrieving them. The complexity of these approaches is related to the type of cycle consistency under consideration. For a cycle consistent and totally cycle inconsistent concrete dSCA these *tuple management* functions can be relatively simple, for a non totally inconsistent concrete dSCA the functions are more complex requiring knowledge of which results are cycle consistent and those which are not. For reasons of simplicity this thesis will consider only cycle consistent and totally cycle inconsistent concrete dSCAs. Two of the more interesting tuple management approaches are:

1. **Queue**. This is the most obvious implementation; and
2. **Indexed array**. This would closely map to a von Neuman architecture.

The tuple management operations will need to:

1. Update the data in the tuple, which consists of:
   (a) inserting newly calculated data into the tuple at the correct position for later extraction; and
   (b) deleting old data from the tuple; and
2. Extract the required data

Updating the tuple values will be managed by the tuple update operation, Υ, which for a cycle inconsistent concrete dSCA will be given as:

$$\Upsilon : A \times A_1 \times \cdots \times A_{2\times Max_N} \times A \rightarrow A_1 \times \cdots \times A_{2\times Max_N}$$

For a cycle consistent concrete dSCA the definition of the tuple update operation is simplified to:

$$\Upsilon : A \times A_1 \times \cdots \times A_{Max_N} \times A \rightarrow A_1 \times \cdots \times A_{Max_N}$$

The intention behind Υ is that the first argument will be program counter value, the next $2 \times Max_N$ arguments are the tuple values from this module at the previous time, and the final argument is the result just calculated.

Selection for a cycle inconsistent dSCA is made by applying a projection operation, $\Pi^{2\times Max_N}_{d_{i,j,pc}}$, on the tuple to select the value at the index $d_{i,j,pc}$, for the j^{th} input of module i at program counter value pc, and is given as:

$$\Pi^{2\times Max_N}_{d_{i,j,pc}} : A_1 \times \cdots \times A_{2\times Max_N} \rightarrow A$$

Again, a cycle consistent concrete dSCA will define the projection operation over the simplified tuple output as:

$$\Pi^{Max_N}_{d_{i,j,pc}} : A_1 \times \cdots \times A_{Max_N} \rightarrow A$$

The two chosen tuple management techniques are now discussed in more detail.

7.2.1 Tuple Management : Queue

Updating Data:

Managing the tuple as a first-in first-out queue requires new data, b, to be added to the left hand side of the tuple and the removal of the rightmost data.

Cycle Inconsistent Tuple Management Definitions

The tuple management definitions for a cycle inconsistent concrete dSCA are defined simultaneously. Where the value of the program counter is less than $Max_N - 1$, then the values in the tuple at n get shifted right to $n + 1$, for $0 \leq n \leq Max_N - 1$,

and the new result added to the tuple at position 0:

$$\Upsilon\left(\begin{array}{l} pc\_val, \\ \left(\begin{array}{l} a_0, \ldots, a_{Max_{N-1}}, \\ a_{Max_N}, \ldots, a_{2\times Max_N-1} \end{array}\right) \\ , b \end{array}\right) = \left(\begin{array}{l} b, a_0, \ldots, a_{Max_N-2}, \\ a_{Max_N-1}, \ldots, a_{(2\times Max_N)-2} \end{array}\right)$$

if the value of the program counter equals $Max_N - 1$ then the process above is performed and then the whole first half of the tuple is copied into the second half:

$$\Upsilon\left(\begin{array}{l} Max_N - 1, \\ \left(\begin{array}{l} a_0, \ldots, a_{Max_N-1}, \\ a_{Max_N}, \ldots, a_{2\times Max_N-1} \end{array}\right) \\ , b \end{array}\right) = copy\left(\begin{array}{l} b, a_0, \ldots, a_{Max_N-2}, \\ a_{Max_N-1}, \ldots, a_{(2\times Max_N)-2} \end{array}\right)$$

$$= \left(\begin{array}{l} b, a_0, \ldots, a_{Max_N-2}, \\ b, a_0, \ldots, a_{Max_N-2} \end{array}\right)$$

The value of programme counter is not used in the queue tuple management operations. It remains in the definition for ease of clarity across models.

Cycle Consistent Tuple Management Definitions

For a cycle consistent concrete dSCA there is no need to copy the data to higher levels of the tuple at $pc = Max_N - 1$ so the definition of Υ is simplified as:

$$\Upsilon(pc\_val, a_0, \ldots, a_{Max_N-1}, b) = (b, a_0, \ldots, a_{Max_N-2})$$

Selecting Data:

Cycle Inconsistent Tuple Management Definitions

The value of d_{i,j,pc_val} in the projection function, $\Pi^{2\times Max_N}_{d_{i,j,pc\_val}}$, is directly proportional to the time that the result was calculated. Consider the output of module m_i at time $t-1 \in T$, which is what other modules will be restricted to observe, it will consist of

a tuple of results as follows:

$$\begin{pmatrix} V_h(t-1,a,x), \\ V_h(t-2,a,x), \\ \ldots, \\ V_h(t-Max_N,a,x) \\ V_h(t-(Max_N+1),a,x), \\ \ldots, \\ V_h(t-(2\times Max_N),a,x) \end{pmatrix}$$

If it is intended to select the result calculated at time t-4 then it will have been shifted $(t-1-(t-4)$ indexes to the right of the start of the tuple, i.e. it will be at index 3 (assuming that Max_N is of course greater than 4) - note that indexing of tuple elements starts at 0.

Cycle Consistent Tuple Management Definitions

The cycle consistent concrete dSCA will have the same definition, but is restricted to values up to Max_N time cycles ago.

7.2.2 Tuple Management : Indexed Array

Updating Data:

Managing the tuple as an indexed array ensures that new data is entered into to a predetermined position in the array, whilst overwriting any existing data held in that position. Since the program counter value is available to modules and uniquely identifies values in the range of $0,\ldots,Max_N-1$ this value is chosen to indicate where in the tuple results will be placed.

Cycle Inconsistent Tuple Management Definitions

In a cycle inconsistent concrete dSCA if the value of the program counter is less than Max_N-1 then the newly calculated value is entered into the array at position

indicated by the program counter as follows:

$$\Upsilon\left(\begin{array}{l} pc\_val, \\ \left(\begin{array}{l} a_0, \ldots, a_{Max_{N-1}}, \\ a_{Max_N}, \ldots, a_{2\times Max_N-1} \end{array}\right) \\ , b \end{array}\right) = \left(\begin{array}{l} a_0, \ldots, a_{pc\_val-1}, b, a_{pc\_val+1}, \ldots, a_{Max_N-1}, \\ a_{Max_N}, \ldots, a_{(2\times Max_N)-1} \end{array}\right)$$

When the program counter equals $Max_N - 1$ then the newly calculated vale is entered at index $Max_N - 1$ and subsequently the values with indexes $0, \ldots, Max_N - 1$ are copied to indexes $Max_N, \ldots, 2 \times Max_N - 1$, as:

$$\Upsilon\left(\begin{array}{l} Max_N - 1, \\ \left(\begin{array}{l} a_1, \ldots, a_{Max_N}, \\ a_{Max_N+11}, \ldots, a_{2\times Max_N} \end{array}\right) \\ , b \end{array}\right) = copy\left(\begin{array}{l} a_0, \ldots, a_{Max_N-2}, b, \\ a_{Max_N}, \ldots, a_{(2\times Max_N)-1} \end{array}\right)$$

$$= \left(\begin{array}{l} a_0, \ldots, a_{Max_N-2}, b, \\ a_0, \ldots, a_{Max_N-2}, b \end{array}\right)$$

Cycle Consistent Tuple Management Definitions

For the cycle consistent concrete dSCA the definition would be:

$$\Upsilon(pc\_val, a_0, \ldots, a_{Max_N-1}, b) = (a_0, \ldots, a_{pc\_val-1}, b, a_{pc\_val+1}, \ldots a_{Max_N-1})$$

Selecting Data

Cycle Inconsistent Tuple Management Definitions

Retrieval of the correct datum from the tuple, i.e. the identification of the correct value for d_{i,j,pc_val} in the projection function, $\Pi^{2\times Max_N}_{d_{i,j,pc\_val}}$, relies on knowledge of the program counter value when the result was calculated, and selecting the appropriate index value.

Cycle Consistent Tuple Management Definitions

The cycle consistent concrete dSCA has a similar definition but is limited to retrieval from a tuple of length Max_N.

7.2.3 Cycle Consistency and Execution Order

Concrete dSCAs have the same properties as abstract dSCAs regarding execution order and cycle consistency, and has been shown, a cycle consistent dSCA requires a tuple of only Max_N length, whereas a cycle inconsistent concrete dSCA requires one with a length of $2 \times Max_N$.

7.2.4 Concrete dSCA Components

Data and Time: As with SCAs, a concrete dSCA will process data from a set A which is augmented with tuples of the length Max_N to form the set A_{tup}, at times t from a global clock $T = \{0, 1, 2, \ldots\}$.

Channels: The modules in a concrete dSCA communicate via the channels of a network. Each channel has unit bandwidth, with respect to the carrier set A_{tup} and each channel is uni-directional. A channel may only transmit a single datum $a \in A_{tup}$ at any one time, in one direction, where the tuples will be of length tl between all modules except from the program counter, which is of length 1, i.e. a single datum. Channels also have the properties that they may branch infinitely, but they may never merge. When a channel branches, the intent is that the datum being transmitted along the channel is "copied" and transmitted along each of the new branches.

Modules: Each module is capable of processing its inputs and producing one output, which in all cases apart from the program counter, will be a tuple. Take a module $m_i \in M$, where $i > 0$ and which has $n(i) + 2 \in \mathbb{N}$ input channels; then the first input channel is always from the program counter, the second from the module itself, and the remaining inputs from sources or other modules in the network. the processing

performed by this module is defined as the total function:

$$F_i : A_{tup} \times A_{tup} \times A_{tup}^{n(i)} \rightarrow A_{tup}.$$

The intent being that if values $b_0, b_1, b_2, \ldots b_{n(i)+1} \in A_{tup}$ were to arrive on $m_i's$ input channels, then m_i computes $F_i(b_0, b_1, b_2, \ldots, b_{n(i)+1})$, where the F_i is made from Max_N component specifications $f_{i,0}, \ldots, f_{i,Max_N-1}$ and the appropriate component is selected by the value of the program counter, b_0, and for $0 \leq pc\_val \leq Max_N - 1$, f_{i,pc_val} will select the appropriate arguments from $b_0, \ldots b_{n(i)+1} \in A$ for its calculation. Each component specification will follow the following form:

$$f_{i,pc}(b_0, b_1, b_2, \ldots, b_{n(i)+1}) = \Upsilon\left(b_0, b_1, f_{i,pc}^{op}\left(b_0, \Pi_{d_{i,2,pc}}^{tl}(b_2), \ldots, \Pi_{d_{i,p(i)-1,pc}}^{tl}\left(b_{n(i)+1}\right)\right)\right)$$

where Υ is the chosen tuple management operation, Π the associated projection operation, $f_{i,pc}^{op}$ is the actual calculation performed by the module for that value of the program counter and tl is either $2 \times Max_N$ if the concrete dSCA is totally cycle inconsistent, or Max_N if it is cycle consistent.

Module m_0 is defined as the program counter module, and the notion that this module is referred to as m_{pc} is adopted. The program counter module is always specified as:

$$V_{pc}(t+1, a, x) = \begin{cases} mod\left(add\left(V_{pc}(t, a, x), 1\right) Max_N\right) & \text{if } V_{pc}(t, a, x) = 0 \\ \vdots & \\ mod\left(add\left(V_{pc}(t, a, x), 1\right) Max_N\right) & \text{if } V_{pc}(t, a, x) = Max_N - 1 \end{cases}$$

Source Modules (Input): Data is read into the network at sources, also known as input modules. Sources have no input channels and a single output channel, which as with other channels, may branch. A network with n sources will process n input streams, $a_1, \ldots, a_n \in [T \rightarrow A]$ (note that tuples are not allowed for inputs) with the convention that $a_i(t)$ is supplied as input by source $i \in I_{in}$ at time $t \in T$.

Sink Modules (Output): Data is read out of the network by sink modules; by definition, sink modules have a single input channel and no output channels. Since the modules will be producing tuple outputs it is necessary to select the required value from the tuple output to obtain a sensible result.

7.3 Formal Definition of Concrete dSCAs

Let N be a synchronous network over data set A_{tup}, which is the data set A augmented with tuples of length tl (where tl is Max_N if the dSCA is cycle consistent or $2 \times Max_N$ otherwise), with clock T. If N has $n > 0$ sources then the input to N is represented as the streams $a_1, \ldots, a_n \in [T \rightarrow A]$. It is assumed that N has $k > 0$ modules, $m_0, \ldots m_k$, and that each module m_i, where $i \in \mathbb{N}$, has a maximum number, Max_N, of components in its functional specification, and that module m_0 is a special program counter module usually denoted as m_{pc}.

It is further assumed that for any vector $x_0, \ldots x_k \in A_{tup}^{k \times k}$, where for $i \in \mathbb{N}_k$, a vector of values, x_i, is defined for $1 \leq i \leq k$ such that $x_i = (x_{i,0}, \ldots, x_{i,Max_N-1}) \in A_{tup}^k$. That is to say, that for each module m_i, where $1 \leq i \leq k$, there are Max_N initial states defined and that for each value of the program counter, $0 \leq l \leq tl - 1$, the initial value tuple is of the form $(x_{i,l,0}, \ldots, x_{i,l,tl-1})$, and that this will serve to specify the networks initial values. The intention is that module m_i holds the value $x_{i,t}$ at time $t \in 0, l, Max_N - 1$. Additionally, for the program counter module the initial values $x_0 = (x_{0,0}, \ldots, x_{0,Max_N-1})$ are defined where x_{0,pc_val} is a single value from A

Further, the **termination assumption** from the original definition of SCA and abstract dSCAs is kept such that:

> "We assume that at each time $t \in T$ there is a value output from each module, and that this value can always by determined uniquely from the

time t, the set of inputs a, and the set of initial values x." ([TT94])

The value held by module m_i, at time t, can be determined, as required by the termination assumption, by introducing functions $V_0, \ldots, V_k$ where for $i = 0, \ldots, k$ the following is defined:

$$V_i : T \times [T \rightarrow A]^n \times A_{tup}^k \rightarrow A_{tup}$$

These functions are known as the network's Value Functions. By exploiting the termination assumption and the synchronous nature of the network, the output of every module in the network is either specified initially, or is specified in terms of the values held at previous time cycles. Value Functions for a module can therefore be defined in distinct components - for the Initial State and one for the State Transition , i.e. for each module m_i where $i \in \mathbb{N}_k$ the following can be defined:

$$\begin{aligned} &V_i\,(0, a, x)\,, \\ &\vdots \\ &V_i\,(Max_N - 1, a, x)\,, \\ &V_i\,(t + 1, a, x)\,. \end{aligned}$$

The nature of the network is such that it is not until $t = Max_N - 1$ that the initial value is meaningful, i.e. produces a tuple that is filled with all the correct values, thus it is only the value at $t = Max_N - 1$ that is of interest; values before that could be filled with undefined or some other chosen values. However, to ease correctness proofs it is useful to have the state at time $t = 0$ being a well defined and known state.

An alternative approach would be to define one initial state phase for the value function, at time $t = 0$ which is equivalent to the previous approach at $t = Max_N$ and then begin processing from that point.

This extension will use the first approach to allow easier comparison to the abstract dSCA which the concrete may be derived from.

To provide the complete definition of Value Functions, it is necessary to consider the following

- Wiring Functions, $\gamma_{pc}(i,j)$ and $\beta_{pc}(i,j)$; and
- Delay Function, $\delta_{i,j}(t,a,x)$.

Wiring Functions

Just like modules in an abstract dSCA, the modules in a concrete dSCA are wired together in different ways depending on the value of the program counter. Consider the network N which has $k > 0$ modules and $n > 0$ sources, then a module m_i ($i \in \mathbb{N}_k$) will have an associated function specification, F_i, that requires $n(i)+2 > 0$ arguments. These arguments will arrive on $m_i's$ input channels and will be filled with data from the set A_{tup} from either a source, an adjacent module or the program counter module. Two operations, $\gamma_{pc\_val}(i,j)$ and $\beta_{pc\_val}(i,j)$ are introduced that determine for module m_i's j^{th} input whether it is from a source or an adjacent module, and what the index of that source or module is at the pc_val^{th} cycle.

For a module m_i with inputs $j = 0, \ldots n(i)+2$, the operation $\gamma_{pc\_val}(i,j)$ is defined as follows:

$$\gamma_{pc\_val} : \mathbb{N}_k \times \mathbb{N} \rightarrow \{S, M, U\}$$

where S indicates a source module and M indicates a module, and U indicates the input is not connected/needed for the pc_val^{th} component functional specification of F_i.

$\beta_{pc\_val}(i,j)$ is similarly defined as:

$$\beta : \mathbb{N}_k \times \mathbb{N} \rightarrow N_k \cup \{\omega\}$$

where ω represents a special value for no connection.

The following six conditions always hold where $i \in \mathbb{N}_k$, $pc\_val \in 0, \ldots, Max_N - 1$ and $j \in 0, \ldots, n(i)+1$:

1. $\beta_{pc\_val}(i,j) \downarrow \wedge \gamma_{pc}(i,j) \downarrow$ with the intended meaning that for all values of the program counter and for all inputs of all modules the wiring functions $\beta_{pc\_val}(i,j)$ and $\gamma_{pc\_val}(i,j)$ are defined;

2. $\gamma_{pc\_val}(i,j) = S \Rightarrow 1 \leq \beta_{pc\_val}(i,j) \leq n$ with the intended meaning that if the j^{th} input channel of module m_i at counter value *pc_val* comes from a source, then the index of that source, provided by the β-wiring function, must be within the valid source indices $1, \ldots, n$;

3. $\gamma_{pc\_val}(i,j) = M \Rightarrow 1 \leq \beta_{pc\_val}(i,j) \leq k$ with the intended meaning that if the j^{th} input channel of module m_i at counter value *pc* comes from a module, then the index of that source, provided by the β-wiring function, must be within the valid module indices $0, \ldots, k$;

4. $\gamma_{pc\_val}(i,j) = U \Leftrightarrow \beta_{pc\_val}(i,j) = \omega$ with the intended meaning that if the j^{th} input channel of module m_i at counter value *pc_val* is not connected, then the value of the β-wiring function, must indicate the special non connected value ω;

5. $\gamma_{pc\_val}(i,0) = pc \wedge \beta_{pc\_val}(i,0) = M$ with the intended meaning that the $zero^{th}$ input of each module m_i is wired to the program counter module; and

and for $i \in 1, \ldots, k$

1. $\gamma_{pc\_val}(i,1) = i \wedge \beta_{pc\_val}(i,1) = M$ with the intended meaning that the 1^{st} input of each module m_i, except the program counter module, is wired to that module (recall that the program counter only has only input, and this case is covered by the previous condition).

A consequence of the first condition is the need to define values of $\gamma_{pc}(i,j)$ and $\beta_{pc}(i,j)$ even for when the component of the functional specification under execution uses a subset of the modules inputs - hence the introduction of the values U and ω.

Delay Functions

For each input channel j of module m_i, where $i \in \mathbb{N}_k$, a delay is associated, $\delta_{i,j}(t, a, x)$, that indicates the delay between the time cycle the input was produced and the current time. It is defined as:

$$\delta_{i,j} : T \times [T \rightarrow A]^n \times A^k_{tup} \rightarrow T$$

To preclude the construction of predictive circuits - where the value of $\delta_{i,j,pc}(t, a, x)$ is such that it looks forward in time, the temporal condition is introduced that for any time $t \in T$, inputs $a \in [T \rightarrow A]^n$, initial values $x \in A^k_{tup}$, the value of delay must mean looking at times less that t:

$$\delta_{i,j,pc}(t, a, x) < t$$

Since the purpose of the introduction of concrete dSCAs is to remove the necessity to look back greater than the previous time unit, the delay function is restricted to be the unit delay introduced in the original SCA definition. Recall that values calculated at times $t-1, t-2, \ldots, t-tl$ are preserved in the tuple produced at time $t-1$. The suffix of *pc* introduced in abstract dSCAs is no longer needed as the value of the delay function is no longer dependant upon values of the program counter.

The restrictions placed on $\delta_{i,j}$ for concrete dSCAs, where $i \in \mathbb{N}_k$ and $j \in 0, \ldots, n(i)+1$, are:

1. $\delta_{i,j}(t, a, x) \downarrow$ with the intended meaning that for all inputs of all modules the delay function $\delta_{i,j}(t, a, x)$ is defined; and

2. $\delta_{i,j}(t, a, x) = t-1)$ with the intended meaning that for all inputs of all modules the delay function $\delta_{i,j}(t, a, x)$ is the unit delay.

Initial State

The Initial State defines the state of modules in then network N. For module m_i, where $i \in \mathbb{N}_k$, the Initial State is defined for times $t = 0, 1, \ldots, Max_N - 1$.

Since $x_{i,pc}$, the pc^{th} element of the i^{th} vector of the set of initial values x, is intentionally the tuple of data values held by module m_i at time $t = pc\_val$, for $0 \leq pc\_val \leq Max_N - 1$, and pc_val is the value of the counter, then it is appropriate to define:

$$V_i(pc\_val, a, x) = x_{i,pc\_val}$$

for $i = 0, \ldots, k$ and $pc\_val = 0, \ldots, Max_N - 1$.

The set of initial values is constructed in such a way that at time $t = Max_N - 1$ the output tuple for module m_i is loaded with all necessary initial values in the order specified by the tuple management scheme under use. It is permissable for the values of the initial state prior to $t = Max_N - 1$ to be undefined since they do not partake in any of the computation.

For module m_0 the program counter (also written as m_{pc}), the value of x_0 is always defined as:

$$x_0 = ((1), (2), \ldots, (Max_N - 1), (0))$$

State Transition

For module m_i, where $i \in \mathbb{N}_k$, the intention behind the module specification:

$$F_i : A_{tup} \times A_{tup} \times A_{tup}^{n(i)} \rightarrow A_{tup}$$

is that if $b_0, \ldots, b_{n(i)+1}$ are the values selected by means of its delay functions $\delta_{i,0}, \ldots, \delta_{i,n(i)+1}$ from past data along its input channels, then $F_i\left(b_0, b_1, b_2, \ldots, b_{n(i)+1}\right)$ is the value held at time t. The definition of F_i consists of Max_N component specifications, one for each value of the programme counter, and may include the "undefined" operation u - where the intention is that the module performs no processing at that value of the counter and simply outputs, u (the introduction of the undefined value is required to ensure each module outputs a value for each of its Max_N component

functional specifications). Max_N component specifications are defined for F_i as follows:

$$F_i\begin{pmatrix} b_0, \\ b_1, \\ \ldots, \\ b_{n(i)+1} \end{pmatrix} = \begin{cases} f_{i,0}\begin{pmatrix} b_{0,0}, \\ b_{1,0}, \\ b_{2,0}, \\ \ldots, \\ b_{n(i,0),0} \end{pmatrix} & \text{if } b_0 = 0 \\ f_{i,1}\begin{pmatrix} b_{0,1}, \\ b_{1,1}, \\ b_{2,1}, \\ \ldots, \\ b_{n(i,1),1} \end{pmatrix} & \text{if } b_0 = 1 \\ \vdots & \vdots \\ f_{i,Max_N-1}\begin{pmatrix} b_{0,Max_N-1}, \\ b_{1,Max_N-1}, \\ b_{2,Max_N-1}, \\ \ldots, \\ b_{n(i,Max_N-1),Max_N-1} \end{pmatrix} & \text{if } b_0 = Max_N - 1 \end{cases}$$

where values $\left(b_{0,0}, b_{1,0}, \ldots, b_{n(i,1),0}, b_{2,1}, \ldots, b_{n(i,Max_N)-1}, Max_N\right) \in \left\{b_0, \ldots, b_{n(i)+1}\right\}$, i.e. arguments in the component specification are selected from appropriate arguments in the functional definition of the SCA. Each f_{i,pc_val}, where $i \in \mathbb{N}_k$, will be defined as:

$$f_{i,pc\_val}\begin{pmatrix} b_0, \\ b_1, \\ b_2, \\ \ldots, \\ b_{n(i,pc\_val)} \end{pmatrix} = \Upsilon\left(b_0, b_1, f^{op}_{i,pc\_val}\begin{pmatrix} \Pi^{tl}_{d_{i,2,pc\_val}}\left(b_2\right), \\ \ldots, \\ \Pi^{tl}_{d_{i,n(i)-1,pc\_val}}\left(b_{n(i,pc\_val)}\right) \end{pmatrix}\right)$$

where Υ is the appropriate tuple management operation for the tuple length, the projection functions $\Pi^{tl}_{d_{i,2,pc\_val}}, \ldots, \Pi^{tl}_{d_{i,n(i),pc\_val}}$, select the appropriate data from the tuples arriving on $m_i's$ input channels, and f^{op}_{i,pc_val} is the actual calculation performed.

However, for $j = 0, \ldots, n(i)+1$, the j^{th} input is either supplied by some source at some previous time, in which case, $b_j = a_u(\delta_{i,j}(t, a, x))$, or it is supplied by some other

module in the network at some previous time, in which case $b_j = V_u(\delta_{i,j}(t,a,x),a,x)$. Accordingly, $V_i(t,a,x)$ is defined for $j = 2,\ldots,n(i)$ as:

$$V_i(t,a,x) = \begin{cases} \Upsilon\left(b_0, b_1, f_{i,0}\left(b_2, b_3, \ldots, b_{n(i,0)}\right)\right) & \text{if } b_0 = 0 \\ \vdots & \\ \Upsilon\left(b_0, b_1, f_{i,Max_N-1}\left(b_2, b_3, \ldots, b_{n(i,Max_N-1)}\right)\right) & \text{if } b_0 = Max_N - 1 \end{cases}$$

where for $j = 2,\ldots,r(i)$:

$$b_{j,pc\_val} = \begin{cases} a_q\left(\delta_{i,j}\left(t,a,x\right)\right) & \text{if } \gamma_{b_0}(i,j) = S\wedge \\ & \beta_{b_0}(i,j) = q \\ \Pi^{tl}_{d_{i,j,pc\_val}}\left(V_q\left(\delta_{i,j}\left(t,a,x\right),a,x\right)\right) & \text{if } \gamma_{b_0}(i,j) = M\wedge \\ & \beta_{b_0}(i,j) = q \end{cases}$$

For j=0, the definition of concrete dSCA dictates that the 0^{th} input comes from the program counter module, m_{pc}. Thus it is appropriate to define:

$$\begin{aligned} b_0 &= V_{pc}(\delta_{i,0}(t,a,x),a,x) \\ &= V_{pc}(t,a,x). \end{aligned}$$

The 1^{st} input to a module comes from itself, thus for j=1 it is appropriate to define:

$$\begin{aligned} b_1 &= V_i(\delta_{i,1}(t,a,x),a,x) \\ &= V_i(t,a,x) \end{aligned}$$

For the program counter module, m_{pc}, the following is defined:

$$V_{pc}(t+1,a,x) = \begin{cases} mod\left(add\left(V_{pc}(t,a,x),1\right),Max_N\right) & \text{if } V_{pc}(t,a,x) = 0 \\ \vdots & \\ mod\left(add\left(V_{pc}(t,a,x),1\right),Max_N\right) & \text{if } V_{pc}(t,a,x) = Max_N - 1 \end{cases}$$

Network Output

V_{out} is defined as the vector representing the output from network N. Consider that N has $m > 0$ sinks, then V_{out} would be constructed as $V_{out} = (V_{s_1},\ldots,V_{s_m})$, where $s_1,\ldots,s_m \in \{1,\ldots,k\}$, i.e.:

$$V_{out} = (V_{s_1},\ldots,V_{s_m}).$$

Since V_{out} for a concrete dSCA is a vector of tuples, each tuple containing the last Max_N results, any comparison of dSCA to other models requires the extraction of the necessary values from the tuples for comparison. Such a mapping will be dependant upon the tuple management scheme used, and whether the concrete dSCA is cycle consistent or not.

7.3.1 Defining Shape of an concrete dSCA

As with abstract dSCAs, it is possible to implement the same algorithm on several concrete dSCAs, differing only by the number of modules and values of Max_N. To distinguish between such concrete dSCAs the pair $\nabla = (k, Max_N)$ is defined as the **Defining Shape of a concrete dSCA**, as for abstract dSCAs, where $k > 0$ is the number of modules in the dSCA network and Max_N is the number of component specifications each module in the dSCA network has.

7.3.2 Defining Size of an concrete dSCA

The **Defining Size of a concrete dSCA** with a defining shape of $\nabla = (k, Max_N)$ is defined to be $\Delta = (k \times Max_N)$.

7.4 Correctness

By inspection it can be seen that concrete dSCAs are simple syntactic extension to the original SCA model as such syntactic correctness of a concrete dSCA can be shown by applying the techniques given for the original SCA.

7.5 Algebraic Specification

Concrete dSCAs can be specified algebraically using a similar format to that used for SCAs and abstract dSCAs. An example of the Generalised Railroad Crossing

Problem, described later in this thesis, represented as a concrete dSCA specification can be seen in Appendix E.

7.6 Concluding Comments

Concrete dSCA are the final (syntactic) extension to the SCA model introduced, and are necessary to remove the abstract concepts of an abstract dSCA. Hobley showed that non unit delay SCAs could be represented as SCAs given a suitable mechanism for the buffering of values, what we have done is again demonstrate this, but in a cleaner and more general way, as well as implementing the dynamic rewiring that we require.

7.7 Sources

The work on Concrete dSCAs is all my own work. However, credit is given to Hobley for the initial discussion on relaxing unit-delay requirements of SCAs and suggestions on how these non-unit delays may be implemented as unit delay SCAs.

Chapter 8

Generalised Railroad Crossing Problem Represented as various SCAs

8.1 Introduction

The GRCP case study can be represented in all of the SCA forms so far presented. In this chapter a solution to the problem is provided for the following three forms of SCAs:

- Synchronous Concurrent Algorithm;
- Abstract Dynamic Synchronous Concurrent Algorithm (2 forms with differing defining shapes are given); and
- Concrete Dynamic Synchronous Concurrent Algorithm.

As previously indicated, we are not going to claim that the solutions are formally correct, what is of more interest to us is that each of the models can be used to construct a representation of the solution to the problem. We semantically discuss the correctness of each model, and then in later chapters we discuss a convenient

method of demonstrating correctness between models in a hierarchy, which it turns out each of our descriptions in this chapter are.

During the introduction of SCAs and dSCAs we used the carrier algebra A to define which data and operations were possible within the SCA. For the remainder of this thesis we will deal with a specific instance of this algebra and will call this the machine algebra, or M_A. M_A is specified in Appendix A.

8.2 Case Study as an SCA

8.2.1 Informal Definition

Recall that the proposed solution to the GRCP with 2 tracks consisted of 8 sensors for the tracks, 2 sensors to indicate the gate positions, and the associated logic to move the gates in relation to the values held by the sensors.

It was proposed that the tracks in the region of interest, R, would be named tk_1 and tk_2. Each track would have two sensors sub-systems on it, one to the left of the gates and the other to the right of the gates, and each sensor sub-system would be constructed from two sensors, each capable of counting how many trains have passed in a particular direction, with the intention being that one sensor captures trains moving in to R and the other trains moving out of R, as shown in Figure 8.1.

The solution identified 2 distinct pieces of logic, one that interpreted the sensors and another that controlled the actual gate.

To identify if a train is in R the following logic test is performed:

$$inR(t) = \begin{pmatrix} ((s_{1,1}(t) - s_{1,3}(t) > 0) \vee (s_{1,4}(t) - s_{1,2}(t) > 0)) \vee \\ ((s_{2,1}(t) - s_{2,3}(t) > 0) \vee (s_{2,4}(t) - s_{2,2}(t) > 0)) \end{pmatrix}.$$

Simplistically, this could be translated into a single SCA module that takes 8 input streams as its input, one for each sensor, and produces a single output. However, for the purpose of demonstrating the same example across SCAs and dSCAs an SCA

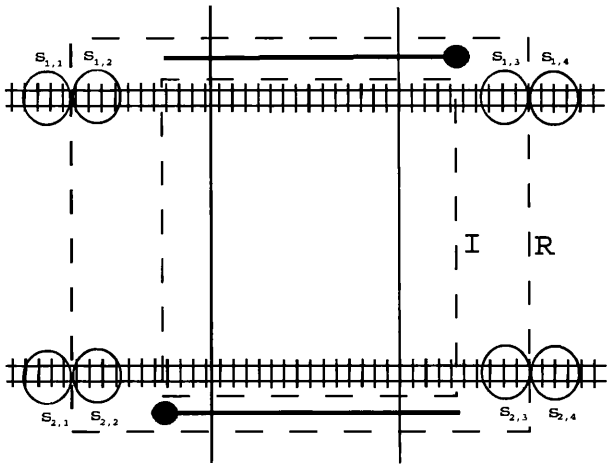

Figure 8.1: Physical GRCP Solution

will be developed where the functional definitions of modules are unit elements from the machine algebra, M_A (such an SCA is said to be atomic with respect to M_A, or simply an *atomic SCA*). Such an SCA implementation is graphically shown in Figure 8.2.

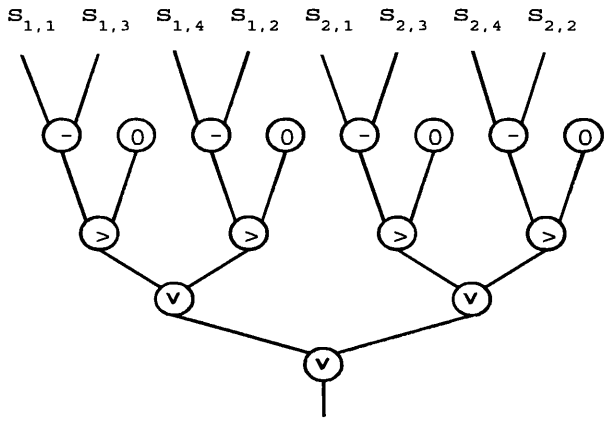

Figure 8.2: SCA Implementation of Sensor Logic

The logic for controlling the gates introduced the gate function, $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means it is up. This function will be implemented as a sensor on the gate providing an output in the required range,

A $motor(t)$ function was defined as:

$$motor(t) = \left(\begin{array}{ll} if & \left(\begin{array}{l} ((inR(t) = False \wedge g(t) = 90) \vee \\ (inR(t) = True \wedge g(t) = 0) \end{array} \right) then\ stay \\ else & if\ (inR(t) = True \wedge g(t) > 0) \\ & then\ down \\ & else\ up \end{array} \right)$$

An SCA that would represent this logic, as shown graphically in Figure 8.3.

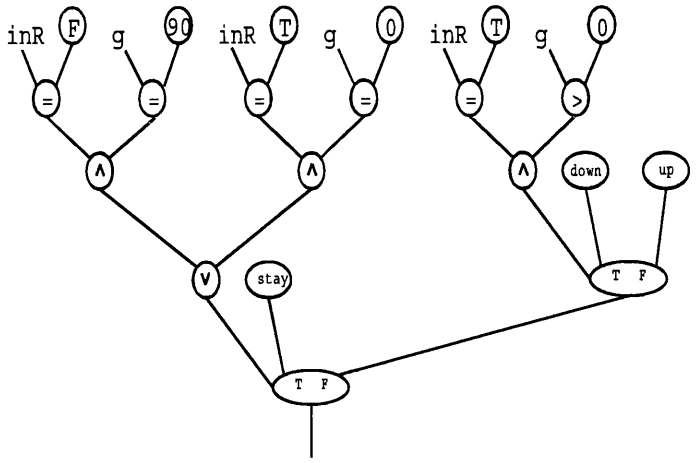

Figure 8.3: SCA Implementation of Motor Logic

In this implementation $inR(t)$ is the input from track sensors and $g(t)$ will be an actual input. Recall that we defined a reactive system to be a system such as that depicted in Figure 8.4.

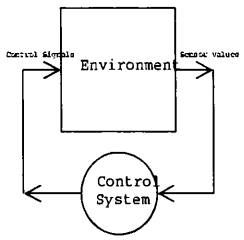

Figure 8.4: Example Reactive System

The overall SCA will be graphically represented as shown in Figure 8.5 which by showing one of the tracks and the gate sensor/controller sets the context of the SCA within the reactive system under consideration.

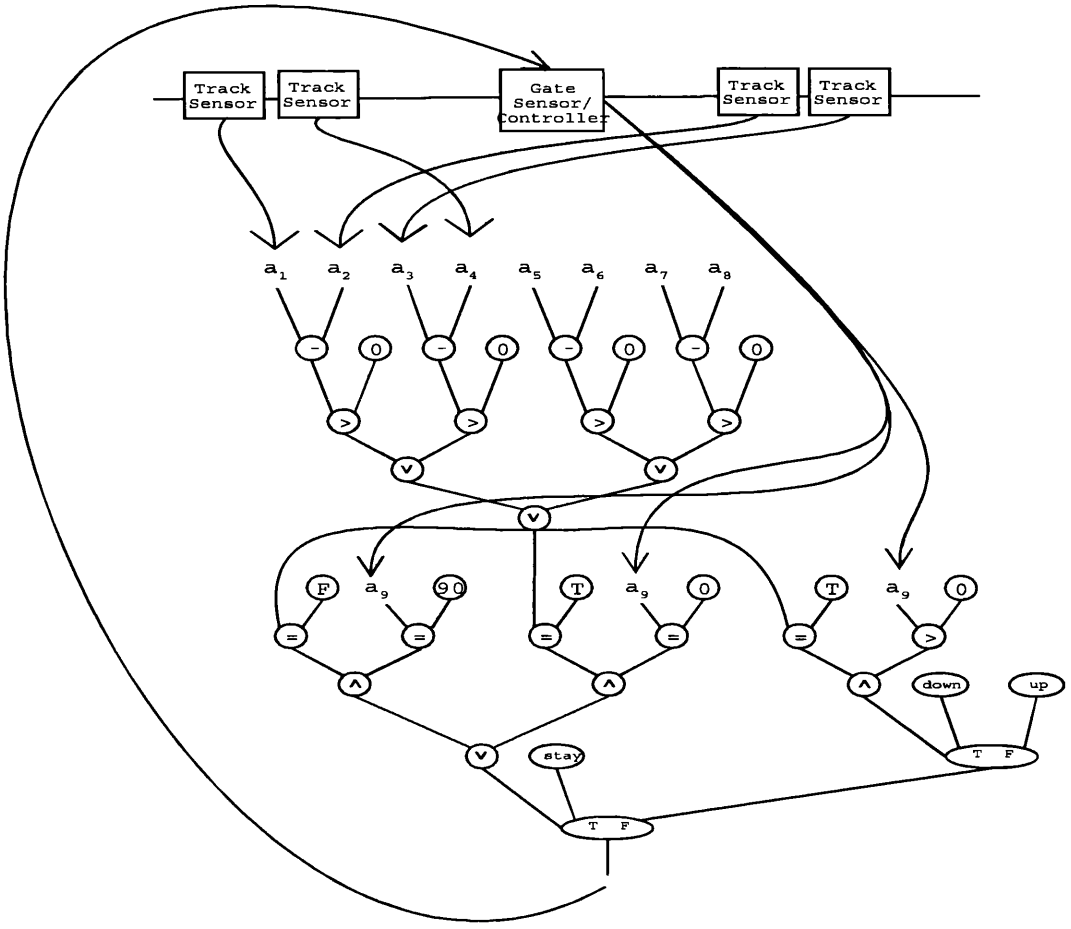

Figure 8.5: Complete SCA Implementation of GRCP

For future transformations the SCA implementation of the GRCP requires the modules in Figure 8.5 to be numbered in a breadth first and left to right manner from the bottom most module, which will be module 1, giving a total of 36 modules. For the purposes of defining the SCA the inputs are also renamed as shown in Table 8.1.

Diagram Name	SCA Name	Diagram Name	SCA Name
$s_{1,1}$	a_1	$s_{1,3}$	a_6
$s_{1,3}$	a_2	$s_{1,4}$	a_7
$s_{1,4}$	a_3	$s_{1,2}$	a_8
$s_{1,2}$	a_4	g	a_9
$s_{1,1}$	a_5		

Table 8.1: Renaming Inputs

8.2.2 Formal Definition

Wiring Functions

The γ wiring functions are defined by the following three definitions, where for $i = 1, \ldots, 15, 22, \ldots, 28$ and $j = 1, 2$:

$$\gamma(i, j) = M$$

with:

$$\gamma(1, 3) = M$$
$$\gamma(4, 3) = M$$

and:

$\gamma(11, 1) = S$	$\gamma(31, 1) = S$	$\gamma(31, 2) = S$
$\gamma(13, 1) = S$	$\gamma(33, 1) = S$	$\gamma(33, 2) = S$
$\gamma(15, 1) = S$	$\gamma(35, 1) = S$	$\gamma(35, 2) = S$
$\gamma(29, 1) = S$	$\gamma(29, 2) = S$	

The β wiring functions for the SCA solution to the GRCP are defined in Table 8.2.

Delay Functions

The delay function will be the unit delay for all inputs to all modules, thus for $i = 1, 2, \ldots, 35$ and $j = 1, 2, 3$ as follows:

$$\beta(i, j) \downarrow \Rightarrow \delta_{i,j}(t, a, x) = t - 1$$

$\beta(1,1) = 2$	$\beta(7,1) = 14$	$\beta(15,1) = 9$	$\beta(27,1) = 33$
$\beta(1,2) = 3$	$\beta(7,2) = 15$	$\beta(15,2) = 21$	$\beta(27,2) = 34$
$\beta(1,3) = 4$	$\beta(10,1) = 22$	$\beta(22,1) = 23$	$\beta(28,1) = 35$
$\beta(2,1) = 5$	$\beta(10,2) = 16$	$\beta(22,2) = 24$	$\beta(28,2) = 36$
$\beta(2,2) = 6$	$\beta(11,1) = 9$	$\beta(23,1) = 25$	$\beta(29,1) = 1$
$\beta(4,1) = 7$	$\beta(11,2) = 17$	$\beta(23,2) = 26$	$\beta(29,2) = 2$
$\beta(4,2) = 8$	$\beta(12,1) = 22$	$\beta(24,1) = 27$	$\beta(31,1) = 3$
$\beta(4,3) = 9$	$\beta(12,2) = 18$	$\beta(24,2) = 28$	$\beta(31,2) = 4$
$\beta(5,1) = 10$	$\beta(13,1) = 9$	$\beta(25,1) = 29$	$\beta(33,1) = 5$
$\beta(5,2) = 11$	$\beta(13,2) = 19$	$\beta(25,2) = 30$	$\beta(33,2) = 6$
$\beta(6,1) = 12$	$\beta(14,1) = 22$	$\beta(26,1) = 31$	$\beta(35,1) = 7$
$\beta(6,2) = 13$	$\beta(14,2) = 20$	$\beta(26,2) = 32$	$\beta(35,2) = 8$

Table 8.2: $\beta-$ Wiring Functions for SCA

Value Functions: Initial State

The actual values in the initial state vector x for the GRCP are not given in the original definition, however, the following assumptions are made:

- at initialisation there are no trains in R;
- that the gates are fully up; and
- the initial output signal *stay*.

The other initial values in the system are provided in such a way that the *stay* signal will be issued until the first input signals have propagated their way through the system. Table 8.3 shows the initial state phase definition of the value functions for this network.

Value Functions: State Transition

The state transition phase definition of the value functions for the control system

$V_1(0,a,x) = stay$	$V_2(0,a,x) = true$	$V_3(0,a,x) = stay$	$V_4(0,a,x) = up$
$V_5(0,a,x) = true$	$V_6(0,a,x) = false$	$V_7(0,a,x) = false$	$V_8(0,a,x) = down$
$V_9(0,a,x) = up$	$V_{10}(0,a,x) = true$	$V_{11}(0,a,x) = true$	$V_{12}(0,a,x) = false$
$V_{13}(0,a,x) = false$	$V_{14}(0,a,x) = false$	$V_{15}(0,a,x) = true$	$V_{16}(0,a,x) = false$
$V_{17}(0,a,x) = 90$	$V_{18}(0,a,x) = true$	$V_{19}(0,a,x) = 0$	$V_{20}(0,a,x) = true$
$V_{21}(0,a,x) = 0$	$V_{22}(0,a,x) = false$	$V_{23}(0,a,x) = false$	$V_{24}(0,a,x) = false$
$V_{25}(0,a,x) = false$	$V_{26}(0,a,x) = false$	$V_{27}(0,a,x) = false$	$V_{28}(0,a,x) = false$
$V_{29}(0,a,x) = 0$	$V_{30}(0,a,x) = 0$	$V_{31}(0,a,x) = 0$	$V_{32}(0,a,x) = 0$
$V_{33}(0,a,x) = 0$	$V_{34}(0,a,x) = 0$	$V_{35}(0,a,x) = 0$	$V_{36}(0,a,x) = 0$

Table 8.3: Initial State Values for SCA

are defined as follows:

$$
\begin{aligned}
V_1(t+1,a,x) &= cond\,(V_2(t,a,x), V_3(t,a,x), V_4(t,a,x)) \\
V_2(t+1,a,x) &= or(V_5(t,a,x), V_6(t,a,x)) \\
V_3(t+1,a,x) &= start \\
V_4(t+1,a,x) &= cond\,(V_7(t,a,x), V_8(t,a,x), V_9(t,a,x)) \\
V_5(t+1,a,x) &= and(V_{10}(t,a,x), V_{11}(t,a,x)) \\
V_6(t+1,a,x) &= and(V_{12}(t,a,x), V_{13}(t,a,x)) \\
V_7(t+1,a,x) &= and(V_{14}(t,a,x), V_{15}(t,a,x)) \\
V_8(t+1,a,x) &= down \\
V_9(t+1,a,x) &= up \\
V_{10}(t+1,a,x) &= eq(V_{22}(t,a,x), V_{16}(t,a,x)) \\
V_{11}(t+1,a,x) &= eq(a_9(t), V_{17}(t,a,x)) \\
V_{12}(t+1,a,x) &= eq(V_{22}(t,a,x), V_{18}(t,a,x)) \\
V_{13}(t+1,a,x) &= eq(a_9(t), V_{19}(t,a,x)) \\
V_{14}(t+1,a,x) &= eq(V_{22}(t,a,x), V_{20}(t,a,x)) \\
V_{15}(t+1,a,x) &= gt(a_9(t), V_{21}(t,a,x)) \\
V_{16}(t+1,a,x) &= false \\
V_{17}(t+1,a,x) &= 90 \\
V_{18}(t+1,a,x) &= true \\
V_{19}(t+1,a,x) &= 0 \\
V_{20}(t+1,a,x) &= true \\
V_{21}(t+1,a,x) &= 0 \\
V_{22}(t+1,a,x) &= or(V_{23}(t,a,x), V_{24}(t,a,x)) \\
V_{23}(t+1,a,x) &= or(V_{25}(t,a,x), V_{26}(t,a,x)) \\
V_{24}(t+1,a,x) &= or(V_{27}(t,a,x), V_{28}(t,a,x)) \\
V_{25}(t+1,a,x) &= gt(V_{29}(t,a,x), V_{30}(t,a,x)) \\
V_{26}(t+1,a,x) &= gt(V_{31}(t,a,x), V_{32}(t,a,x)) \\
V_{27}(t+1,a,x) &= gt(V_{33}(t,a,x), V_{34}(t,a,x)) \\
V_{28}(t+1,a,x) &= gt(V_{35}(t,a,x), V_{36}(t,a,x)) \\
V_{29}(t+1,a,x) &= sub(a_1(t), a_2(t)) \\
V_{30}(t+1,a,x) &= 0 \\
V_{31}(t+1,a,x) &= sub(a_3(t), a_4(t))
\end{aligned}
$$

$$V_{32}(t+1,a,x) = 0$$
$$V_{33}(t+1,a,x) = sub(a_5(t), a_6(t))$$
$$V_{34}(t+1,a,x) = 0$$
$$V_{35}(t+1,a,x) = sub(a_7(t), a_8(t))$$
$$V_{36}(t+1,a,x) = 0$$

The complete algebraic specification of the SCA is given in Annex B.

8.2.3 Correctness

There are two conditions that it are required to demonstrate as true according to the definition of the GRCP - namely the **Safety Property**, where the gate is down during all occupancy intervals, and the **Utility Property**, where the gate is up when no train is in the crossing.

To demonstrate correctness, an appeal is made to a semantic argument, divided into two parts (a syntactic proof along the lines identified in Chapter 5 using retimings could be constructed, but this would require a more mathematical specification than provided). First it is shown that if there are any number of trains in R, then they are correctly identified as being so. Secondly, depending on the identification of train(s) or not, the appropriate action is taken by the gate motor.

Lemma 8.2.1. *The existence of trains in R (therefore are either in the region of interest I or are heading into it)are identified by the output of the sensor subsystem being greater than zero.*

Figure 8.2 represents the sensor subsystem that is constructed from sets of similar logic. The basic unit is the subtraction of two values followed by a comparison with zero - the output being boolean.

Consider sensors $s_{1,1}$ and $s_{1,3}$ whose intention is to count trains entering and exiting from R from a left-right direction on the top track. As a train enters R the value of $s_{1,1}$ is incremented and when the train leaves R the value of $s_{1,3}$ is incremented. As

discussed previously, a train entering R means the gates must be down before that train enters the region of interest I, and a train leaving R implies that it is no longer in I.

It is clear that whilst there are trains in R there will be a difference in the values of $s_{1,1}$ and $s_{1,3}$. Logic is constructed to obtain the difference between the two sensor values and then compare this result with zero - a boolean true value indicating that a train or trains are in R on the top track going in a left to right direction.

Similar logic is constructed for the top track in a right-left direction, using sensors $s_{1,4}$ and $s_{1,2}$. For the second track, sensors $s_{2,1}$ and $s_{2,3}$ are fed to similar logic for trains moving in a left-right direction and sensors $s_{2,4}$ and $s_{2,2}$ used for right-left direction on the second track.

The result of all four sensor logic elements will be four boolean values, which are then consecutively "OR"-ed together, using the standard interpretation of boolean or operation, the result being a single boolean value of true, if there are any trains in R, or false if there are no trains in R.

Note that the unit delay between modules will introduce a delay of 4 time units before signals from the sensors propagate into the network. Given the speed of modern processors this is unlikely to be an issue. The benefit of an SCA approach is that this value can be identified, and in larger examples may even be used to feedback to the design. It should certainly be used in this example to help identify the distance that the sensor detecting trains entering into R are placed from R.

Lemma 8.2.2. *If a train enters R then the barrier is instructed to go down if it is not fully down, or stay down if it is already so.*

Figure 8.6 shows the motor logic and of particular interest are the two sections labelled A and B.

Consider logic block A, we are interested in the case where there is a train in R, $inR = T$ and if the gate is down then the whole of block A, through the OR and

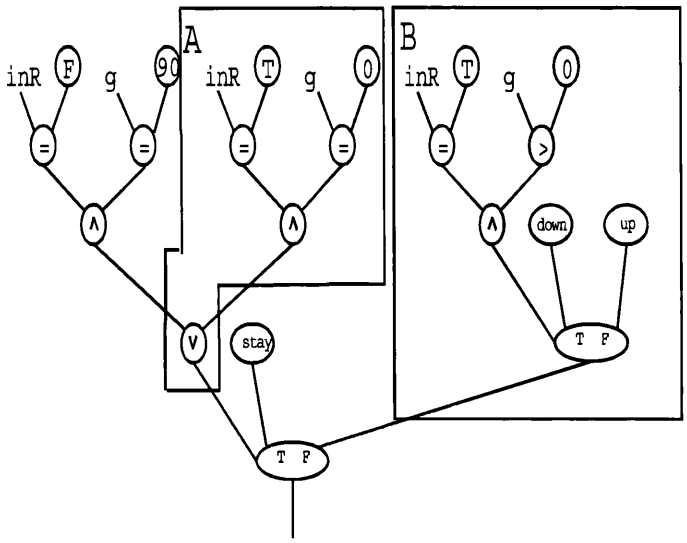

Figure 8.6: GRCP SCA Down Logic

AND operations, will result in a value of true. If this result is true, then the output module will pass the STAY result through. That is to say that if there is a train in R and the gates are down then they will be requested to stay down.

Alternatively, the gate will not be fully down and so the logic in block A will be false meaning the output module will pass through the logic from block B. The top left logic of block B will be true since train is in R and the gate not down, thus the DOWN signal will pass through the conditional gate in block B, and subsequently through the output module.

Lemma 8.2.3. *If all trains have left R then the barrier is instructed to go up if it is not fully up, or to stay up if it is already so.*

Figure 8.7 shows the motor logic and again of particular interest are the two sections labelled as A and B.

Looking at the logic in block A, if there are no trains in R and the gate is fully up, then the output module is provided a true signal and thus passes the stay signal.

If the gate is not fully up, the logic block A results in a false signal going to the output module and thus the result of logic block B is passed. Block B itself will

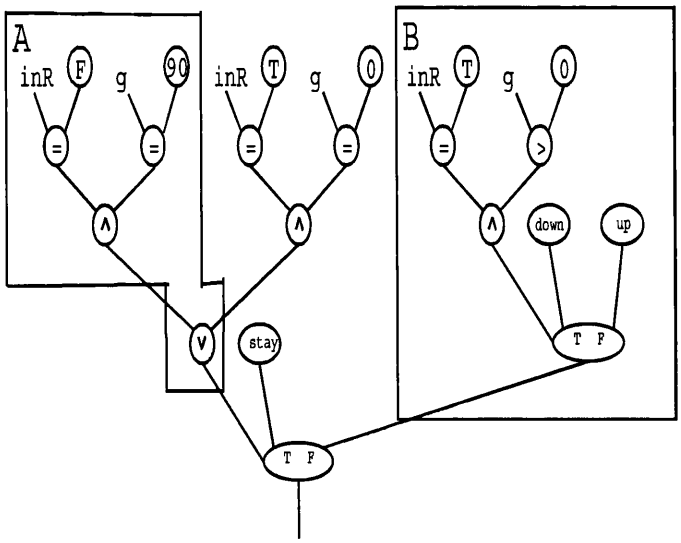

Figure 8.7: GRCP SCA Up Logic

output the UP signal since there is no train in R the conditional logic is false.

Safety Property

It has been shown that the SCA can correctly identify whether a train is in R or not, and further, that the correct signal (either *down* or *stay*) is sent to the gate motor depending on whether there are trains in R or not.

This does not fully address the safety property which is concerned with the region of interest I. For this property to be met, the gates must be fully down before the train reaches I, i.e. there is a defined delay between a identifying a train is in R and the train reaching I, i.e. the gates being fully down. By considering the graph it can be seen that it takes a minimum of 8 clock cycles before the system can react to a change in sensor values. In addition there is a delay, dependent upon the motor speed, required whilst the gate lowers.

Thus, together with a knowledge of train speeds towards I and the speed of closing a gate, it is possible to determine the appropriate distance from I that the entry sensor must be placed (the boundary of R) so that the safety property will be upheld, such

that for all times t in time intervals, λ_i: the gate is down:

$$t \in \bigcup_i \lambda_i \Longrightarrow g(t) = 0$$

Utility Property

The utility property states that the gate is up for all time intervals where there are no trains in R. In a similar manner to the safety property, it has been shown that the SCA can correctly identify when the train passes the bounday of R (i.e. there should be no trains in I) and that if this is the case then the gate motors are sent either the *up* or *stay* signal. Placing the exit sensor on the edge of R means there is a slight delay whilst the gates are opening, but allows us to confirm there are no trains in I - on the assumption that the distance between R and I is sufficient for a whole train to be held within.

The length of time required before a train enters R and after a train leaves R, to allow a reaction to the sensors and time for the gates to lower and open are the values ξ_1 and ξ_2 in the formal definition of the utility property:

$$t \notin \bigcup_i [\tau_i - \xi_1, \upsilon_i + \xi_2] \Longrightarrow g(t) = 90$$

The SCA implementation of the GRCP is shown in Appendix B.

8.3 Case Study as an Abstract dSCA

Two abstract dSCA implementations of the GRCP will be constructed: the first, which will be referred to as the Form 1 abstract dSCA, will be where the defining shape represents a simple SCA arrangement, i.e. $\nabla = (36, 1)$; and the second one, referred to as the Form 2 abstract dSCA, will be where the defining shape represents a single processor machine, i.e. $\nabla = (1, 36)$.

8.3.1 Form One Formal Definition

The first form for the abstract dSCA implementation is one where the defining shape is $\nabla = (36, 1)$, or simply it is an abstract dSCA that resembles the SCA. This first form can be diagrammatically seen in Figure 8.8, and differs from Figure 8.5 by the introduction of module m_{pc} and the associated wirings.

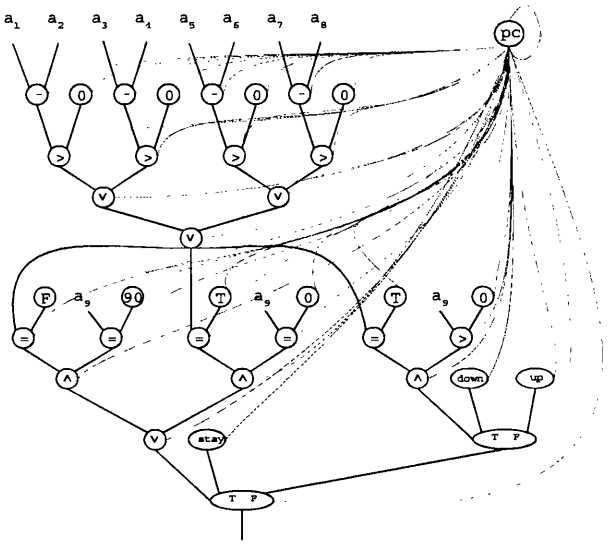

Figure 8.8: Form One abstract dSCA GRCP Solution

Wiring Functions

The definitions of the β-wiring functions for the first form of abstract dSCA are given in Table 8.4. It should be noted that there is quite some similarity between the definition of β-wiring for this abstract dSCA and those for the corresponding SCA. This similarity is intentional.

Table 8.5 shows the wirings to the ω element.

The definition of the γ wiring functions can be decribed using the following two definitions, where for $i = 1, 2, 3, \ldots, 28$ and $j = 0, \ldots, 3$:

$$\gamma(i,j) = \begin{cases} U & \text{if } \beta_0(i,j) = \omega \\ M & \text{otherwise} \end{cases}$$

$\beta_0(1,0) = pc$	$\beta_0(7,0) = pc$	$\beta_0(14,1) = 22$	$\beta_0(24,1) = 27$	$\beta_0(31,0) = pc$
$\beta_0(1,1) = 2$	$\beta_0(7,1) = 14$	$\beta_0(14,2) = 20$	$\beta_0(24,2) = 28$	$\beta_0(31,1) = 3$
$\beta_0(1,2) = 3$	$\beta_0(7,2) = 15$	$\beta_0(15,0) = pc$	$\beta_0(25,0) = pc$	$\beta_0(31,2) = 4$
$\beta_0(1,3) = 4$	$\beta_0(8,0) = pc$	$\beta_0(15,1) = 9$	$\beta_0(25,1) = 29$	$\beta_0(32,0) = pc$
$\beta_0(2,0) = pc$	$\beta_0(9,0) = pc$	$\beta_0(15,2) = 21$	$\beta_0(25,2) = 30$	$\beta_0(33,0) = pc$
$\beta_0(2,1) = 5$	$\beta_0(10,0) = pc$	$\beta_0(16,0) = pc$	$\beta_0(26,0) = pc$	$\beta_0(33,1) = 5$
$\beta_0(2,2) = 6$	$\beta_0(10,1) = 22$	$\beta_0(17,0) = pc$	$\beta_0(26,1) = 31$	$\beta_0(33,2) = 6$
$\beta_0(3,0) = pc$	$\beta_0(10,2) = 16$	$\beta_0(18,0) = pc$	$\beta_0(26,2) = 32$	$\beta_0(34,0) = pc$
$\beta_0(4,0) = pc$	$\beta_0(11,0) = pc$	$\beta_0(19,0) = pc$	$\beta_0(27,0) = pc$	$\beta_0(35,0) = pc$
$\beta_0(4,1) = 7$	$\beta_0(11,1) = 9$	$\beta_0(20,0) = pc$	$\beta_0(27,1) = 33$	$\beta_0(35,1) = 7$
$\beta_0(4,2) = 8$	$\beta_0(11,2) = 17$	$\beta_0(21,0) = pc$	$\beta_0(27,2) = 34$	$\beta_0(35,2) = 8$
$\beta_0(4,3) = 9$	$\beta_0(12,0) = pc$	$\beta_0(22,0) = pc$	$\beta_0(28,0) = pc$	$\beta_0(36,0) = pc$
$\beta_0(5,0) = pc$	$\beta_0(12,1) = 22$	$\beta_0(22,1) = 23$	$\beta_0(28,1) = 35$	$\beta_0(pc,1) = pc$
$\beta_0(5,1) = 10$	$\beta_0(12,2) = 18$	$\beta_0(22,2) = 24$	$\beta_0(28,2) = 36$	
$\beta_0(5,2) = 11$	$\beta_0(13,0) = pc$	$\beta_0(23,0) = pc$	$\beta_0(29,0) = pc$	
$\beta_0(6,0) = pc$	$\beta_0(13,1) = 9$	$\beta_0(23,1) = 25$	$\beta_0(29,1) = 1$	
$\beta_0(6,1) = 12$	$\beta_0(13,2) = 19$	$\beta_0(23,2) = 26$	$\beta_0(29,2) = 2$	
$\beta_0(6,2) = 13$	$\beta_0(14,0) = pc$	$\beta_0(24,0) = pc$	$\beta_0(30,0) = pc$	

Table 8.4: $\beta-$ Wiring Functions for Form 1 adSCA

and the following:

$$\gamma_0(11,1) = S$$
$$\gamma_0(13,1) = S$$
$$\gamma_0(15,1) = S$$
$$\gamma_0(29,1) = S$$
$$\gamma_0(31,1) = S$$
$$\gamma_0(33,1) = S$$
$$\gamma_0(35,1) = S$$
$$\gamma_0(29,2) = S$$
$$\gamma_0(31,2) = S$$
$$\gamma_0(33,2) = S$$
$$\gamma_0(35,2) = S$$

Delay Functions

Since this abstract dSCA is supposed to be a representation of the original SCA, then it is correct to define the delay function to be the unit delay for all inputs to all modules. It is therefore possible to describe the delay functions using a single

$\beta_0(2,3)=\omega$	$\beta_0(3,1)=\omega$	$\beta_0(3,2)=\omega$	$\beta_0(3,3)=\omega$	$\beta_0(5,3)=\omega$
$\beta_0(6,3)=\omega$	$\beta_0(7,3)=\omega$	$\beta_0(8,1)=\omega$	$\beta_0(8,2)=\omega$	$\beta_0(8,3)=\omega$
$\beta_0(9,1)=\omega$	$\beta_0(9,2)=\omega$	$\beta_0(9,3)=\omega$	$\beta_0(10,3)=\omega$	$\beta_0(11,3)=\omega$
$\beta_0(12,3)=\omega$	$\beta_0(13,3)=\omega$	$\beta_0(15,3)=\omega$	$\beta_0(15,3)=\omega$	$\beta_0(16,1)=\omega$
$\beta_0(16,2)=\omega$	$\beta_0(16,3)=\omega$	$\beta_0(17,1)=\omega$	$\beta_0(17,2)=\omega$	$\beta_0(17,3)=\omega$
$\beta_0(18,1)=\omega$	$\beta_0(18,2)=\omega$	$\beta_0(18,3)=\omega$	$\beta_0(19,1)=\omega$	$\beta_0(19,2)=\omega$
$\beta_0(19,3)=\omega$	$\beta_0(20,1)=\omega$	$\beta_0(20,2)=\omega$	$\beta_0(20,3)=\omega$	$\beta_0(21,1)=\omega$
$\beta_0(21,2)=\omega$	$\beta_0(21,3)=\omega$	$\beta_0(22,3)=\omega$	$\beta_0(23,3)=\omega$	$\beta_0(24,3)=\omega$
$\beta_0(25,3)=\omega$	$\beta_0(26,3)=\omega$	$\beta_0(27,3)=\omega$	$\beta_0(28,3)=\omega$	$\beta_0(29,3)=\omega$
$\beta_0(30,1)=\omega$	$\beta_0(30,2)=\omega$	$\beta_0(30,3)=\omega$	$\beta_0(31,3)=\omega$	$\beta_0(32,1)=\omega$
$\beta_0(32,2)=\omega$	$\beta_0(32,3)=\omega$	$\beta_0(33,3)=\omega$	$\beta_0(34,1)=\omega$	$\beta_0(34,2)=\omega$
$\beta_0(34,3)=\omega$	$\beta_0(35,3)=\omega$	$\beta_0(36,1)=\omega$	$\beta_0(36,2)=\omega$	$\beta_0(36,3)=\omega$

Table 8.5: $\beta-$ Wiring Functions to ω for Form 1 adSCA

equation for $i = 1, 2, \ldots, 35$ and $j = 0, 1, 2, 3$ as follows:

$$\delta_{i,j,0}(t, a, x) = t - 1$$

and

$$\delta_{pc,0,0}(t, a, x) = t - 1$$

Value Function: Initial State

These are defined in Table 8.6.

$V_1(0,a,x) = stay$	$V_2(0,a,x) = true$	$V_3(0,a,x) = stay$	$V_4(0,a,x) = up$
$V_5(0,a,x) = true$	$V_6(0,a,x) = false$	$V_7(0,a,x) = false$	$V_8(0,a,x) = down$
$V_9(0,a,x) = up$	$V_{10}(0,a,x) = true$	$V_{11}(0,a,x) = true$	$V_{12}(0,a,x) = false$
$V_{13}(0,a,x) = false$	$V_{14}(0,a,x) = false$	$V_{15}(0,a,x) = true$	$V_{16}(0,a,x) = false$
$V_{17}(0,a,x) = 90$	$V_{18}(0,a,x) = true$	$V_{19}(0,a,x) = 0$	$V_{20}(0,a,x) = true$
$V_{21}(0,a,x) = 0$	$V_{22}(0,a,x) = false$	$V_{23}(0,a,x) = false$	$V_{24}(0,a,x) = false$
$V_{25}(0,a,x) = false$	$V_{26}(0,a,x) = false$	$V_{27}(0,a,x) = false$	$V_{28}(0,a,x) = false$
$V_{29}(0,a,x) = 0$	$V_{30}(0,a,x) = 0$	$V_{31}(0,a,x) = 0$	$V_{32}(0,a,x) = 0$
$V_{33}(0,a,x) = 0$	$V_{34}(0,a,x) = 0$	$V_{35}(0,a,x) = 0$	$V_{36}(0,a,x) = 0$
$V_{pc}(0,a,x) = 0$			

Table 8.6: Initial State Values for abstract dSCA (Form 1)

Value Functions: State Transition

The corresponding definition of the State Transition phase for Value Functions for the abstract dSCA solution to the GRCP are given as follows:

$$
\begin{array}{ll}
V_1(t+1,a,x) = cond(V_2(t,a,x), V_3(t,a,x), V_4(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_2(t+1,a,x) = or(V_5(t,a,x), V_6(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_3(t+1,a,x) = start & \text{if } V_{pc}(t,a,x) = 0 \\
V_4(t+1,a,x) = cond(V_7(t,a,x), V_8(t,a,x), V_9(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_5(t+1,a,x) = and(V_{10}(t,a,x), V_{11}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_6(t+1,a,x) = and(V_{12}(t,a,x), V_{13}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_7(t+1,a,x) = and(V_{14}(t,a,x), V_{15}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_8(t+1,a,x) = down & \text{if } V_{pc}(t,a,x) = 0 \\
V_9(t+1,a,x) = up & \text{if } V_{pc}(t,a,x) = 0 \\
V_{10}(t+1,a,x) = eq(V_{22}(t,a,x), V_{16}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{11}(t+1,a,x) = eq(a_9(t), V_{17}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{12}(t+1,a,x) = eq(V_{22}(t,a,x), V_{18}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{13}(t+1,a,x) = eq(a_9(t), V_{19}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{14}(t+1,a,x) = eq(V_{22}(t,a,x), V_{20}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{15}(t+1,a,x) = gt(a_9(t), V_{21}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{16}(t+1,a,x) = false & \text{if } V_{pc}(t,a,x) = 0 \\
V_{17}(t+1,a,x) = 90 & \text{if } V_{pc}(t,a,x) = 0 \\
V_{18}(t+1,a,x) = true & \text{if } V_{pc}(t,a,x) = 0 \\
V_{19}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0 \\
V_{20}(t+1,a,x) = true & \text{if } V_{pc}(t,a,x) = 0 \\
V_{21}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0 \\
V_{22}(t+1,a,x) = or(V_{23}(t,a,x), V_{24}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{23}(t+1,a,x) = or(V_{25}(t,a,x), V_{26}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{24}(t+1,a,x) = or(V_{27}(t,a,x), V_{28}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{25}(t+1,a,x) = gt(V_{29}(t,a,x), V_{30}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{26}(t+1,a,x) = gt(V_{31}(t,a,x), V_{32}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{27}(t+1,a,x) = gt(V_{33}(t,a,x), V_{34}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{28}(t+1,a,x) = gt(V_{35}(t,a,x), V_{36}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{29}(t+1,a,x) = sub(a_1(t), a_2(t)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{30}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0 \\
V_{31}(t+1,a,x) = sub(a_3(t), a_4(t)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{32}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0 \\
V_{33}(t+1,a,x) = sub(a_5(t), a_6(t)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{34}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0 \\
V_{35}(t+1,a,x) = sub(a_7(t), a_8(t)) & \text{if } V_{pc}(t,a,x) = 0 \\
V_{36}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0
\end{array}
$$

With the program counter module defined as:

$$V_{pc}(t+1,a,x) = mod\left(add\left(V_{pc}(t,a,x),1\right),Max_N\right) \quad \text{if } V_{pc}(t,a,x) = 0$$

The algebraic specification of the Form 1 abstract dSCA is provided in Appendix C.

Correctness

Lemma 8.3.1. *The form 1 abstract dSCA solution to the GRCP is equivalent to the SCA implementation.*

Correctness of this abstract dSCA is addressed by appealing to the structural similarities between the SCA and the Form 1 abstract dSCA, see Figure 8.5 for the SCA and Figure 8.8 for the Form 1 abstract dSCA.

Now consider the initial state values of the SCA, which are repeated in Table 8.7:

$V_1(0,a,x) = stay$	$V_2(0,a,x) = true$	$V_3(0,a,x) = stay$	$V_4(0,a,x) = up$
$V_5(0,a,x) = true$	$V_6(0,a,x) = false$	$V_7(0,a,x) = false$	$V_8(0,a,x) = down$
$V_9(0,a,x) = up$	$V_{10}(0,a,x) = true$	$V_{11}(0,a,x) = true$	$V_{12}(0,a,x) = false$
$V_{13}(0,a,x) = false$	$V_{14}(0,a,x) = false$	$V_{15}(0,a,x) = true$	$V_{16}(0,a,x) = false$
$V_{17}(0,a,x) = 90$	$V_{18}(0,a,x) = true$	$V_{19}(0,a,x) = 0$	$V_{20}(0,a,x) = true$
$V_{21}(0,a,x) = 0$	$V_{22}(0,a,x) = false$	$V_{23}(0,a,x) = false$	$V_{24}(0,a,x) = false$
$V_{25}(0,a,x) = false$	$V_{26}(0,a,x) = false$	$V_{27}(0,a,x) = false$	$V_{28}(0,a,x) = false$
$V_{29}(0,a,x) = 0$	$V_{30}(0,a,x) = 0$	$V_{31}(0,a,x) = 0$	$V_{32}(0,a,x) = 0$
$V_{33}(0,a,x) = 0$	$V_{34}(0,a,x) = 0$	$V_{35}(0,a,x) = 0$	$V_{36}(0,a,x) = 0$

Table 8.7: Initial State Values for SCA

A direct one-to-one mapping can be seen between these values and those given in Table 8.6.

Similarly, the state transition definitions of the value functions of the SCA, repeated here:

$$
\begin{aligned}
V_1(t+1,a,x) &= cond\,(V_2(t,a,x), V_3(t,a,x), V_4(t,a,x)) \\
V_2(t+1,a,x) &= or(V_5(t,a,x), V_6(t,a,x)) \\
V_3(t+1,a,x) &= start \\
V_4(t+1,a,x) &= cond\,(V_7(t,a,x), V_8(t,a,x), V_9(t,a,x)) \\
V_5(t+1,a,x) &= and(V_{10}(t,a,x), V_{11}(t,a,x)) \\
V_6(t+1,a,x) &= and(V_{12}(t,a,x), V_{13}(t,a,x)) \\
V_7(t+1,a,x) &= and(V_{14}(t,a,x), V_{15}(t,a,x)) \\
V_8(t+1,a,x) &= down \\
V_9(t+1,a,x) &= up \\
V_{10}(t+1,a,x) &= eq(V_{22}(t,a,x), V_{16}(t,a,x)) \\
V_{11}(t+1,a,x) &= eq(a_9(t), V_{17}(t,a,x)) \\
V_{12}(t+1,a,x) &= eq(V_{22}(t,a,x), V_{18}(t,a,x)) \\
V_{13}(t+1,a,x) &= eq(a_9(t), V_{19}(t,a,x)) \\
V_{14}(t+1,a,x) &= eq(V_{22}(t,a,x), V_{20}(t,a,x)) \\
V_{15}(t+1,a,x) &= gt(a_9(t), V_{21}(t,a,x)) \\
V_{16}(t+1,a,x) &= false \\
V_{17}(t+1,a,x) &= 90 \\
V_{18}(t+1,a,x) &= true \\
V_{19}(t+1,a,x) &= 0 \\
V_{20}(t+1,a,x) &= true \\
V_{21}(t+1,a,x) &= 0 \\
V_{22}(t+1,a,x) &= or(V_{23}(t,a,x), V_{24}(t,a,x)) \\
V_{23}(t+1,a,x) &= or(V_{25}(t,a,x), V_{26}(t,a,x)) \\
V_{24}(t+1,a,x) &= or(V_{27}(t,a,x), V_{28}(t,a,x)) \\
V_{25}(t+1,a,x) &= gt(V_{29}(t,a,x), V_{30}(t,a,x)) \\
V_{26}(t+1,a,x) &= gt(V_{31}(t,a,x), V_{32}(t,a,x)) \\
V_{27}(t+1,a,x) &= gt(V_{33}(t,a,x), V_{34}(t,a,x)) \\
V_{28}(t+1,a,x) &= gt(V_{35}(t,a,x), V_{36}(t,a,x)) \\
V_{29}(t+1,a,x) &= sub(a_1(t), a_2(t)) \\
V_{30}(t+1,a,x) &= 0 \\
V_{31}(t+1,a,x) &= sub(a_3(t), a_4(t)) \\
V_{32}(t+1,a,x) &= 0 \\
V_{33}(t+1,a,x) &= sub(a_5(t), a_6(t)) \\
V_{34}(t+1,a,x) &= 0 \\
V_{35}(t+1,a,x) &= sub(a_7(t), a_8(t)) \\
V_{36}(t+1,a,x) &= 0
\end{aligned}
$$

demonstrate a one-to-one mapping to the state transition definitions of the value functions of the form 1 abstract dSCA, shown above (where the reference to the program counter is added).

Since $V_{pc}(t,a,x)$ is the result of adding 1 to the previous value mod $Max_N = 1$

then the answer will always be 0 thus the Form 1 adSCA State Transition definitions for the Value Functions will directly equate to the SCA State Transition definitions.

It can therefore be seen by inspection that the SCA and Form 1 abstract dSCA are equivalent.

8.3.2 Form Two Formal Definition

The 2nd form abstract dSCA is a single module implementation, where the defining shape is given as $\nabla = (35, 1)$. In this case, the value of Max_N will be 36; and whilst it is difficult to diagrammatically show an abstract dSCA where $Max_N > 1$ the shape is indicate in Figure 8.9.

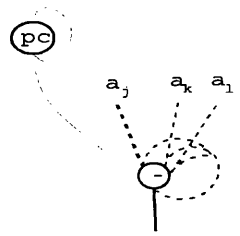

Figure 8.9: Form Two abstract dSCA GRCP Solution

It is decided to implement a cycle consistent abstract dSCA and therefore the execution order needs to adhere to the principles of cycle consistency. This is achieved by deriving the execution order from the module numbers in the Form 1 abstract dSCA. Table 8.8 shows the proposed execution order:

A simple inspection of this execution order will demonstrate that the resulting abstract dSCA is cycle consistent.

Wiring Functions

β wiring functions are defined in Table 8.9.

It is also the case that the 0^{th} input for each module is wired to the program counter,

Form 1	Form 2		Form 1	Form 2		Form 1	Form 2	
Module	Module	PC Val	Module	Module	PC Val	Module	Module	PC Val
1	1	0	13	1	12	25	1	24
2	1	1	14	1	13	26	1	25
3	1	2	15	1	14	27	1	26
4	1	3	16	1	15	28	1	27
5	1	4	17	1	16	29	1	28
6	1	5	18	1	17	30	1	29
7	1	6	19	1	18	31	1	30
8	1	7	20	1	19	32	1	31
9	1	8	21	1	20	33	1	32
10	1	9	22	1	21	34	1	33
11	1	10	23	1	22	35	1	34
12	1	11	24	1	23	36	1	35

Table 8.8: Execution Order of Form 2 abstract dSCA

so that for $pc\_val = 0, \ldots, Max_N - 1$:

$$\beta_{pc\_val}(1, 0) = pc$$

and that for the program counter the definition is that for $pc\_val = 0, \ldots, Max_N - 1$:

$$\beta_{pc\_val}(pc, 0) = pc$$

The γ wiring functions, indicating whether a module is linked to a module, source or unconnected are defined in Table 8.10.

It is also the case that the 0^{th} input for each module is wired to the program counter, so that for $pc\_val = 0, \ldots, Max_N - 1$:

$$\beta_{pc\_val}(1, 0) = M$$

and that for the program counter the definition is that for $pc\_val = 0, \ldots, Max_N - 1$:

$$\beta_{pc\_val}(pc, 0) = M$$

$\beta_0(1,1)=1$	$\beta_7(1,1)=\omega$	$\beta_{14}(1,1)=9$	$\beta_{21}(1,1)=1$	$\beta_{28}(1,1)=1$
$\beta_0(1,2)=1$	$\beta_7(1,2)=\omega$	$\beta_{14}(1,2)=1$	$\beta_{21}(1,2)=1$	$\beta_{28}(1,2)=2$
$\beta_0(1,3)=1$	$\beta_7(1,3)=\omega$	$\beta_{14}(1,3)=\omega$	$\beta_{21}(1,3)=\omega$	$\beta_{28}(1,3)=\omega$
$\beta_1(1,1)=1$	$\beta_8(1,1)=\omega$	$\beta_{15}(1,1)=\omega$	$\beta_{22}(1,1)=1$	$\beta_{29}(1,1)=\omega$
$\beta_1(1,2)=1$	$\beta_8(1,2)=\omega$	$\beta_{15}(1,2)=\omega$	$\beta_{22}(1,2)=1$	$\beta_{29}(1,2)=\omega$
$\beta_1(1,3)=\omega$	$\beta_8(1,3)=\omega$	$\beta_{15}(1,3)=\omega$	$\beta_{22}(1,3)=\omega$	$\beta_{29}(1,3)=\omega$
$\beta_2(1,1)=\omega$	$\beta_9(1,1)=1$	$\beta_{16}(1,1)=\omega$	$\beta_{23}(1,1)=1$	$\beta_{30}(1,1)=3$
$\beta_2(1,2)=\omega$	$\beta_9(1,2)=1$	$\beta_{16}(1,2)=\omega$	$\beta_{23}(1,2)=1$	$\beta_{30}(1,2)=4$
$\beta_2(1,3)=\omega$	$\beta_9(1,3)=\omega$	$\beta_{16}(1,3)=\omega$	$\beta_{23}(1,3)=\omega$	$\beta_{30}(1,3)=\omega$
$\beta_3(1,1)=1$	$\beta_{10}(1,1)=9$	$\beta_{17}(1,1)=\omega$	$\beta_{24}(1,1)=1$	$\beta_{31}(1,1)=\omega$
$\beta_3(1,2)=1$	$\beta_{10}(1,2)=1$	$\beta_{17}(1,2)=\omega$	$\beta_{24}(1,2)=1$	$\beta_{31}(1,2)=\omega$
$\beta_3(1,3)=1$	$\beta_{10}(1,3)=\omega$	$\beta_{17}(1,3)=\omega$	$\beta_{24}(1,3)=\omega$	$\beta_{31}(1,3)=\omega$
$\beta_4(1,1)=1$	$\beta_{11}(1,1)=1$	$\beta_{18}(1,1)=\omega$	$\beta_{25}(1,1)=1$	$\beta_{32}(1,1)=5$
$\beta_4(1,2)=1$	$\beta_{11}(1,2)=1$	$\beta_{18}(1,2)=\omega$	$\beta_{25}(1,2)=1$	$\beta_{32}(1,2)=6$
$\beta_4(1,3)=\omega$	$\beta_{11}(1,3)=\omega$	$\beta_{18}(1,3)=\omega$	$\beta_{25}(1,3)=\omega$	$\beta_{32}(1,3)=\omega$
$\beta_5(1,1)=1$	$\beta_{12}(1,1)=9$	$\beta_{19}(1,1)=\omega$	$\beta_{26}(1,1)=1$	$\beta_{33}(1,1)=\omega$
$\beta_5(1,2)=1$	$\beta_{12}(1,2)=1$	$\beta_{19}(1,2)=\omega$	$\beta_{26}(1,2)=1$	$\beta_{33}(1,2)=\omega$
$\beta_5(1,3)=\omega$	$\beta_{12}(1,3)=\omega$	$\beta_{19}(1,3)=\omega$	$\beta_{26}(1,3)=\omega$	$\beta_{33}(1,3)=\omega$
$\beta_6(1,1)=1$	$\beta_{13}(1,1)=1$	$\beta_{20}(1,1)=\omega$	$\beta_{27}(1,1)=1$	$\beta_{34}(1,1)=7$
$\beta_6(1,2)=1$	$\beta_{13}(1,2)=1$	$\beta_{20}(1,2)=\omega$	$\beta_{27}(1,2)=1$	$\beta_{34}(1,2)=8$
$\beta_6(1,3)=\omega$	$\beta_{13}(1,3)=\omega$	$\beta_{20}(1,3)=\omega$	$\beta_{27}(1,3)=\omega$	$\beta_{34}(1,3)=\omega$
		$\beta_{35}(1,1)=\omega$	$\beta_{35}(1,2)=\omega$	$\beta_{35}(1,3)=\omega$

Table 8.9: $\beta-$ Wiring Functions for Form 2 adSCA

Delay Functions

There is only one module within the 2nd form of abstract dSCA, and the delay functions will need to reflect this, i.e. values required will have been calculated some time in the past (bounded by Max_N).

For complicated examples it will be difficult to do this by hand, and in Chapter 11 this thesis provides a mechanical way of identifying these delays. It is defined that for an abstract dSCA all delays where the wiring is to an input are unit delays, as are the delays for the 0^{th} argument (which goes to the program counter).

Consider the module that executes at $pc = 36$ in the Form 2 abstract dSCA, from Table 8.8 it is possible to identify that this was module 1 in the Form 1 dSCA. It is also

$\gamma_0(1,1) = M$	$\gamma_7(1,1) = U$	$\gamma_{14}(1,1) = S$	$\gamma_{21}(1,1) = M$	$\gamma_{28}(1,1) = S$
$\gamma_0(1,2) = M$	$\gamma_7(1,2) = U$	$\gamma_{14}(1,2) = M$	$\gamma_{21}(1,2) = M$	$\gamma_{28}(1,2) = S$
$\gamma_0(1,3) = M$	$\gamma_7(1,3) = U$	$\gamma_{14}(1,3) = U$	$\gamma_{21}(1,3) = U$	$\gamma_{28}(1,3) = U$
$\gamma_1(1,1) = M$	$\gamma_8(1,1) = U$	$\gamma_{15}(1,1) = U$	$\gamma_{22}(1,1) = M$	$\gamma_{29}(1,1) = U$
$\gamma_1(1,2) = M$	$\gamma_8(1,2) = U$	$\gamma_{15}(1,2) = U$	$\gamma_{22}(1,2) = M$	$\gamma_{29}(1,2) = U$
$\gamma_1(1,3) = U$	$\gamma_8(1,3) = U$	$\gamma_{15}(1,3) = U$	$\gamma_{22}(1,3) = U$	$\gamma_{29}(1,3) = U$
$\gamma_2(1,1) = U$	$\gamma_9(1,1) = M$	$\gamma_{16}(1,1) = U$	$\gamma_{23}(1,1) = M$	$\gamma_{30}(1,1) = S$
$\gamma_2(1,2) = U$	$\gamma_9(1,2) = M$	$\gamma_{16}(1,2) = U$	$\gamma_{23}(1,2) = M$	$\gamma_{30}(1,2) = S$
$\gamma_2(1,3) = U$	$\gamma_9(1,3) = U$	$\gamma_{16}(1,3) = U$	$\gamma_{23}(1,3) = U$	$\gamma_{30}(1,3) = U$
$\gamma_3(1,1) = M$	$\gamma_{10}(1,1) = S$	$\gamma_{17}(1,1) = U$	$\gamma_{24}(1,1) = M$	$\gamma_{31}(1,1) = U$
$\gamma_3(1,2) = M$	$\gamma_{10}(1,2) = M$	$\gamma_{17}(1,2) = U$	$\gamma_{24}(1,2) = M$	$\gamma_{31}(1,2) = U$
$\gamma_3(1,3) = M$	$\gamma_{10}(1,3) = U$	$\gamma_{17}(1,3) = U$	$\gamma_{24}(1,3) = U$	$\gamma_{31}(1,3) = U$
$\gamma_4(1,1) = M$	$\gamma_{11}(1,1) = M$	$\gamma_{18}(1,1) = U$	$\gamma_{25}(1,1) = M$	$\gamma_{32}(1,1) = S$
$\gamma_4(1,2) = M$	$\gamma_{11}(1,2) = M$	$\gamma_{18}(1,2) = U$	$\gamma_{25}(1,2) = M$	$\gamma_{32}(1,2) = S$
$\gamma_4(1,3) = U$	$\gamma_{11}(1,3) = U$	$\gamma_{18}(1,3) = U$	$\gamma_{25}(1,3) = U$	$\gamma_{32}(1,3) = U$
$\gamma_5(1,1) = M$	$\gamma_{12}(1,1) = S$	$\gamma_{19}(1,1) = U$	$\gamma_{26}(1,1) = M$	$\gamma_{33}(1,1) = U$
$\gamma_5(1,2) = M$	$\gamma_{12}(1,2) = M$	$\gamma_{19}(1,2) = U$	$\gamma_{26}(1,2) = M$	$\gamma_{33}(1,2) = U$
$\gamma_5(1,3) = U$	$\gamma_{12}(1,3) = U$	$\gamma_{19}(1,3) = U$	$\gamma_{26}(1,3) = U$	$\gamma_{33}(1,3) = U$
$\gamma_6(1,1) = M$	$\gamma_{13}(1,1) = M$	$\gamma_{20}(1,1) = U$	$\gamma_{27}(1,1) = M$	$\gamma_{34}(1,1) = S$
$\gamma_6(1,2) = M$	$\gamma_{13}(1,2) = M$	$\gamma_{20}(1,2) = U$	$\gamma_{27}(1,2) = M$	$\gamma_{34}(1,2) = S$
$\gamma_6(1,3) = U$	$\gamma_{13}(1,3) = U$	$\gamma_{20}(1,3) = U$	$\gamma_{27}(1,3) = U$	$\gamma_{34}(1,3) = U$
		$\gamma_{35}(1,1) = U$	$\gamma_{35}(1,2) = U$	$\gamma_{35}(1,3) = U$

Table 8.10: $\gamma-$ Wiring Functions for Form 2 acvSCA

known that the values used as inputs to module 1 in the Form 1 abstract dSCA come from modules 2, 3 and 4. Again, using Table 8.8 it can be identified that modules 2,3, and 4 are now executed on module 1 at values of $pc = 1, 2, 3$ respectively. It is therefore the case that the first input to Form 2 module 1 at $pc = 36$ was calculated at $pc = 1$, or 35 clock cycles ago. Similar maths can be applied to the other arguments to obtain the value of the delays for those inputs. The inputs to module 1 at $pc = 36$ will therefore be:

$$\delta_{1,1,0} = t - 34$$
$$\delta_{1,1,0} = t - 33$$
$$\delta_{1,1,0} = t - 32$$

A similar process can be applied to all the delay functions in Form 2, and the resultant delay functions for all inputs which relate to the situation where $\gamma_y(i,j) = M$ are shown in Table 8.12.

$\delta_{1,1,0}(t,a,x) = t-35$	$\delta_{1,1,6}(t,a,x) = t-29$	$\delta_{1,2,21}(t,a,x) = t-34$
$\delta_{1,2,0}(t,a,x) = t-34$	$\delta_{1,2,6}(t,a,x) = t-28$	$\delta_{1,1,22}(t,a,x) = t-34$
$\delta_{1,3,0}(t,a,x) = t-33$	$\delta_{1,1,9}(t,a,x) = t-24$	$\delta_{1,2,22}(t,a,x) = t-33$
$\delta_{1,1,1}(t,a,x) = t-33$	$\delta_{1,2,9}(t,a,x) = t-30$	$\delta_{1,1,23}(t,a,x) = t-33$
$\delta_{1,2,1}(t,a,x) = t-32$	$\delta_{1,2,10}(t,a,x) = t-30$	$\delta_{1,2,23}(t,a,x) = t-32$
$\delta_{1,1,3}(t,a,x) = t-33$	$\delta_{1,1,11}(t,a,x) = t-26$	$\delta_{1,1,24}(t,a,x) = t-32$
$\delta_{1,2,3}(t,a,x) = t-32$	$\delta_{1,2,11}(t,a,x) = t-30$	$\delta_{1,2,24}(t,a,x) = t-31$
$\delta_{1,3,3}(t,a,x) = t-31$	$\delta_{1,2,12}(t,a,x) = t-30$	$\delta_{1,1,25}(t,a,x) = t-31$
$\delta_{1,1,4}(t,a,x) = t-31$	$\delta_{1,1,13}(t,a,x) = t-28$	$\delta_{1,2,25}(t,a,x) = t-30$
$\delta_{1,2,4}(t,a,x) = t-30$	$\delta_{1,2,13}(t,a,x) = t-30$	$\delta_{1,1,26}(t,a,x) = t-30$
$\delta_{1,1,5}(t,a,x) = t-30$	$\delta_{1,2,14}(t,a,x) = t-30$	$\delta_{1,2,26}(t,a,x) = t-29$
$\delta_{1,2,5}(t,a,x) = t-29$	$\delta_{1,1,21}(t,a,x) = t-35$	$\delta_{1,1,27}(t,a,x) = t-29$
		$\delta_{1,1,27}(t,a,x) = t-28$

Table 8.12: Non-unit Delay Functions for Form 2 acvSCA

Value Functions: Initial State

The Initial State definitions of the Value Functions for the 2nd form of the abstract dSCA solution are defined for both modules $i = 0, 1$. For the program counter, the initial states are defined, for values of program counter $pc\_val = 0, \ldots, 35$, as:

$$V_{pc}(pc\_val, a, x) = x_{pc,pc\_val}$$

with the initial values for the program counter defined for $pc\_val = 0, \ldots, 35$ as:

$$x_{pc,pc\_val} = mod(add(pc\_val + 1), Max_N)$$

Initial State value equation for module 1 are related closely to those in the Form 1 abstract dSCA. The Form 2 values are given in Table 8.13.

Value Functions: State Transition

$V_1(0,a,x) = stay$	$V_1(1,a,x) = true$	$V_1(2,a,x) = stay$
$V_1(3,a,x) = up$	$V_1(4,a,x) = true$	$V_1(5,a,x) = false$
$V_1(6,a,x) = false$	$V_1(7,a,x) = down$	$V_1(8,a,x) = up$
$V_1(9,a,x) = true$	$V_1(10,a,x) = true$	$V_1(11,a,x) = false$
$V_1(12,a,x) = false$	$V_1(13,a,x) = false$	$V_1(14,a,x) = true$
$V_1(15,a,x) = false$	$V_1(16,a,x) = 90$	$V_1(17,a,x) = true$
$V_1(18,a,x) = 0$	$V_1(19,a,x) = true$	$V_1(20,a,x) = 0$
$V_1(21,a,x) = false$	$V_1(22,a,x) = false$	$V_1(23,a,x) = false$
$V_1(24,a,x) = false$	$V_1(25,a,x) = false$	$V_1(26,a,x) = false$
$V_1(27,a,x) = false$	$V_1(28,a,x) = 0$	$V_1(29,a,x) = 0$
$V_1(30,a,x) = 0$	$V_1(31,a,x) = 0$	$V_1(32,a,x) = 0$
$V_1(33,a,x) = 0$	$V_1(34,a,x) = 0$	$V_1(35,a,x) = 0$

Table 8.13: Initial State Values for abstract dSCA (Form 2)

The State Transition definition of the program counter Value Function is defined as follows:

$$V_{pc}(t+1,a,x) = \begin{cases} add(V_{pc}(t,a,x),1) mod 36 & \text{if } V_{pc}(t,a,x) = 0 \\ \vdots & \\ add(V_{pc}(t,a,x),1) mod 36 & \text{if } V_{pc}(t,a,x) = 35 \end{cases}$$

For module 1 the State Transition definition of the Value Function is constructed to take account of the Max_N cases. The complete definition is as follows.

$$
V_1(t,a,x) = \begin{cases}
cond\begin{pmatrix} V_1(t-35,a,x), \\ V_1(t-34,a,x), \\ V_1(t-33,a,x) \end{pmatrix} & \text{if } V_{pc}(t-1,a,x) = 0 \\
or(V_1(t-33,a,x), V_6(t-32,a,x)) & \text{if } V_{pc}(t-1,a,x) = 1 \\
start & \text{if } V_{pc}(t-1,a,x) = 2 \\
cond\begin{pmatrix} V_1(t-33,a,x), \\ V_1(t-32,a,x), \\ V_1(t-31,a,x) \end{pmatrix} & \text{if } V_{pc}(t-1,a,x) = 3 \\
and(V_1(t-31,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 4 \\
and(V_1(t-30,a,x), V_1(t-29,a,x)) & \text{if } V_{pc}(t-1,a,x) = 5 \\
and(V_1(t-29,a,x), V_1(t-28,a,x)) & \text{if } V_{pc}(t-1,a,x) = 6 \\
down & \text{if } V_{pc}(t-1,a,x) = 7 \\
up & \text{if } V_{pc}(t-1,a,x) = 8 \\
eq(V_1(t-24,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 9 \\
eq(a_9(t), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 10 \\
eq(V_1(t-26,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 11 \\
eq(a_9(t), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 12 \\
eq(V_1(t-28,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 13 \\
gt(a_9(t), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 14 \\
false & \text{if } V_{pc}(t-1,a,x) = 15 \\
90 & \text{if } V_{pc}(t-1,a,x) = 16 \\
true & \text{if } V_{pc}(t-1,a,x) = 17 \\
0 & \text{if } V_{pc}(t-1,a,x) = 18 \\
true & \text{if } V_{pc}(t-1,a,x) = 19 \\
0 & \text{if } V_{pc}(t-1,a,x) = 20 \\
or(V_1(t-35,a,x), V_1(t-34,a,x)) & \text{if } V_{pc}(t-1,a,x) = 21 \\
or(V_1(t-34,a,x), V_1(t-33,a,x)) & \text{if } V_{pc}(t-1,a,x) = 22 \\
or(V_1(t-33,a,x), V_1(t-32,a,x)) & \text{if } V_{pc}(t-1,a,x) = 23 \\
gt(V_1(t-33,a,x), V_1(t-31,a,x)) & \text{if } V_{pc}(t-1,a,x) = 24 \\
gt(V_1(t-31,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 25 \\
gt(V_1(t-30,a,x), V_1(t-29,a,x)) & \text{if } V_{pc}(t-1,a,x) = 26 \\
gt(V_1(t-29,a,x), V_1(t-28,a,x)) & \text{if } V_{pc}(t-1,a,x) = 27 \\
sub(a_1(t), a_2(t)) & \text{if } V_{pc}(t-1,a,x) = 28 \\
0 & \text{if } V_{pc}(t-1,a,x) = 29 \\
sub(a_3(t), a_4(t)) & \text{if } V_{pc}(t-1,a,x) = 30 \\
0 & \text{if } V_{pc}(t-1,a,x) = 31 \\
sub(a_5(t), a_6(t)) & \text{if } V_{pc}(t-1,a,x) = 32 \\
0 & \text{if } V_{pc}(t-1,a,x) = 33 \\
sub(a_7(t), a_8(t)) & \text{if } V_{pc}(t-1,a,x) = 34 \\
0 & \text{if } V_{pc}(t-1,a,x) = 35
\end{cases}
$$

Correctness

Lemma 8.3.2. *The Form 2 abstract dSCA is a correct implementation of the Form 1 abstract dSCA.*

Consider any time $t \in T$ then the value of the program counter will be t mod Max_N. By inspection, it can be shown that values and operations in the initial values and state transition functions in the Form 2 abstract dSCA map directly to the Form 2 abstract dSCA.

The algebraic specification of the Form 2 Abstract dSCA is shown in Appendix D.

8.4 Case Study as a Concrete dSCA

A concrete dSCA implementation of the Form 2 abstract dSCA solution to the GRCP will be considered in this chapter. The model will be cycle consistent, and therefore will have a system comprising of one main module which manipulates a tuple of length Max_N. There will also be the explicit definition of the single program counter module. This situation is diagramtically shown in Figure 8.10.

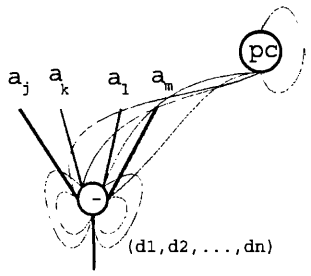

Figure 8.10: Concrete dSCA Physical GRCP Solution

It has already been stated in this thesis that there are a range of tuple management functions that could be selected, and in this exposition the indexed-array approach will be adopted.

Wiring Functions

All wirings will either be to module 1, the programme counter module, or to an input (or by the nature of an cdSCA will not be connected). The varying β-wiring functions are defined in Table 8.14 (note that this table does not show arguments 0 and 1). In comparison to the Form 2 abstract dSCA the index of arguments has been duly shifted by one to accommodate that concrete dSCA definition of the 0^{th} argument being from the program counter and the 1^{st} argument from the module itself.

$\beta_0(1,2) = 1$	$\beta_7(1,2) = \omega$	$\beta_{14}(1,2) = 9$	$\beta_{21}(1,2) = 1$	$\beta_{28}(1,2) = 1$
$\beta_0(1,3) = 1$	$\beta_7(1,3) = \omega$	$\beta_{14}(1,3) = 1$	$\beta_{21}(1,3) = 1$	$\beta_{28}(1,3) = 2$
$\beta_0(1,4) = 1$	$\beta_7(1,4) = \omega$	$\beta_{14}(1,4) = \omega$	$\beta_{21}(1,4) = \omega$	$\beta_{28}(1,4) = \omega$
$\beta_1(1,2) = pc$	$\beta_8(1,2) = \omega$	$\beta_{15}(1,2) = \omega$	$\beta_{22}(1,2) = 1$	$\beta_{29}(1,2) = \omega$
$\beta_1(1,3) = pc$	$\beta_8(1,3) = \omega$	$\beta_{15}(1,3) = \omega$	$\beta_{22}(1,3) = 1$	$\beta_{29}(1,3) = \omega$
$\beta_1(1,4) = \omega$	$\beta_8(1,4) = \omega$	$\beta_{15}(1,4) = \omega$	$\beta_{22}(1,4) = \omega$	$\beta_{29}(1,4) = \omega$
$\beta_2(1,2) = \omega$	$\beta_9(1,2) = 1$	$\beta_{16}(1,2) = \omega$	$\beta_{23}(1,2) = 1$	$\beta_{30}(1,2) = 3$
$\beta_2(1,3) = \omega$	$\beta_9(1,3) = 1$	$\beta_{16}(1,3) = \omega$	$\beta_{23}(1,3) = 1$	$\beta_{30}(1,3) = 4$
$\beta_2(1,4) = \omega$	$\beta_9(1,4) = \omega$	$\beta_{16}(1,4) = \omega$	$\beta_{23}(1,4) = \omega$	$\beta_{30}(1,4) = \omega$
$\beta_3(1,2) = 1$	$\beta_{10}(1,2) = 9$	$\beta_{17}(1,2) = \omega$	$\beta_{24}(1,2) = 1$	$\beta_{31}(1,2) = \omega$
$\beta_3(1,3) = 1$	$\beta_{10}(1,3) = 1$	$\beta_{17}(1,3) = \omega$	$\beta_{24}(1,3) = 1$	$\beta_{31}(1,3) = \omega$
$\beta_3(1,4) = 1$	$\beta_{10}(1,4) = \omega$	$\beta_{17}(1,4) = \omega$	$\beta_{24}(1,4) = \omega$	$\beta_{31}(1,4) = \omega$
$\beta_4(1,2) = 1$	$\beta_{11}(1,2) = 1$	$\beta_{18}(1,2) = \omega$	$\beta_{25}(1,2) = 1$	$\beta_{32}(1,2) = 5$
$\beta_4(1,3) = 1$	$\beta_{11}(1,3) = 1$	$\beta_{18}(1,3) = \omega$	$\beta_{25}(1,3) = 1$	$\beta_{32}(1,3) = 6$
$\beta_4(1,4) = \omega$	$\beta_{11}(1,4) = \omega$	$\beta_{18}(1,4) = \omega$	$\beta_{25}(1,4) = \omega$	$\beta_{32}(1,4) = \omega$
$\beta_5(1,2) = 1$	$\beta_{12}(1,2) = 9$	$\beta_{19}(1,2) = \omega$	$\beta_{26}(1,2) = 1$	$\beta_{33}(1,2) = \omega$
$\beta_5(1,3) = 1$	$\beta_{12}(1,3) = 1$	$\beta_{19}(1,3) = \omega$	$\beta_{26}(1,3) = 1$	$\beta_{33}(1,3) = \omega$
$\beta_5(1,4) = \omega$	$\beta_{12}(1,4) = \omega$	$\beta_{19}(1,4) = \omega$	$\beta_{26}(1,4) = \omega$	$\beta_{33}(1,4) = \omega$
$\beta_6(1,2) = 1$	$\beta_{13}(1,2) = 1$	$\beta_{20}(1,2) = \omega$	$\beta_{27}(1,2) = 1$	$\beta_{34}(1,2) = 7$
$\beta_6(1,3) = 1$	$\beta_{13}(1,3) = 1$	$\beta_{20}(1,3) = \omega$	$\beta_{27}(1,3) = 1$	$\beta_{34}(1,3) = 8$
$\beta_6(1,4) = \omega$	$\beta_{13}(1,4) = \omega$	$\beta_{20}(1,4) = \omega$	$\beta_{27}(1,4) = \omega$	$\beta_{34}(1,4) = \omega$
		$\beta_{35}(1,2) = \omega$	$\beta_{35}(1,3) = \omega$	$\beta_{35}(1,4) = \omega$

Table 8.14: $\beta-$ Wiring Functions for cdSCA

It is also the case that for $0 \leq pc\_val \leq Max_N - 1$:

- the 0^{th} input for each module is wired to the program counter:

$$\beta_{pc\_val}(1,0) = pc,$$

- the 1^{st} input for each module is wired to the module itself:

$$\beta_{pc\_val}(1,1) = 1,$$

- and that for the program counter the definition is:

$$\beta_{pc\_val}(pc,0) = pc.$$

For the γ-wiring functions, indicating whether a module is linked to a module, source or unconnected, it is also the case that:

- the 0^{th} input for each module is wired to the program counter:

$$\gamma_y(1,0) = M \; for \; 1 \leq y \leq Max_N,$$

- the 1^{st} input for each module is wired to the module itself:

$$\gamma_y(1,1) = M \; for \; 1 \leq y \leq Max_N,$$

- and that for the program counter the definition is:

$$\gamma_y(pc,0) = M \; for \; 0 \leq y \leq Max_N$$

The remainder of the γ-wiring functions are defined in Table 8.15. Again these too a close resemblance to those given for the Form 2 abstract dSCA but with a corresponding shift in argument index to accommodate the above definitions for arguments 0 and 1.

$\gamma_0(1,2) = M$	$\gamma_7(1,2) = U$	$\gamma_{14}(1,2) = S$	$\gamma_{21}(1,2) = M$	$\gamma_{28}(1,2) = S$
$\gamma_0(1,3) = M$	$\gamma_7(1,3) = U$	$\gamma_{14}(1,3) = M$	$\gamma_{21}(1,3) = M$	$\gamma_{28}(1,3) = S$
$\gamma_0(1,4) = M$	$\gamma_7(1,4) = U$	$\gamma_{14}(1,4) = U$	$\gamma_{21}(1,4) = U$	$\gamma_{28}(1,4) = U$
$\gamma_1(1,2) = pc$	$\gamma_8(1,2) = U$	$\gamma_{15}(1,2) = U$	$\gamma_{22}(1,2) = M$	$\gamma_{29}(1,2) = U$
$\gamma_1(1,3) = pc$	$\gamma_8(1,3) = U$	$\gamma_{15}(1,3) = U$	$\gamma_{22}(1,3) = M$	$\gamma_{29}(1,3) = U$
$\gamma_1(1,4) = U$	$\gamma_8(1,4) = U$	$\gamma_{15}(1,4) = U$	$\gamma_{22}(1,4) = U$	$\gamma_{29}(1,4) = U$
$\gamma_2(1,2) = U$	$\gamma_9(1,2) = M$	$\gamma_{16}(1,2) = U$	$\gamma_{23}(1,2) = M$	$\gamma_{30}(1,2) = S$
$\gamma_2(1,3) = U$	$\gamma_9(1,3) = M$	$\gamma_{16}(1,3) = U$	$\gamma_{23}(1,3) = M$	$\gamma_{30}(1,3) = S$
$\gamma_2(1,4) = U$	$\gamma_9(1,4) = U$	$\gamma_{16}(1,4) = U$	$\gamma_{23}(1,4) = U$	$\gamma_{30}(1,4) = U$
$\gamma_3(1,2) = M$	$\gamma_{10}(1,2) = S$	$\gamma_{17}(1,2) = U$	$\gamma_{24}(1,2) = M$	$\gamma_{31}(1,2) = U$
$\gamma_3(1,3) = M$	$\gamma_{10}(1,3) = M$	$\gamma_{17}(1,3) = U$	$\gamma_{24}(1,3) = M$	$\gamma_{31}(1,3) = U$
$\gamma_3(1,4) = M$	$\gamma_{10}(1,4) = U$	$\gamma_{17}(1,4) = U$	$\gamma_{24}(1,4) = U$	$\gamma_{31}(1,4) = U$
$\gamma_4(1,2) = M$	$\gamma_{11}(1,2) = M$	$\gamma_{18}(1,2) = U$	$\gamma_{25}(1,2) = M$	$\gamma_{32}(1,2) = S$
$\gamma_4(1,3) = M$	$\gamma_{11}(1,3) = M$	$\gamma_{18}(1,3) = U$	$\gamma_{25}(1,3) = M$	$\gamma_{32}(1,3) = S$
$\gamma_4(1,4) = U$	$\gamma_{11}(1,4) = U$	$\gamma_{18}(1,4) = U$	$\gamma_{25}(1,4) = U$	$\gamma_{32}(1,4) = U$
$\gamma_5(1,2) = M$	$\gamma_{12}(1,2) = S$	$\gamma_{19}(1,2) = U$	$\gamma_{26}(1,2) = M$	$\gamma_{33}(1,2) = U$
$\gamma_5(1,3) = M$	$\gamma_{12}(1,3) = M$	$\gamma_{19}(1,3) = U$	$\gamma_{26}(1,3) = M$	$\gamma_{33}(1,3) = U$
$\gamma_5(1,4) = U$	$\gamma_{12}(1,4) = U$	$\gamma_{19}(1,4) = U$	$\gamma_{26}(1,4) = U$	$\gamma_{33}(1,4) = U$
$\gamma_6(1,2) = M$	$\gamma_{13}(1,2) = M$	$\gamma_{20}(1,2) = U$	$\gamma_{27}(1,2) = M$	$\gamma_{34}(1,2) = S$
$\gamma_6(1,3) = M$	$\gamma_{13}(1,3) = M$	$\gamma_{20}(1,3) = U$	$\gamma_{27}(1,3) = M$	$\gamma_{34}(1,3) = S$
$\gamma_6(1,4) = U$	$\gamma_{13}(1,4) = U$	$\gamma_{20}(1,4) = U$	$\gamma_{27}(1,4) = U$	$\gamma_{34}(1,4) = U$
		$\gamma_{35}(1,2) = U$	$\gamma_{35}(1,3) = U$	$\gamma_{35}(1,4) = U$

Table 8.15: $\gamma-$ Wiring Functions for Form 2 acvSCA

Delay Functions

For a concrete dSCA the delay functions are always the unit delay as all look-backs over time are now captured within the tuple. It is therefore defined that for $0 \leq pc\_val \leq Max_N - 1$, $i = 1$ and $0 \leq j \leq 4$:

$$\delta_{1,j,pc}(t, a, x) = t - 1$$

The program counter module has only one input, from itself, and this is also by definition always unit delay, so it is appropriate to define for $0 \leq pc\_val \leq Max_N - 1$:

$$\delta_{pc,0,pc\_val}(t, a, x) = t - 1$$

Value Function: Initial State

The Initial State definitions for the Value Functions for module m_1 of the concrete dSCA solution to the GRCP needs to reflect the first Max_N, or 36, initial states. It is known from the definition of concrete dSCAs that only the values at time $t = 0$ and $t = 35$ are of use to computation and comparison of correctness. It is appropriate to define, for $t = 0$:

$$V_1(pc\_val, a, x) = \begin{pmatrix} stay, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u & & & & \end{pmatrix}$$

and for $1 \leq pc\_val \leq Max_N - 2$ that:

$$V_1(pc\_val, a, x) = \begin{pmatrix} stay, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u & & & & \end{pmatrix}$$

The final definition, for $t = 35$, is:

$$V_1(35, a, x) = \begin{pmatrix} stay, & true, & stay, & up, & true, & false, & false, & down, \\ up, & true, & true, & false, & false, & false, & true, & false, \\ 90, & true, & 0, & true, & 0, & false, & false, & false, \\ false, & false, & false, & false, & 0, & 0, & 0, & 0, \\ 0, & 0, & 0, & 0 & & & & \end{pmatrix}$$

The Initial State definition of the Value Function for module m_{pc} is given in accordance with the definition of concrete dSCAs as:

$$\begin{aligned} &V_0(0, a, x) = 1 \\ &V_0(1, a, x) = 2 \\ &\vdots \\ &V_0(Max_N - 2, a, x) = Max_N - 1 \\ &V_0(Max_N - 1, a, x) = 0 \end{aligned}$$

Value Functions: State Transition

The state transition definition for the value function for the single module reflects the Max_N cases that need to be covered, and also include the tuple management operations. The definition of the program counter module is:

$$V_{pc}(t+1,a,x) = \begin{cases} mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t,a,x) = 0 \\ \vdots & \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t,a,x) = Max_N - 1 \end{cases}$$

with the definition of module 1 shown overleaf.

$$
V_1(t+1,a,x) = \begin{cases}
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), cond\,(\Pi_1^{35}(V_1(t,a,x)), \Pi_2^{35}(V_1(t,a,x)), \Pi_3^{35}(V_1(t,a,x)))) & \text{if } V_{pc}(t,a,x) = 0 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), or\,(\Pi_4^{35}(V_1(t,a,x)), \Pi_5^{35}(V_1(t,a,x)))) & \text{if } V_{pc}(t,a,x) = 1 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), start) & \text{if } V_{pc}(t,a,x) = 2 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), cond\,(\Pi_6^{35}(V_1(t,a,x)), \Pi_7^{35}(V_1(t,a,x)), \Pi_8^{35}(V_1(t,a,x)))) & \text{if } V_{pc}(t,a,x) = 3 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), and\,\langle\Pi_9^{35}(V_1(t,a,x)), \Pi_{10}^{35}(V_1(t,a,x))\rangle) & \text{if } V_{pc}(t,a,x) = 4 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), and\,\langle\Pi_{11}^{35}(V_1(t,a,x)), \Pi_{12}^{35}(V_1(t,a,x))\rangle) & \text{if } V_{pc}(t,a,x) = 5 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), and\,\langle\Pi_{13}^{35}(V_1(t,a,x)), \Pi_{14}^{35}(V_1(t,a,x))\rangle) & \text{if } V_{pc}(t,a,x) = 6 \\
\Upsilon\,(pc, V_1(t,a,x), down) & \text{if } V_{pc}(t,a,x) = 7 \\
\Upsilon\,(pc, V_1(t,a,x), up) & \text{if } V_{pc}(t,a,x) = 8 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), eq\,(\Pi_{21}^{35}(V_1(t,a,x)), \Pi_{15}^{35}(V_1(t,a,x)))) & \text{if } V_{pc}(t,a,x) = 9 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), eq\,\langle a_9(t), \Pi_{16}^{35}(V_1(t,a,x))\rangle) & \text{if } V_{pc}(t,a,x) = 10 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), eq\,\langle\Pi_{21}^{35}(V_1(t,a,x)), \Pi_{17}^{35}(V_1(t,a,x))\rangle) & \text{if } V_{pc}(t,a,x) = 11 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), eq\,\langle a_9(t), \Pi_{18}^{35}(V_1(t,a,x))\rangle) & \text{if } V_{pc}(t,a,x) = 12 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), eq\,\langle\Pi_{21}^{35}(V_1(t,a,x)), \Pi_{19}^{35}(V_1(t,a,x))\rangle) & \text{if } V_{pc}(t,a,x) = 13 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), gt\,\langle a_9(t), \Pi_{20}^{35}(V_1(t,a,x))\rangle) & \text{if } V_{pc}(t,a,x) = 14 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), false) & \text{if } V_{pc}(t,a,x) = 15 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), 90) & \text{if } V_{pc}(t,a,x) = 16 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), true) & \text{if } V_{pc}(t,a,x) = 17 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), 0) & \text{if } V_{pc}(t,a,x) = 18 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), true) & \text{if } V_{pc}(t,a,x) = 19 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), 0) & \text{if } V_{pc}(t,a,x) = 20 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), or\,(\Pi_1^{23}(V_1(t,a,x)), \Pi_1^{24}(V_1(t,a,x)))) & \text{if } V_{pc}(t,a,x) = 21 \\
\Upsilon\,\langle V_{pc}(t,a,x), V_1(t,a,x), or\,\langle\Pi_1^{24}(V_1(t,a,x)), \Pi_1^{25}(V_1(t,a,x))\rangle\rangle & \text{if } V_{pc}(t,a,x) = 22 \\
\Upsilon\,\langle V_{pc}(t,a,x), V_1(t,a,x), or\,\langle\Pi_1^{26}(V_1(t,a,x)), \Pi_1^{27}(V_1(t,a,x))\rangle\rangle & \text{if } V_{pc}(t,a,x) = 23 \\
\Upsilon\,\langle V_{pc}(t,a,x), V_1(t,a,x), gt\,\langle\Pi_1^{28}(V_1(t,a,x)), \Pi_1^{29}(V_1(t,a,x))\rangle\rangle & \text{if } V_{pc}(t,a,x) = 24 \\
\Upsilon\,\langle V_{pc}(t,a,x), V_1(t,a,x), gt\,\langle\Pi_1^{30}(V_1(t,a,x)), \Pi_1^{31}(V_1(t,a,x))\rangle\rangle & \text{if } V_{pc}(t,a,x) = 25 \\
\Upsilon\,\langle V_{pc}(t,a,x), V_1(t,a,x), gt\,\langle\Pi_1^{32}(V_1(t,a,x)), \Pi_1^{33}(V_1(t,a,x))\rangle\rangle & \text{if } V_{pc}(t,a,x) = 26 \\
\Upsilon\,\langle V_{pc}(t,a,x), V_1(t,a,x), gt\,\langle\Pi_1^{34}(V_1(t,a,x)), \Pi_1^{35}(V_1(t,a,x))\rangle\rangle & \text{if } V_{pc}(t,a,x) = 27 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), sub\,\langle\, a_1(t), a_2(t)\,\rangle) & \text{if } V_{pc}(t,a,x) = 28 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), 0) & \text{if } V_{pc}(t,a,x) = 29 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), sub\,\langle\, a_3(t), a_4(t)\,\rangle) & \text{if } V_{pc}(t,a,x) = 30 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), 0) & \text{if } V_{pc}(t,a,x) = 31 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), sub\,\langle\, a_5(t), a_6(t)\,\rangle) & \text{if } V_{pc}(t,a,x) = 32 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), 0) & \text{if } V_{pc}(t,a,x) = 33 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), sub\,\langle\, a_7(t), a_8(t)\,\rangle) & \text{if } V_{pc}(t,a,x) = 34 \\
\Upsilon\,(V_{pc}(t,a,x), V_1(t,a,x), 0) & \text{if } V_{pc}(t,a,x) = 35
\end{cases}
$$

Network output: V_{out}

Network output will be from module 1, i.e. :

$$V_{out}(t, a, x) = V_1(t, a, x)$$

The correct value would have to be projects out if a comparison to an abstract dSCA or original SCA is to be made. The execution order indicates that a new answer will be available every 36^{th} clock cycle starting at time $t = 0$, and due to the tuple management operation, the value to be compared will be held in the 0^{th} element of the array.

8.4.1 Correctness

The complete definition of the concrete dSCA is captured in Appendix E.

By inspection and application of the mapping of modules (from the execution order) it can be seen that the concrete dSCA represents the Form 2 abstract dSCA. Consider the value of V_{out} of the Form 1 dSCA at time $t = 0$, from Table 8.6 it can be seen that it is equal to *stay*. The equivalent value in the Form 2 abstract dSCA is provided as the 0^{th} element in the tuple of V_1. Recall that this tuple is defined as:

$$V_1(pc\_val, a, x) = \begin{pmatrix} stay, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u, & ,u, & u, & u, & u, \\ u, & u, & u, & u & & & & \end{pmatrix}$$

it can be seen that the 0^{th} element is *stay*, the same as for the Form 1 abstract dSCA.

A similar process can be applied for other times, notably important results will be produced every 36 clock cycles.

8.5 Concluding Comments

Four solutions to the GRCP example have been given, and this chapter has provided a discussion relating to the correctness of each model with respect to the "previous"

model.

It is now claimed that a SCA can be seen as the mathematical representation of a computation, and that the concrete dSCA is a mathematical model of the computing device that will implement the computation.

It is further claimed that the use of abstract dSCAs supports the mathematical transformation of a SCA to a concrete dSCA using a number of mappings. The challenge taken forward into the next part of this thesis is how to mathematically define these mappings and transformations algebraically.

8.6 Sources

This chapter is all my own work.

Part III

Transformations

Chapter 9

Concept of SCA Transformations

9.1 Introduction

In the previous chapters it has been shown how the solution to the GRCP could be represented as: an original SCA, two different abstract dSCAs and a concrete dSCA. The correctness of each of those models has also been discussed. The reader may have noticed that one model has, in some sense, been derived from a previous model. For example the concrete dSCA is an implementation of the Form 2 abstract dSCA, which itself is an implementation of the Form 1 abstract dSCA, which can be seen to be an implementation of the original SCA implementation. Given such a hierarchy, this thesis now proposes that there are mechanical methods to transform from one model to another. Future discussions are restricted to the following transformations:

1. A k-module SCA network to an abstract dSCA network with a defining shape of $\nabla = (k, 1)$;

2. An abstract dSCA network with defining shape of $\nabla = (k, 1)$ to an abstract dSCA network with defining shape of $\nabla = (n, k)$; and

3. An **abstract** dSCA network with defining shape of $\nabla = (n, k)$ to a **concrete** dSCA with defining shape of $\nabla = (n, k)$.

Each transformation will be defined algebraically.

Such a series of transformations is analogous to refinement steps commonly found in the development of safety related systems (though refinement steps are usually applied to the transformation of a mathematical specification to a program). Program transformation is a large field, which this thesis does not intend to delve into in depth - the reader is pointed to Stephenson's PhD thesis ([Ste95]) which covers a wide range of program transformation in her literature study.

In this introduction reference has been made to the various SCAs existing in a hierarchy. Poole, Holden and Tucker ([PHT98]) have previously considered hierarchies of Spatially Extended Systems, of which SCAs are a form, and this provides a useful alternative method for consideration of correctness in addition to the Type I and Type II notion discussed previously in this thesis. They set out to demonstrate how one SCA can abstract, approximate, or implement another SCA, and introduced the Integrative Hierarchy Problem:

> "**Integrative hierarchy problem**: Develop a mathematical theory that is able to relate and integrate different mathematical models at different levels of abstraction" ([PHT98])

Poole, Holden and Tucker argue that to compare two SCAs, the following must be considered:

- spaces;
- clocks;
- global states; and
- input streams.

Consider two networks, N_1 and N_2 with each network having non-empty sets I_1 and I_2 of modules, computing with respect to clocks T_1 and T_2, sets In_1 and In_2 of inputs. The previous discussions on SCAs has discussed the channels between modules, but for hierarchies we formally introduce sets Ch_1 and Ch_2 to explicitly refer to channels in network N_1 and N_2 respectively. The networks will therefore have the sets $M_{A1}^{Ch_1}$ and $M_{A2}^{Ch_2}$ of initial states and sets $[T_1 \rightarrow M_{A1}]^{In_1}$ and $[T_2 \rightarrow M_{A2}]^{In_2}$ of input streams. It is intended that the behaviour of the network N_2 is an abstraction of the behaviour of network N_1 then it should be possible to construct the necessary mappings.

Spaces

For our purposes, the space of an SCA is analogous to the modules within networks. An SCA respacing function can therefore be introduced that maps modules within N_1 to N_2:

$$\pi : I_1 \rightarrow I_2$$

This mapping is a surjective function, with the intention that each module $i \in I_1$ in network N_1 is abstracted by the module $\pi(i) \in I_2$ in network N_2.

Clocks

Mapping between two clocks, T_1 and T_2, is achieved by the introduction of retimings. Retimings were introduced by Harman and Tucker in [HT89] and [HT90]; and these should not be confused with the concept of retimings introduced by Leiserson and Saxe (whose retimings relate to improving the timing behaviour of a circuit by reallocation of registers - see [LS91] for details).

A clock is defined to be an algebra consisting of a set of natural numbers, the constant 0 and the successor operation $t + 1$. If R and T are two such clocks, then a retiming $\lambda : T \rightarrow R$ is a mapping between them capturing the concept that $\lambda(t)$ is

the time cycle on R that corresponds with the time cycle $t \in T$. This is demonstrated in figure 9.1.

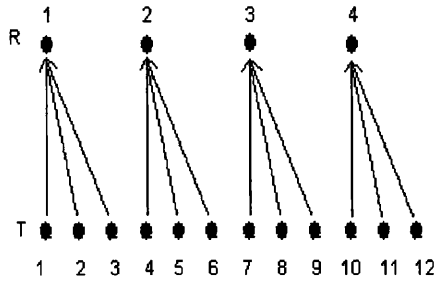

Figure 9.1: Retiming

More formally, let $T = (T, 0, t + 1)$ and $R = (R, 0, r + 1)$ be two clock algebras, then if each clock cycle in R corresponds to more than one clock cycle in T then R is at a higher abstraction than T. Further, $\lambda : T \rightarrow R$ is a retiming from T to R, iff:

1. $\lambda(0) = 0$;
2. λ is surjective, that is to say that for all $r \in R$ there is a $t \in T$ such that $\lambda(t) = r$; and
3. λ is monotonic so that for any $t, t' \in T$ if $t \leq t'$ then $\lambda(t) \leq \lambda(t')$.

The set of all retimings from T to R is denoted as $Ret(T, R)$. A few useful operations relating to retimings are now discussed. $\bar{\lambda}$ is known as the immersion of a retiming $\lambda \in Ret(T, R)$ and is defined as:

$$\bar{\lambda}(r) = least\ t \in T\ such\ that\ \lambda(t) = r$$

$Start_\lambda$ identifies the first clock cycle in the group that could be retimed to a value in the other clock. For example $Start_\lambda(5)$ in Figure 9.1 would be 4. It is defined as applying the immersion to the result of the retiming of the clock under consideration:

$$Start_\lambda = (\bar{\lambda}\lambda)$$

Global States

The global state of an SCA at time $t \in T$ is defined as the set of values held by all the channels at time $t \in T$. There exists a global state abstraction mapping, ϕ of the form:

$$\phi : M_{A1}^{Ch_1} \rightarrow M_{A2}^{Ch_2}$$

where the intention is that a global state $s \in M_{A1}^{Ch_1}$ of SCA N_1 is abstracted in SCA N_2 by the state $\phi(s) \in M_{A2}^{Ch_2}$.

It is sometimes necessary to provide a data abstraction function within the global state map if the algebras in each network are not the same. Poole, Holden and Tucker demonstrate the use of this when considering the hierarchy between two systolic convolvers, the first working on an algebra representing bits, and an abstraction using the carrier set M_A.

Input Streams

Input streams for network N_1 can be mapped to those in N_2 by means of the stream abstraction function:

$$\theta : [T_1 \rightarrow M_{A1}]^{In_1} \rightarrow [T_2 \rightarrow M_{A2}]^{In_2}$$

with the intention that streams $a \in [T_1 \rightarrow M_{A1}]^{In_1}$ for N_1 are abstracted in N_2 by $\theta(a) \in [T_2 \rightarrow M_{A2}]^{In_2}$.

SCA Equivalence

To show the equivalence of two SCAs in a hierarchy, it is therefore necessary to show the following diagram commutes:

$$\begin{array}{ccccc}
T_2 & \times [T_2 \rightarrow M_{A2}]^{In_2} \times & M_{A2}^{Ch_2} & \xrightarrow{V_2} & M_{A2}^{Ch_2} \\
\uparrow \lambda & \uparrow \theta & \uparrow \phi & & \uparrow \phi \\
Start_\lambda & \times [T_1 \rightarrow M_{A1}]^{In_1} \times & M_{A1}^{Ch_1} & \xrightarrow{V_1} & M_{A1}^{Ch_1}
\end{array}$$

Correctness within a hierarchy would therefore allow the syntactic demonstration of the correctness of a model M_2 against a model M_1 - if M_1 is already shown to be correct with respect to some specification S, then showing M_1 and M_2 are in a hierarchy, such that π, λ, ϕ and θ exist and a diagram for the above commutes, would demonstrate that M_2 is also correct with respect to S. Consider the SCA implementation of the GRCP to be M_1, the Form 1 abstract dSCA implementation to be M_2, the Form 2 abstract dSCA implementation to be M_3 and the concrete dSCA implementation to be M_4, then if the notation $A \rhd B$ is used to mean that B is a correct implementation of A within a hierarchy, then the correctness of the concrete dSCA with respect to the specification S can be asserted iff:

$$S \rhd M_1 \rhd M_2 \rhd M_3 \rhd M_4$$

In the exposition of transformations in this thesis a restriction is placed that the machine algebra M_A will be consistent across the SCA models. The implication of this is that there will not be an investigation of the alterations of datatypes across the models, which in turn may affect timings and mappings used.

Before discussing the transformations a discussion is provided next on a number of fundamental algebras that will be used in the definition of the SCAs and the transformations. Finally, Chapters 10, 11 and 12 describe the transformations in detail and include a walk through of how the GRCP example is transformed from an SCA to a concrete dSCA.

9.2 Fundamental Algebra Specifications

There are four types of fundamental algebra specifications used in this thesis:

1. Synchronous Concurrent Algorithms (SCA, adSCA and cdSCA)

2. Machine Algebra

3. Lists

4. Forms of equations: Value Functions, wiring and delay functions, etc.

The important elements of each specification are discussed in the next set of sections, and full definitions of relevant specifications are provided in Appendix A.

9.2.1 SCA Algebraic Specification

As defined in Chapter 5.7.2, the specification of an SCA will be written as

Begin	
Specification	SCA_Name
Import	T, M_A
Sorts	[]
Constants Symbols	[]
VF Function Names	$V_i : T \times [T \rightarrow M_A]^n \times M_A^k \rightarrow M_A,$
γFunction Names	$\gamma(i,j) : \mathbb{N}_k \times N \rightarrow \mathbb{N}_k,$
βFunction Names	$\beta(i,j) : \mathbb{N}_k \times N \rightarrow \{S, M\},$
δFunction Names	$\delta_{i,j} : T \times [T \rightarrow M_A]^n \times M_A^k \rightarrow T,$
IV Equations	$V_1(0,a,x) = x_0,$
	$\ldots,$
	$V_m(0,a,x) = x_m$
ST Equations	$V_1(t+1,a,x) = f_i,$
	$\ldots,$
	$V_m(t+1,a,x) = f_i$
γEquations	$\gamma(0,0) = x,$
	$\ldots,$
	$\gamma(i,j) = x$
βEquations	$\beta(i,j) = L,$
	$\ldots,$
	$\beta(i,j) = L$
δEquations	$\delta_{i,j}(t+1,a,x) = t,$
	$\ldots,$
	$\delta_{i,j}(t+1,a,x) = t$
End	

which of course is a convenient way of writing:

$$\left(\begin{array}{l}
\mathbb{N}, \mathbb{B}, \\
0, true, false, u \\
succ : \mathbb{N} \rightarrow \mathbb{N}, add : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}, cond : \mathbb{B} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}, \ldots, \\
V_i : T \times [T \rightarrow M_A]^n \times M_A^k \rightarrow M_A, \\
\gamma(i,j) : \mathbb{N}_k \times N \rightarrow \mathbb{N}_k, \beta(i,j) : \mathbb{N}_k \times N \rightarrow \{S, M\}, \\
\delta_{i,j} : T \times [T \rightarrow M_A]^n \times M_A^k \rightarrow T, \\
succ(n) = n+1, add(succ(a), b) = add(a, succ(b)), \ldots, \\
\vdots \\
V_0(0, a, x) = 5, \ldots, \\
V_0(t+1, a, x) = add(V_1(t, a, x), 2), \ldots, \\
\gamma(1,0) = M, \ldots, \\
\beta(1,0) = 2, \ldots, \\
\delta_{1,0}(t, a, x) = t - 1, \ldots
\end{array}\right)$$

To enable the construction of such specifications and provide access to the constituent parts, the *SCAAlgebra* specification is provided. It has one construction operation, $CreateSCA$, which takes enough arguments to create a representation of an SCA in the algebraic notation. There are also 13 decomposition operations that provide access to the various components of an SCA. The constructor operation, $CreateSCA$, is given as:

$$\begin{aligned}
CreateSCA: \quad & Name \times ImpList \times SortList \times ConsList \times \\
& VFOpList \times \gamma OpList \times \beta OpList \times \delta OpList \times \\
& IVEqList \times STEqList \times \gamma EqList \times \beta EqList \times \delta EqList \rightarrow SCA
\end{aligned}$$

and has the following definition:

$$CreateSCA\left(\begin{array}{l}
name, \\
import, \\
sorts, \\
constants, \\
opsVF, \\
ops\gamma, \\
ops\beta, \\
ops\delta, \\
eqsVFIV, \\
eqsVFST, \\
eqs\gamma, \\
eqs\beta, \\
eqs\delta,
\end{array}\right) = \left(\begin{array}{ll}
\textbf{Begin} & \\
 & \textbf{Specification } name \\
 & \textbf{Import } import \\
 & \textbf{Sorts } sorts \\
 & \textbf{Constant Symbols } constants \\
 & \textbf{VF Function Names } opsVF \\
 & \gamma \textbf{ Function Names } ops\gamma \\
 & \beta \textbf{ Function Names } ops\beta \\
 & \delta \textbf{ Function Names } ops \\
 & \textbf{IV Equations } eqsVFIV \\
 & \textbf{ST Equations } eqsVFST \\
 & \gamma \textbf{ Equations } eqs\gamma \\
 & \beta \textbf{ Equations } eqs\beta \\
 & \delta \textbf{ Equations } eqs\delta \\
\textbf{End} &
\end{array}\right)$$

Decomposition operations provide access to the component parts of an SCA specification, e.g. the γ-wiring functions. As an example, consider the **GetβEqs** operation whose purpose is to return the list of imported specifications in the definition of an SCA. It is given as:

$$Get\beta Eqs : SCAAlgebra \rightarrow \beta SCAEqList$$

and defined as:

$$Get\beta Eqs \left(\begin{array}{l} \textbf{Begin} \\ \quad \textbf{Specification } name \\ \quad \textbf{Import } import \\ \quad \textbf{Sorts } sorts \\ \quad \textbf{Constant Symbols } constants \\ \quad \textbf{VF Function Names } opsVF \\ \quad \gamma \textbf{ Function Names } ops\gamma \\ \quad \beta \textbf{ Function Names } ops\beta \\ \quad \delta \textbf{ Function Names } ops \\ \quad \textbf{IV Equations } eqsVFIV \\ \quad \textbf{ST Equations } eqsVFST \\ \quad \gamma \textbf{ Equations } eqs\gamma \\ \quad \beta \textbf{ Equations } eqs\beta \\ \quad \delta \textbf{ Equations } eqs\delta \\ \textbf{End} \end{array} \right) = eqs\beta$$

Similar operations are defined to allow access to all constituent parts of an SCA and the complete definition of the SCA Manage specification is given in Appendix A.1.

Since abstract and concrete dSCAs are syntactic extensions to SCAs then the specifications for those will be defined in a similar manner. For brevity, this thesis will not define these specifications.

9.2.2 Machine Algebra Specification

The Machine Algebra, denoted as M_A, is the carrier algebra A so far used in the definition of Synchronous Concurrent Algorithms. It is renamed to focus the reader

on the objective of the thesis, that of 'compiling' an SCA to a target machine architecture. Operations within M_A are said to be atomic, i.e. they cannot be further subdivided with relation to the level of abstraction currently under consideration.

M_A includes operations that depend upon the underlying machine that transformations are targeted at. For the purposes of this thesis a target machine that can perform simple mathematical and logical operations over the set of natural numbers and booleans, as shown in Table 9.1 (where the usual meaning is applied to the operations) will be used.

Natural	Boolean	Combined
add	or	eq
sub	and	lt
mult	not	gt
div		cond

Table 9.1: Operations in M_A

It is important that operations in M_A can handle the undefined element, u, in any of its arguments. The result of an operation where any argument is u will be u, even in the case of boolean operations, e.g. the OR operation where it might be expected that a u for one argument and a *true* for the other would result in *true*, will result in u. This is done because of the field in which this thesis is positioned where an undefined value would be erroneous and thus the undefined value should be propagated so it can be handled outside the computation system. M_A is fully defined in Appendix A.2.

9.2.3 List Algebra Specifications

The SCA specifications contains many lists, each of which will have a corresponding list specifications. Lists are required for the following:

1. Imported types (ImpList).

2. Sorts (SortList).
3. Constants (ConsList).
4. Value Function operation definitions (VFOpList).
5. γ-wiring function operation definitions ($\gamma OpList$).
6. β-wiring function operation definitions ($\beta OpList$).
7. Delay function operation definitions ($\delta OpList$).
8. Initial Value equations ($ISVEqList$ and $dSCAISVEqList$).
9. State Transition equations ($STVEqList$ and $dSCASTVEqList$).
10. γ-wiring function equations ($\gamma SCAEqList$ and $\gamma dSCAEqList$).
11. β-wiring function equations ($\beta SCAEqList$ and $\beta dSCAEqList$).
12. Delay function equations ($\delta SCAEqList$ and $\delta dSCAEqList$).
13. Project function equations ($ProjEqList$).
14. Mapping function equations ($MapList$).

Each specification, is similarly defined, with the main difference the definition of the *GetEl* operation. *GetEl* returns a particular elements from a list of equations from a defined position. All the specifications have standard head and tail operations.

This thesis will be mainly concerned with Equation Lists in an specification, rather than, for example, the operation or constant lists. The $STVEqList$ specification is now discussed in detail, and then a discussion on the differences of the *GetEl* operation for various other equation list specifications will be performed.

SCA State Transition Equation List Specification

Within the *STEqList* specification a single composition function is provided, _, _, that enables the recursive creation of a lists of elements. Three decomposition operations, *hd*, *tl* and *GetEl* are provided that extract the head of a list, the tail of a list and an element from a particular position in a list respectively. The empty list will always be represented by the constant []

To create a list the specification provides one infix operation:

$$\_, \_ : STVEquation \times STVEqList \rightarrow STVEqList$$

Thus, $(a, [])$ and $(a, b, c, [])$ are both equation lists.

For the decomposition operations, consider the following example list:

$$\begin{aligned} &V_1(t, a, x) = add(1, 7), \\ &V_2(t, a, x) = mult(3, 6), \\ &\vdots, \\ &V_n(t, a, x) = sub(6.3), \\ &[] \end{aligned}$$

The *hd* operation returns the head of a list, it is given as:

$$hd : STVEqList \rightarrow STVEquation$$

and is therefore be defined as:

$$\begin{aligned} hd\,([]) &= [] \\ hd\,(a, as) &= a \end{aligned}$$

such that the *hd* of the above list is:

$$V_1(t, a, x) = add(1, 7)$$

The *tl* operation, which returns the tail of a list, is given as:

$$tl : STVEqList \rightarrow STVEquation$$

and is defined:

$$tl\,([]) = []$$
$$tl\,(a, as) = as$$

such that the tl of the above list would be:

$$V_2(t, a, x) = mult(3, 6),$$
$$\vdots,$$
$$V_n(t, a, x) = sub(6.3),$$
$$[]$$

The final list operation allows the selection of a particular equation out of a list. In the case of the *STEqList*, each equation is a value function that has a particular module number associated with it; the operation of *GetEl* is therefore to select the State Transition equation from the list that corresponds to a particular module number. *GetEl* it is given as:

$$GetEl : STVEqList \times N \rightarrow STVEquation$$

and is simultaneously defined as:

$$\begin{aligned} GetEl([], n) &= [] \\ GetEl\,((V_n(t, a, x) = z, vs), n) &= (V_n(t, a, x) = z) \\ GetEl\,((V_p(t, a, x) = z, vs), n) &= GetEl(vs, n) \end{aligned}$$

The *GetEl* operation is the main operation that changes in all of the list specifications and is now discussed for the remainder of the lists.

Initial State Value Equations List Specification

Definition of the Initial State for Value Functions are of the form:

$$V_n(0, a, x) = x_n$$

the retrieval of an Initial State definition for a Value Function is performed in the same manner as for the State Transition definition of a Value Functions, that is to

say by recursing over the list of equations until the correct element is found. *GetEl* is given as:

$$GetEl : ISVEqList \times N \rightarrow ISVEquation$$

and is simultaneously defined as:

$$\begin{aligned} GetEl([], n) &= null \\ GetEl((V_n(t,a,x) = z, vs), n) &= (V_n(t,a,x) = z) \\ GetEl((V_p(t,a,x) = z, vs), n) &= GetEl(vs, n) \end{aligned}$$

The *GetEl* operation for a dSCA Initial State Value Equation List is similarly defined with the appropriate types.

dSCA State Transition Value Equations List Specification

DSCA State Transition Value Functions are of the form

$$V_n(t+1, a, x) = f_n(arg_1, \ldots, \arg_{p(i)})$$

the retrieval of an Initial State definition of the value function is performed in the same manner as for the State Transition definition of the Value Function, that is to say by recursing over the list of equations until the correct element is found. *GetEl* is given as:

$$GetEl : dSCASTVEqList \times N \rightarrow dSCASTVEquation$$

and is simultaneously defined as:

$$\begin{aligned} GetEl([], n) &= null \\ GetEl((V_n(t,a,x) = z, vs), n) &= (V_n(t,a,x) = z) \\ GetEl((V_p(t,a,x) = z, vs), n) &= GetEl(vs, n) \end{aligned}$$

γ Wiring Function Equation List Specification

There are two forms of γ-wiring function lists, one for SCAs and the other for dSCAs. For an SCA the γ-wiring functions are of the form:

$$\gamma(i, j) = X$$

thus selection will be based on the variables i and j, and it is therefore appropriate to define:

$$GetEl : \gamma SCAEqList \times N^2 \rightarrow \gamma SCAEquation$$

as:

$$\begin{aligned} GetEl([]\, i, j) &= null \\ GetEl\,((\gamma(i,j) = X, vs), i, j) &= (\gamma(i,j) = X) \\ GetEl\,((\gamma(m,n) = X, vs), n) &= GetEl(vs, i, j) \end{aligned}$$

For a dSCA, the γ-wiring function is of the form:

$$\gamma_z(i,j) = X$$

and selection will therefore be based on the variables i, j **and** z. It is therefore appropriate to define:

$$GetEl : \gamma dSCAEqList \times N^3 \rightarrow \gamma dSCAEquation$$

as:

$$\begin{aligned} GetEl([]\, i, j, z) &= null \\ GetEl\,((\gamma_z(i,j) = X, vs), i, j, z) &= (\gamma_z(i,j) = X) \\ GetEl\,((\gamma_z(m,n) = X, vs), i, j, z) &= GetEl(vs, i, j, z) \end{aligned}$$

β Wiring Function Equation List Specification

The β-wiring operation are extracted in a similar manner as for the γ-wiring operation. Again, there are two forms of β-wiring function lists, one for SCAs and the other for dSCAs. In the SCA the β-wiring functions are of the form:

$$\beta(i,j) = X$$

thus selection will be based on the variables i and j, and it is therefore appropriate to define:

$$GetEl : \beta SCAEqList \times N^2 \rightarrow \beta SCAEquation$$

as:

$$\begin{aligned} GetEl([]\, i, j) &= null \\ GetEl\,((\beta(i,j) = X, vs), i, j) &= (\beta(i,j) = X) \\ GetEl\,((\beta(m,n) = X, vs), n) &= GetEl(vs, i, j) \end{aligned}$$

For a dSCA, the β-wiring function is of the form:

$$\beta_z(i,j) = X$$

thus selection will be based on the variables i, j **and** z. It is therefore appropriate to define:

$$GetEl : \beta dSCAEqList \times N^3 \rightarrow \beta dSCAEquation$$

as:

$$\begin{aligned} GetEl([]\, i, j, z) &= null \\ GetEl\,((\beta_z(i,j) = X, vs), i, j, z) &= (\beta_z(i,j) = X) \\ GetEl\,((\beta_z(m,n) = X, vs), i, j, z) &= GetEl(vs, i, j, z) \end{aligned}$$

Delay Function List Specification

Delay functions also have 2 forms, one for both SCAs and concrete dSCAs, and another for abstract dSCAs. In SCAs and concrete dSCAs the delay functions are of the form:

$$\delta_{i,j}(t, a, x) = X$$

and selection is therefore based on the variables i and j, and it is therefore appropriate to define:

$$GetEl : \delta EqList \times N^2 \rightarrow \delta Equation$$

as:

$$\begin{aligned} GetEl([]\, i, j) &= null \\ GetEl\,((\delta_{i,j}(t, a, x), vs), i, j) &= (\delta_{i,j}(t, a, x) = X) \\ GetEl\,((\delta_{m,n}(t, a, x) = X, vs), n) &= GetEl(vs, i, j) \end{aligned}$$

For concrete dSCAs, the delay function is of the format:

$$\delta_{i,j,z}(t,a,x) = X$$

thus selection is based on the variables i, j and z. *GetEl* is given as:

$$GetEl : \delta dSCAEqList \times N^3 \rightarrow \delta dSCAEquation$$

and defined as:

$$\begin{aligned} GetEl([]\, i,j,z) &= null \\ GetEl\,((\delta_{i,j,z}(t,a,x),vs),i,j,z) &= (\delta_{i,j,z}(t,a,x) = X) \\ GetEl\,((\delta_{m,n,p}(t,a,x) = X,vs),i,j,z) &= GetEl(vs,i,j,z) \end{aligned}$$

Mapping Function List Specification

The Mapping Function List contains elements the mapping (or inverse mapping), these are of the form

$$\Xi(i,j) = (x,y)$$

and elements are therefore selected by means of the variables i and j, using the *GetEl* operation of the Mapping Function List specification:

$$GetEl : MapEqList \times N^2 \rightarrow MapEquation$$

which is defined as:

$$\begin{aligned} GetEl([]\, i,j) &= null \\ GetEl\,((\Xi(i,j) = X,vs),i,j) &= (\Xi(i,j) = X) \\ GetEl\,((\Xi(m,n) = X,vs),n) &= GetEl(vs,i,j) \end{aligned}$$

it is similarly defined for the inverse mapping.

Project Functions List Specification

Projection functions are of the form

$$d^N_{i,j,z} :\rightarrow N$$

thus it is appropriate to define the *GetEl* operation:

$$GetEl : ProjEqList \times N^3 \rightarrow ProjEquation$$

as:

$$\begin{aligned} GetEl([]\, i, j, z) &= null \\ GetEl\left((d^N_{i,j,z} = X, vs), i, j, z\right) &= \left(d^N_{i,j,z} = X\right) \\ GetEl\left((d^N_{m,n,p} = X, vs), i, j, z\right) &= GetEl(vs, i, j, z) \end{aligned}$$

9.2.4 SCA Value Functions

SCA State Transition Equation Specification (STVEquation)

An SCA State Transition definition of the Value Function is an equation of type STVEquation. The STVEquation specification defines how these definitons of Value Functions (a restricted form of equation) are constructed. The specification contains one operation for constructing and two operations for decomposing a Value Function.

A State Transition definition is an equation made up from two terms, one of the form $V_i(t, a, x)$ and the other $f_i(arg_1, \ldots, arg_n)$, as follows:

$$V_i(t, a, x) = f_i(arg_1, \ldots, arg_n)$$

The i^{th} module in a network N with k modules and n inputs will have an operation component of:

$$V_i : T \times [T \rightarrow M_A]^n \times M_A^k \rightarrow M_A$$

where T represents some imported clock, and algebra M_A is the imported specification from which data in the network is selected.

The equational component in the SCA specification for the i^{th} module will consist of two entries, one that defines the value at time $t = 0$:

$$V_i(0, a, x) = e$$

and one that defines the value at all other times $t \in T$ for some clock T:

$$V_i(t, a, x) = e'$$

This section is dealing with the State Transition, and thus define it is appropriate to give one construction operation that takes a module number, a time term and a VFOpTerm:

$$CreateVF : N \times Term \times VFOpTerm \rightarrow STEquation$$

and to define it as:

$$CreateVF(n, t, t_2) = (V_n(t, a, x) = t_2)$$

A constructed Value Function equation is made up of two component parts,

$$VFCallTerm = VFOpDef$$

where the $VFCallTerm$ type are terms that are of the form $V_n(t, a, x)$ and the $VFOpTerm$ type as terms built from elements of M_A. For example, $CreateVF(n, t+1, add(5, 4)$ would result in $V_n(t + 1, a, x) = add(5, 4)$

The decomposition operations provided in the Value Function specification are used to extract various components from a Value Function definition and also from the components of the Value Function. For example, the $RetTerm$ operation will return a term from an $STEquation$, it is given as:

$$RetTerm : STVEquation \times N \rightarrow Term$$

and defined as:

$$RTerm(V_n(t, a, x) = f_n, 1) = V_n(t, a, x)$$
$$RTerm(V_n(t, a, x) = f_n, 2) = f_n$$

SCA Initial State Equation Specification (ISVEquation)

An SCA Initial State definition of a Value Function is of type ISVEquation, and this specification defines how these definitions of a Value Functions (a restricted form of equation) are constructed. The specification contains one operation for constructing and two operations for decomposing the Initial State component of a Value Function.

The Initial State is an equation made up from two terms as follows:

$$V_i(t, a, x) = x_n$$

One construction operation is defined, and it is given as:

$$CreateVF : N^2 \times VFOpTerm \rightarrow STEquation$$

and defined:

$$CreateVF(n, t, t_2) = (V_n(t, a, x) = t_2)$$

For example, $CreateVF(n, 0, 4)$ would result in $V_n(0, a, x) = 5$

The decomposition operations provided in the Value Function specification are used to extract various components from a Value Function and also from the components of the Value Function. For example, the *RetTerm* operation will return a term from an *STEquation*, it is given as:

$$RetTerm : ISVEquation \times N \rightarrow Term$$

and defined as:

$$RTerm(V_n(0, a, x) = x_n, 1) = V_n(0, a, x)$$
$$RTerm(V_n(0, a, x) = x_n, 2) = x_n$$

9.2.5 VFCallTerm and VFOpTerm Specifications

Both the VFCallTerm and VFOpTerm are specifications of terms, and it is not intended to provide an algebraic definition of terms above and beyond that given in

Chapter 9.2. In this section an object of the form $V_i(t, a, x)$ is called a VFCallTerm and one called a VFOpTerm will be a term conforming to one of the following three definitions:

- An constant from M_A or an operation from M_A whose arguments are VFCall-Terms, for example $add(V_n(t, a, x), V_p(t, a, x)$ or $true$;
- a VFCallTerm, for example $V_n(t, a, x)$; or
- a term representing an input stream, and being of the form $a_i(t)$

A single decomposition operation is defined for each of the $VFCallTerm$ and $VFOpDef$ elements. For the $VFCallTerm$ access is allowed to the index of the $VFCallTerm$ given as:

$$GetInd : VFCallTerm \rightarrow N$$

and defined as:

$$GetInd(V_n(t, a, x)) = n$$

The decomposition operation for the $VFOpDef$ term is one that can return arguments:

$$GetArg : VFOpDef \times N \rightarrow Term$$

defined appropriately over the number of arguments that are possible in an operation built from M_A, for example:

$$\begin{aligned}
&GetArg(op(t_1), 1) = t_1 \\
&GetArg(op(t_1, t_2), 1) = t_1 \\
&GetArg(op(t_1, t_2), 2) = t_2 \\
&GetArg(op(t_1, t_2, t_3), 1) = t_1 \\
&GetArg(op(t_1, t_2, t_3), 2) = t_2 \\
&GetArg(op(t_1, t_2, t_3), 3) = t_3
\end{aligned}$$

9.2.6 Wiring Function Specification

The wiring function specification consist of:

1. $\gamma SCAEquation$;
2. $\gamma dSCAEquation$;
3. $\beta SCAEquation$;
4. $\beta dSCAEquation$.

In this chapter the definition of the $\gamma SCAEquation$ specification is given in detail as an example and the other specifications can be produced in a similar manner.

One operation is required for composition:

$$Build\gamma : N^2 \times \{S, M, U\} \rightarrow \gamma SCAEquation$$

given as:

$$Build\gamma(a, b, X) = (\gamma(a, b) = X)$$

Decomposition is provided by one operation:

$$GetArg : \gamma SCAEquation \times N \rightarrow Term$$

given as:

$$GetArg\,(\gamma(a, b) = X, 1) = \gamma(a, b)$$
$$GetArg\,(\gamma(a, b) = X, 2) = X$$

The $dSCA\gamma Equation$ specification will introduce am additional index for the program counter, therefore the equations are defined appropriately.

9.2.7 Delay Function Specification

As with some of the other specification, there are two forms of the delay function, one for SCAs and concrete dSCAs, and one for abstract dSCAs. First the SCA form is discussed.

One operation is required for composition:

$$Build\delta : N^2 \times Term^3 \rightarrow \delta SCAEquation$$

given as:

$$Build\delta(i, j, a, x, t-1) = (\delta i, j(t, a, x) = t - 1)$$

Decomposition is provided by two operation, the first $GetIndex$ returns the index of the delay function:

$$GetIndex : \delta SCAEquation \rightarrow N^2$$

given as:

$$GetIndex(\delta_{i,j}(t, a, x) = x) = (i, j)$$

and the second decomposition operation is the $GetArg$ operation which returns elements from the arguments of the delay function:

$$GetArg : \delta SCAEquation \times N \rightarrow Term$$

note that we are not interested in the actual values, just that a term is returned:

$$GetArg\,(\delta_{i,j}(a, b, c) = X, 1) = a$$
$$GetArg\,(\delta_{i,j}(a, b, c) = X, 2) = b$$
$$GetArg\,(\delta_{i,j}(a, b, c) = X, 3) = c$$

The abstract dSCA forms are similarly defined but take account of the additional index introduced for the program counter.

9.2.8 Conclusion

This section has provided the details on the fundamental specification used in the act of transforming one SCA model to another. The algebraic nature of these transformations leads to the potential of automation of the process in the future.

9.2.9 Sources

This chapter is all my own work, except for the discussion on hierarchies of SCAs which comes from Poole, Tucker and Holden's work, [PHT98].

Chapter 10

SCA to Abstract dSCA

Purpose of Transformation
To introduce the necessary syntactic sugar required to describe an existing SCA as an abstract dSCA, where the defining shape of the abstract dSCA reflects the shape of the source SCA.

10.1 Process

This chapter describes the process used to build the components of the abstract dSCA with defining shape $\nabla = (k, 1)$ from an SCA with defining shape $\nabla = (k, 1)$. It considers the necessary transformations of:

1. Wiring Functions;
2. Delay Functions;
3. Initial State Value Function Equations; and
4. State Transition Value Functions.

Once the individual transformations are described in detail they are pulled together to provide details of the transformation specification. At the end of this chapter this transformation specification is applied to the SCA implementation of the GRCP to produce a Form 1 abstract dSCA implementation. Finally, the correctness of the generated Form 1 abstract dSCA is discussed. The transformation of the operations part of the specification will not be discussed.

10.1.1 Prerequisites

There are a limited number of prerequisites for this transformation. For the SCA to Form 1 abstract dSCA there are the following prerequisites:

- The source SCA is an atomic SCA;
- Arguments to the functional specification of a state transition phase definition of Value Function are indexed from 1, such that the wiring and delay functions also start with the index 1;
- Module numbering starts at 1, and sequentially increments, i.e. no module is ever denoted as m_0;
- Modules are numbered such that if there are k modules in the network, then they are numbered $1, \ldots, k$; and
- All delays in the source SCA are of unit length.

10.1.2 Wiring Functions

There is a subtle difference between the wiring functions in an SCA and those in a corresponding Form 1 abstract dSCA. This is due to the requirement of the dSCA to have wiring functions for all values of $n(i) + 1$ (the number of arguments to modules) and this value being consistent across all modules.

Additionally, a suffix is introduced to reflect the value of the program counter the wiring function relates to. Since this transformation constructs a simple abstract dSCA that maps to the SCA, then Max_N will only ever reaches 1, and thus the suffix will always be 0.

Each module in the abstract dSCA will replicate the wiring in the SCA and will also get a wiring for its 0^{th} input to wire it to the program counter (this is the reason for the first prerequisite). The program counter's wiring itself must also be created, and the second prerequisite allows the program counter module to assume the index of 0 as described in the definition of abstract dSCAs.

γ-wiring Operations

Consider the SCA γ-wiring function:

$$\gamma(x, y) = z$$

the transformation should produce the corresponding dSCA γ-wiring function:

$$\gamma_0(x, y) = z$$

The *informal* process for generating the γ-wiring functions for a Form 1 abstract dSCA from an SCA is:

- For each module m_i in the target network, where $1 \leq i \leq k$
  - Add the following γ-wiring function to the list of new γ- wiring functions to represent the wiring to the program counter:

$$\gamma_0(i, 0) = M$$

  - For each argument $1 \leq j \leq n(i)$ add:

$$\gamma_0(i, j) = \begin{cases} old\_value & \text{if old\_value exists in the source SCA} \\ U & \text{otherwise} \end{cases}$$

- Add the wiring function for the program counter:

$$\gamma_0(pc, 0) = M$$

Formally, the $Create\gamma s$ operation is introduced:

$$Create\gamma s : SCAAlgebra \rightarrow \gamma EqList$$

which takes the SCA and calls the $B\gamma s$ operation passing it the number of modules and value of $p(i)$ (the number of arguments, and which will be referred to as Max_A in the transformations) from the source SCA. Max_A for the target SCA will be one greater than in the source dSCA since the shape of the SCA is not being altered, but an additional argument is required. Additionally, the $B\gamma s$ operation takes the extracted γ-wiring functions from the source SCA and an empty list (which will eventually contain the Form 1 abstract dSCA ned as:

$$Create\gamma s\left(\ source\_SCA\ \right) = B\gamma s\left(\begin{array}{l} num\_mod(source\_SCA), \\ GetMaxA(source\_SCA), \\ Get\gamma Eqs(source\_SCA), \\ GetMaxA(source\_SCA), \\ Get\gamma Eqs(source\_SCA), \\ [] \end{array}\right)$$

The $B\gamma s$ operation:

$$B\gamma s : N^2 \times \gamma SCAEqList \times \gamma dSCAEqList \rightarrow \gamma dSCAEqList$$

is defined simultaneously in two cases to recurse over the number of modules. The first case is where the number of modules is greater than 0, in this situation recursive calls to the $B\gamma s$ operation are made whilst decrementing the number of modules, and creating a new γ-wiring list from a call to the $B\gamma$ operation appended to the second

argument:

$$B\gamma s\begin{pmatrix} num\_mod, \\ Max_A, \\ eqs, \\ neqs, \end{pmatrix} = B\gamma s\begin{pmatrix} num\_mod - 1, \\ Max_A, \\ eqs, \\ \begin{pmatrix} B\gamma\begin{pmatrix} Max_A, \\ num\_mod, \\ eqs, \end{pmatrix}, neqs \end{pmatrix} \end{pmatrix}$$

where the operation $B\gamma$:

$$B\gamma : N^2 \times \gamma SCAEqList \rightarrow \gamma dSCAEqList$$

will construct the 0^{th} arguments wiring function and calls the $B\gamma Arg$ operation to create the wiring functions for the other arguments $1, \ldots, Max_A$:

$$B\gamma\begin{pmatrix} Max_A, \\ num\_mod, \\ eqs, \end{pmatrix} = \begin{pmatrix} Build\gamma\begin{pmatrix} \gamma_0, \\ num\_mod, \\ 0, \\ M \end{pmatrix}, B\gamma Args\begin{pmatrix} Max_A, \\ num\_mod, \\ eqs, \\ [] \end{pmatrix} \end{pmatrix}$$

$B\gamma Args$ is given as:

$$B\gamma Args : N^2 \times \gamma SCAEqList \times \gamma dSCAEqList \rightarrow \gamma dSCAEqList$$

and is defined simultaneously over the Max_A argument, the first case being:

$$B\gamma Args\begin{pmatrix} arg\_val, \\ num\_mod, \\ eqs, \\ [] \end{pmatrix} = \begin{pmatrix} B\gamma Args\begin{pmatrix} arg\_val - 1, \\ num\_mod, \\ eqs, \\ [] \end{pmatrix}, B\gamma Arg\begin{pmatrix} arg\_val, \\ num\_mod, \\ eqs \end{pmatrix} \end{pmatrix}$$

where the operation $B\gamma Arg$ is used to generate the wiring function, depending upon whether it existed or not in the source SCA:

$$B\gamma Arg : N^2 \times \gamma SCAEqList\times \rightarrow \gamma dSCAEqList$$

it is defined as:

$$B\gamma Arg\begin{pmatrix} arg\_val, \\ num\_mod, \\ eqs \end{pmatrix} = \begin{cases} Build\gamma \begin{pmatrix} \gamma_0, \\ num\_mod, \\ arg\_val, \\ M \end{pmatrix} & \text{if GetEl} \begin{pmatrix} eqs, \\ num\_mod, \\ arg\_val \end{pmatrix} \neq [] \\ Build\gamma \begin{pmatrix} \gamma_0, \\ num\_mod, \\ arg\_val, \\ U \end{pmatrix} & \text{otherwise} \end{cases}$$

The second case of the $B\gamma Args$ operation is defined to return the generated list of wiring functions:

$$B\gamma Args\begin{pmatrix} 0, \\ num\_mod, \\ eqs, \\ neqs \end{pmatrix} = neqs$$

The second case definition for the $B\gamma s$ operation is where the module number under consideration is 0. This module does not exist in the source SCA and so the process generates the wiring function for the program counter:

$$B\gamma s\begin{pmatrix} 0, \\ Max_A, \\ eqs, \\ neqs, \end{pmatrix} = \begin{pmatrix} neqs, Build\gamma \begin{pmatrix} \gamma_0, \\ pc, \\ 0, \\ M \end{pmatrix} \end{pmatrix}$$

β-wiring Operations

Consider the SCA β-wiring function:

$$\beta(x, y) = z$$

the transformation should produce the corresponding dSCA β-wiring function:

$$\beta_0(x, y) = z$$

The *informal* process for generating the β-wiring functions for a Form 1 abstract dSCA from an SCA is:

- For each module m_i in the target network, where $1 \leq i \leq k$:
  - Add the following β-wiring function to the list of new β- wiring functions to represent the wiring to the program counter:

$$\beta_0(i, 0) = pc$$

  - For each argument $1 \leq j \leq n(i)$ add:

$$\beta_0(i, j) = \begin{cases} old\_value & \text{if old\_value exists in the source SCA} \\ U & \text{otherwise} \end{cases}$$

- Add the wiring function for the program counter:

$$\gamma_0(pc, 0) = pc$$

Formally, the $Create\beta s$ operation is introduced as:

$$Create\beta s : SCAAlgebra \rightarrow \beta EqList$$

and it takes the SCA and then calls the $B\beta s$ operation passing it the extracted β-wiring functions from the SCA, an empty list (which will eventually contain the Form 1 abstract dSCA β-wiring functions), and the details of the source SCA used for the same purpose as described in the γ-wiring transformation. It is defined as:

$$Create\beta s\left(\; source\_SCA \; \right) = B\beta s \left(\begin{array}{l} num\_mod(source\_SCA), \\ GetMaxA(source\_SCA), \\ Get\beta Eqs(source\_SCA), \\ [] \end{array} \right)$$

The $B\beta s$ operation:

$$B\beta s : N^2 \times \beta SCAEqList \times \beta dSCAEqList \rightarrow \beta dSCAEqList$$

is defined simultaneously by two cases to recurse over the number of modules. The first case is where the number of modules is greater than 0, in this situation recursive calls to the $B\beta s$ operation are made, decrementing the number of modules, and creating a new β-wiring list from a call to the $B\beta$ operation to be appended to the second argument:

$$B\beta s \begin{pmatrix} num\_mod, \\ Max_A, \\ eqs, \\ neqs, \end{pmatrix} = B\beta s \begin{pmatrix} num\_mod - 1, \\ Max_A, \\ eqs, \\ \begin{pmatrix} B\beta \begin{pmatrix} Max_A, \\ num\_mod, \\ eqs, \end{pmatrix}, neqs \end{pmatrix} \end{pmatrix}$$

where the operation $B\beta$:

$$B\beta : N^2 \times \beta SCAEqList \rightarrow \beta dSCAEqList$$

constructs the 0^{th} arguments wiring function and calls the $B\beta Arg$ operation to create the wiring functions for the other arguments $1, \ldots, Max_A$:

$$B\beta \begin{pmatrix} Max_A, \\ num\_mod, \\ eqs, \end{pmatrix} = \begin{pmatrix} Build\beta \begin{pmatrix} \beta_0, \\ num\_mod, \\ 0, \\ pc \end{pmatrix}, B\beta Args \begin{pmatrix} Max_A, \\ num\_mod, \\ eqs, \\ [] \end{pmatrix} \end{pmatrix}$$

$B\beta Args$ is given as:

$$B\beta Args : N^2 \times \beta SCAEqList \times \beta dSCAEqList \rightarrow \beta dSCAEqList$$

and is defined simultaneously over the Max_A argument, the first case being:

$$B\beta Args\begin{pmatrix} arg\_val, \\ num\_mod, \\ eqs, \\ [] \end{pmatrix} = \left(B\beta Args\begin{pmatrix} arg\_val - 1, \\ num\_mod, \\ eqs, \\ [] \end{pmatrix}, B\beta Arg\begin{pmatrix} arg\_val, \\ num\_mod, \\ eqs \end{pmatrix} \right)$$

where the operation $B\beta Arg$ is used to generate the wiring function depending upon whether it existed or not in the source SCA:

$$B\beta Arg : N^2 \times \beta SCAEqList\times \rightarrow \beta dSCAEqList$$

it is defined as:

$$B\beta Arg\begin{pmatrix} av, \\ nm, \\ eqs \end{pmatrix} = \begin{cases} Build\beta\begin{pmatrix} \beta_0, \\ nm, \\ av, \\ RetTerm(GetEl\begin{pmatrix} eqs, \\ nm, \\ av \end{pmatrix}, 2) \end{pmatrix} & \text{if cond}_1 \\ Build\beta\begin{pmatrix} \beta_0, \\ nm, \\ av, \\ \omega \end{pmatrix} & \text{otherwise} \end{cases}$$

where:

$$cond_1 = GetEl\begin{pmatrix} eqs, \\ nm, \\ av \end{pmatrix} \neq []$$

The second case of the $B\beta Args$ operation is defined to return the already generated list of wiring functions if the argument number under consideration is 0:

$$B\beta Args\begin{pmatrix} 0, \\ num\_mod, \\ eqs, \\ neqs \end{pmatrix} = neqs$$

The second case definition for the $B\beta s$ operation, is where the module under consideration is module number 0, which does not exist in the source SCA, but does in the abstract dSCA, and the wiring function for the program counter is generated in this case as:

$$B\beta s \begin{pmatrix} num\_mod, \\ Max_A, \\ eqs, \\ neqs, \end{pmatrix} = \begin{pmatrix} neqs, Build\beta \begin{pmatrix} \beta_0, \\ pc, \\ 0, \\ pc \end{pmatrix} \end{pmatrix}$$

10.1.3 Delay Functions

Consider the SCA delay function:

$$\delta_{i,j}(t, a, x) = z$$

in the Form 1 abstract dSCA it would be transformed into the form:

$$\delta_{i,j,0}(t, a, x) = z$$

and it is always the case in the Form 1 abstract dSCA that the delay is a unit delta.

Informally, to create the Form 1 abstract dSCA delay functions from the SCA the following process is executed:

- For each module m_i in the target network, where $1 \leq i \leq k$:
  - For each argument j where $0 \leq j \leq n(i)$ generate:

$$\delta_{i,j,0}(t, a, x) = t - 1$$

- Add the delay function for the program counter:

$$\delta_{pc,0,0}(t, a, x) = t - 1.$$

Formally, the $Create\delta s$ operation is introduced to recurse through the list of SCA delay functions and create the list of Form 1 abstract dSCA delay functions. $Create\delta s$ operation is given as:

$$Create\delta s : SCAAlgebra \rightarrow \delta sEqList$$

It is defined as an operation that takes an SCA and extracts the number of modules and the maximum value of $n(i)$ which will be referred to as Max_A through the transformation. $Create\delta s$ calls the $B\delta s$ operation with the above arguments and an empty list (to hold the returned values), it is defined as:

$$Create\delta s(source\_SCA) = B\delta s \begin{pmatrix} nummod(source\_SCA), \\ GetMaxA(source\_SCA), \\ [], \end{pmatrix}$$

The $B\delta s$ operation, given as:

$$B\delta s : N^2 \delta dSCAEqList \rightarrow \delta dSCAEqList$$

is defined by two cases. The first case adresses the situation where the number of modules is greater than 0, where the following definition applies:

$$B\delta s \begin{pmatrix} num\_mod, \\ Max_A, \\ neqs \end{pmatrix} = B\delta s \begin{pmatrix} num\_mod - 1, \\ Max_A, \\ B\delta Args \begin{pmatrix} Max_A, \\ num\_mod, \\ [] \end{pmatrix} \end{pmatrix}$$

with the $B\delta Args$ operation given as:

$$B\delta Args : N^2 \times \delta dSCAEqList \rightarrow dSCAEqList$$

and defined, recursively over the number of arguments, with the first case definition

of:

$$B\delta Args\begin{pmatrix} arg\_val, \\ num\_mod, \\ neqs \end{pmatrix} = \begin{pmatrix} B\delta Args\begin{pmatrix} arg\_val-1, \\ num\_mod, \\ neqs \end{pmatrix}, Build\delta\begin{pmatrix} num\_mod, \\ arg_val, \\ 0, \\ t-1 \end{pmatrix} \end{pmatrix}$$

When the *arg_val* number is 0 the base case for the $B\delta Args$ operation is defined to build a delay function for the 0^{th} argument:

$$B\delta Args\begin{pmatrix} 0, \\ num\_mod, \\ neqs \end{pmatrix} = Build\delta\begin{pmatrix} num\_mod, \\ 0, \\ 0, \\ t-1 \end{pmatrix}$$

The second case of $B\delta s$, where the module number is 0, returns the results calculated so far, appended to the delay function for the program counter:

$$B\delta s\begin{pmatrix} 0, \\ Max_A, \\ neqs \end{pmatrix} = \begin{pmatrix} neqs, Build\delta\begin{pmatrix} num\_mod, \\ 0, \\ 0, \\ t-1 \end{pmatrix} \end{pmatrix}$$

10.1.4 Initial State Equations

Consider an Initial State Equation from the SCA, it will be of the form:

$$V_i(0, a, x) = x_i$$

The corresponding Initial State Equation in the Form 1 abstract dSCA will be:

$$V_i(0, a, x) = x_i$$

It can be seen that there is no transformation to make for the Initial State Equation of the SCA modules; however, the Form 1 abstract dSCA has an additional module,

the program counter. *Informally*, the process for creating Form 1 abstract dSCA Initial State equations is therefore:

- Copy across the Initial State equation from the SCA; and
- Add an Initial State equation for the program counter $V_{pc}(0, a, x) = 0$

Formally, the $CreateISVFs$ operation is introduced:

$$CreateISVFs : SCAAlgebra \rightarrow dSCAISVEqList$$

and it is defined as taking the SCA, extracting the Initial State equations from it, and adding an Initial State equation for the program counter to the resultant list:

$$CreateISVFs(Source\_SCA) = \begin{pmatrix} GetEqIV(Source\_SCA), \\ CreateVF(pc, t, 0) \end{pmatrix}$$

10.1.5 State Transition Equations

State Transition equations in the abstract dSCA differ from those in an SCA by the need to wrap the functionality in a conditional operation. Consider the SCA State Transition equation:

$$V_i(t + 1, a, x) = e$$

when transformed for the Form 1 abstract dSCA it would become:

$$V_i(t + 1, a, x) = e \; if \; v_{pc}(t, a, x) = 0$$

or if written with strict compliance to M_A it would be written as:

$$V_i(t + 1, a, x) = cond(V_{pc}(t, a, x) = 0, e, null)$$

Note that the *cond* requires 3 arguments, and so the null value is placed into the definition for the case where $V_{pc}(t, a, x)$ does not equal 0 - which will not be the case for an abstract dSCA where $Max_N = 1$, as here.

Informally, the process of transformation can be written as:

- For each SCA State Transition equation:
  - Select VFCallTerm;
  - Select VFOpTerm;
  - create new VFOpTerm term $cond(V_{pc}(t,a,x) = 0, rewire(VFOpTerm), null)$ using a version of VFOpTerm that has had its inputs rewired to take account of the new wiring and delay functions; and
  - create new State Transition equation.
- Add the following equation to reflect the program counter:

$$V_{pc}(t+1,a,x) = mod(add(V_{pc}(t,a,x),1),1) \text{ if } \mathrm{V_{pc}(t, a, x)} = 0$$

When describing the abstract dSCA State Transition equations the *cond* operation is turned into the more readable form.

Formally, the $CreateSTVFs$ operation is introduced:

$$CreateSTVFs : SCAAlgebra \rightarrow dSCASTVEqList$$

with the intention that it takes an SCA specification and creates a list of Form 1 abstract dSCA State Transition equations by extracting the SCA State Transition equations and supplying the new wiring and delay functions to the $BSTs$ operation. It is given as:

$$CreateSTVFs(Source\_SCA) = BSTs \left(\begin{array}{l} GetEqIV(Source\_SCA), \\ [], \\ Create\beta s\left(Get\beta Ops(Source\_SCA)\right), \\ Create\gamma s\left(Get\gamma Ops(Source\_SCA)\right), \\ Create\delta s\left(Get\delta Ops(Source\_SCA)\right), \end{array} \right)$$

The operation $BSTs$:

$$BSTs: \; STVEqList \times dSCASTVEqList \times \beta dSCAEqList \times \gamma dSCAEqList \times \delta dSCAEqList \rightarrow dSCASTVEqList$$

is defined recursively over the SCA State Transition equation list, in 2 cases; one case for when there are still equations to process, and the second for when there are no equations left to process. In the first case a recursive call is made to the $BSTs$ operation with the tail of the equation list, and the newly created State Transition equation appended to the 2nd argument:

$$BSTs\begin{pmatrix}(e,eqs),\\ neqs,\\ \beta s,\\ \gamma s,\\ \delta s\end{pmatrix} = BSTs\begin{pmatrix}eqs,\\ \begin{pmatrix}BST\begin{pmatrix}e,\\ \beta s,\\ \gamma s,\\ \delta s\end{pmatrix}, neqs\end{pmatrix},\\ \beta s,\\ \gamma s,\\ \delta s\end{pmatrix}$$

The operation BST:

$$BST: IVEquation \times \beta OpList \times \gamma OpList \times \delta OpList \rightarrow IVEquation$$

is subsequently defined as:

$$BST\begin{pmatrix}e,\\ \beta s,\\ \gamma s,\\ \delta s\end{pmatrix} = CreateVF\begin{pmatrix}RetTerm(e,1),\\ \begin{pmatrix}\begin{pmatrix}m_1' = 0,\\ cond\begin{pmatrix}rewire\begin{pmatrix}RetTerm(e,2),\\ newVFTerm,\\ \beta s,\\ \gamma s,\\ \delta s\end{pmatrix},\\ null\end{pmatrix}\end{pmatrix}\end{pmatrix}\end{pmatrix}$$

where:

$$\begin{aligned} m_1^{'} &= V_{RetTerm(GetEl(\gamma s,i,0),2)}\,(RetTerm(GetEl(\delta s,i,0,0),2),a,x) \\ &= V_{RetTerm(\gamma_0(i,0)=pc,2)}\,(RetTerm(\delta_{i,0,0}(t,a,x)=t-1,2),a,x) \\ &= V_{pc}(t-1,a,x) \end{aligned}$$

but in this example $t = t + 1$ thus:

$$m_1^{'} = V_{pc}(t,a,x)$$

and:

$$newVFTerm = GetIndex(RetTerm(e,1))$$

To complete the definition of State Transition equation transformation a definition of the *rewire* operation is required. Whilst it is possible to provide a definition covering the general case of any number of arguments to an operation, in the example there will only ever be a maximum of 3 arguments (see definition of M_A) and so a specific implementation of *rewire* can be defined. It is given as:

$$rewire : Term \times N \times \beta OpList \times \gamma OpList \times \delta OpList \rightarrow Term$$

and defined as:

$$rewire\begin{pmatrix} op, \\ i, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = op$$

$$rewire\begin{pmatrix} op(t_1), \\ i, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = op\left(wire\begin{pmatrix} t_1, \\ i, \\ 1, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}\right)$$

$$rewire\begin{pmatrix} op(t_1,t_2), \\ i, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = op\left(wire\begin{pmatrix} t_1, \\ i, \\ 1, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_2 \\ i, \\ 2, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}\right)$$

$$rewire\begin{pmatrix} op(t_1,t_2,t_3), \\ i, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = op\left(wire\begin{pmatrix} t_1, \\ i, \\ 1, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_2 \\ i, \\ 2, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_3 \\ i, \\ 3, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}\right)$$

where the *wire* operation is defined as:

$$wire\begin{pmatrix} t \\ i, \\ j, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = \begin{cases} V_{new\_index}(new\_time, a, x) & \text{if } wiring = M \\ a_{new\_index}(new\_time) & \text{if } wiring = S \end{cases}$$

and:

$$wiring = RetTerm(GetEl(\gamma s, i, j, 0), 2)$$

A true implementation would define *new_index* and *new_time* to return the first part of the relevant elements, e.g. $\beta_{i,j,0}$, resulting in a State Transition equation similar to:

$$V_i(t+1, a, x) = op\left(V_{\beta_0(i,1)}(\delta_{i,1,0}(t+1, a, x), a, x)\right)$$

and then at a future point these would be simplified to the values, resulting in:

$$V_i(t+1, a, x) = op\left(V_k(t, a, x)\right)$$

Instead the issue is expediated and simplification is performed now, thus *new_index* and *new_time* are defined to return the 2nd term of the respective wiring and delay functions:

$$new\_index = RetTerm\left(\begin{array}{l} GetEl\left(\begin{array}{l} \beta s, \\ i, \\ j, \\ 0 \end{array}\right), 2 \end{array}\right)$$

$$new\_time = RetTerm\left(\begin{array}{l} GetEl\left(\begin{array}{l} \delta s, \\ i, \\ j, \\ 0 \end{array}\right), 2 \end{array}\right)$$

The second case definition for $BSTs$ simply takes the list of Form 1 abstract dSCA State Transition equations supplied as an argument and returns them, with the State Transition equation for the program counter appended. It is therefore defined as:

$$BSTs\left(\begin{array}{l} [], \\ neqs, \\ \beta s, \\ \gamma s, \\ \delta s \end{array}\right) = \left(\begin{array}{l} neqs, \\ CreateVF\left(\begin{array}{l} V_{pc}(t+1,a,x), \\ cond\left(\begin{array}{l} V_{pc}(t,a,x) = 0, \\ mod(add(V_{pc}(t,a,x),1),1), \\ null \end{array}\right) \end{array}\right) \end{array}\right)$$

10.1.6 Transformation Process

Each of the operations above need to be coordinated together so that a SCA can be transformed into an abstract dSCA. The *Create_adSCA* operation is provided to do this, it is given as:

$$Transform : SCAAlgebra \rightarrow adSCAAlgebra$$

The operation takes the source SCA and the name of the abstract dSCA, along

with the number of modules and number of inputs in the source SCA. It is defined:

$$Transform(SCA_{src}) = CreateadSCA \left(\begin{array}{l} GetName(SCA_{src}), \\ SCAAlgebra\ , \\ [], \\ [], \\ VFOp, \\ \gamma_0 : N^2 \rightarrow \{M, S, U\}, \\ \beta_0 : N^2 \rightarrow N, \\ \delta Op\ , \\ CreateISVFs(SCA_{src}), \\ CreateSTVFs(SCA_{src}), \\ Create\gamma s(SCA_{src}), \\ Create\beta s(SCA_{src}), \\ Create\delta s(SCA_{src}) \end{array} \right)$$

where

$$VFOp = \left(\begin{array}{l} V_0 : T \times M_A^n \times M_A^{k+1} \rightarrow M_A, \\ \vdots, \\ V_{k+1} : T \times M_A^n \times M_A^{k+1} \rightarrow M_A \end{array} \right)$$

$$\delta Op = \left(\begin{array}{l} \delta_{0,0,0} : T \times M_A^n \times M_A^{k+1} \rightarrow T, \\ \vdots, \\ \delta_{i,j,0} : T \times M_A^n \times M_A^{k+1} \rightarrow T \end{array} \right)$$

and

$$k = num\_mod(Src\_SCA)$$
$$j = Get\_MaxA(Src\_SCA)$$
$$n = num\_inp(Src\_SCA)$$

The complete algebraic specification for the SCA to Form 1 abstract dSCA transformation is given in Appendix F.

10.2 Correctness

Theorem 10.2.1. *The transformation of SCA to a Form 1 abstract dSCA preserves correctness.*

The original SCA and transformed result, the Form 1 abstract dSCA, exist in a hierarchy and it is possible to show that the transformation is correct by considering Poole, Holden and Tucker's work on hierarchy of Spatially Expanded Systems.

Let N_{SCA} be $\mathbb{N}_k^{SCA} > 0$ module source SCA network with $n^{SCA} > 0$ sources processing data from a set M_A^{SCA} against a global clock T^{SCA}

Let N_{dSCA} be $\mathbb{N}_k^{dSCA} > 1$ module source SCA network with $n^{dSCA} > 0$ sources processing data from a set M_A^{dSCA} against a global clock T^{dSCA} as generated from N_{SCA} using the SCA to abstract dSCA transformation.

Poole, Holden and Tucker claim that if it is possible to generate appropriate mappings and show the following diagram commutes then the two spatially expanded systems under consideration were correct with respect to each other.

$$\begin{array}{ccccccc}
T_{dSCA} & \times [T_{dSCA} \rightarrow M_{A_{dSCA}}]^{In_{dSCA}} \times & M_{A_{dSCA}}^{Ch_{dSCA}} & \xrightarrow{V_{tgt}} & M_{A_{dSCA}}^{Ch_{dSCA}} \\
\uparrow \lambda & & \uparrow \theta & & \uparrow \phi & & \uparrow \phi \\
Start_{\lambda} & \times [T_{SCA} \rightarrow M_{A_{SCA}}]^{In_{SCA}} \times & M_{A_{SCA}}^{Ch_{SCA}} & \xrightarrow{V_{src}} & M_{A_{src}}^{Ch_{src}}
\end{array}$$

Mappings are needed for four areas:

- spaces;
- clocks;
- global states; and
- input streams.

The mappings are defined as follows:

Spaces. Spaces (modules) in the two networks do not differ for modules m_i where $i \in \mathbb{N}_k^{SCA}$ and $i > 0$. Thus it is appropriate to define the respacing operation $\pi : I_{N_{SCA}} \rightarrow I_{N_{dSCA}}$ as:

$$\pi(i) = i \; for \; 1 < i \leq k$$

Clocks. There is no alteration in timing between the two networks, therefore the retiming between clocks T^{SCA} and T^{dSCA} is $\lambda : T^{SCA} \rightarrow T^{dSCA}$, for $t \in T^{SCA}$ and $s \in T^{dSCA}$ can be appropriately defined as

$$\lambda(t) = s$$

Input Streams. There are no timing or data abstractions require for inputs since these are not altered by the transformation. Thus is it appropriate to define the input stream abstraction $\theta : [T \rightarrow M_A]^{n^{SCA}} \rightarrow [T \rightarrow M_A]^{n^{dSCA}}$ as the identity operation:

$$\begin{aligned} \theta(a)(t) &= a(\lambda(t)) \\ &= a(s) \end{aligned}$$

Global States. It is defined in the transformation that the carrier data set for source SCA and target abstract dSCA are the same, M_A. Thus there is no data abstraction required for consideration. We therefore consider the state abstraction map $\phi : M_A^{Ch_{SCA}} \rightarrow M_A^{Ch_{dSCA}}$ for all states $s \in M_A^{Ch_{SCA}}$ to be defined as follows, for $i \in \mathbb{N}_k^{SCA}$:

$$\phi(s)(i) = s(i)$$

Consider now any module m_i in the SCA, it will have two equations in the SCA specification and two corresponding ones in the abstract dSCA specification.

$$V_{\pi(i)}^{SCA}(\lambda(t), \theta(a), \phi(x)) = V_i^{dSCA}(s, a, x)$$

We now compare the two networks in the Initial State and the State Transition phases.

Initial State Phase Consider the output of the module at time $t = 0$ then

$$\begin{aligned} V_{\pi(i)}^{SCA}(\lambda(0), \theta(a), \phi(x)) &= \phi(x_i) \\ &= x_i \\ &= V_i^{dSCA}(0, a, x) \end{aligned}$$

The transformation process for SCA to abstract dSCA creates the dSCA initial state equation by simply copying it from source SCA, thus the above is correct.

State Transition Phase Consider the output at time $t = t + 1$ then

$$V_{\pi(i)}^{SCA}(\lambda(t+1), \theta(a), \phi(x)) = f_i(b_1, \ldots, b_{n(i)})$$

For $j \in 1, \ldots, n(i)$ then the input is either from another module in the network or is from an input, thus

$$b_j = \begin{cases} V_{\pi(q)}(\delta_{\pi(q),j}(\lambda(t), a, x), a, x) & \text{if } \beta(\pi(i), j) = q \wedge \gamma(\pi(i), j) = M \\ \phi(a_q(t)) & \text{if } \beta(\pi(i), j) = q \wedge \gamma(\pi(i), j) = S \end{cases}$$

or rewritten as

$$b_j = \begin{cases} V_q(s, a, x) & \text{if } \beta(i, j) = \wedge\gamma(i, j) = M \\ a_q(s)) & \text{if } \beta(i, j) = \wedge\gamma(i, j) = S \end{cases}$$

thus the mapping functions provide the same functionality as the process for rewiring in the transformation specification. Finally, the state transition equation in the dSCA is copied directly from the same numbered module and included in a conditional statement:

$$cond(V_{pc}(s, a, x) = 0, f_i, null)$$

The value of V_{pc} is always zero (consider the definition of the program counter values in a Form 1 abstract dSCA), therefore m_i in the abstract dSCA always executes f_i at all times $t > 0$ and it can therefore be written that:

$$V_{\pi(i)}^{SCA}(\lambda(t+1), \theta(a), \phi(x)) = V_i^{dSCA}(s+1, a, x)$$

10.3 Generalised Railroad Crossing Problem SCA Transformed to an Abstract dSCA

This section contains a manual walk through of the transformation of the SCA solution to the GRCP using the specification provided in the previous chapter. The input to the process is the algebraic specification of the SCA, as shown in Annex B, and the first step is to confirm that it meets the prerequisites for transformation.

The discussion following shows the transformation in process.

γ-Wiring Equation Transformation

The γ-wiring functions transform by way of the $Create\gamma s$ operation that takes the SCA as an input. This is defined:

$$Create\gamma s\left(\ source\_SCA \ \right) = B\gamma s \left(\begin{array}{l} num\_mod(source\_SCA), \\ GetMaxA(source\_SCA), \\ Get\gamma Eqs(source\_SCA), \\ [] \end{array} \right)$$

The call to $Get\gamma Eqs$ extracts the following details from the source SCA:

$$\begin{array}{llll}
\gamma(1,1) = M, & \gamma(7,1) = M, & \gamma(15,1) = S, & \gamma(27,1) = M, \\
\gamma(1,2) = M, & \gamma(7,2) = M, & \gamma(15,2) = M, & \gamma(27,2) = M, \\
\gamma(1,3) = M, & \gamma(10,1) = M, & \gamma(22,1) = M, & \gamma(28,1) = M, \\
\gamma(2,1) = M, & \gamma(10,2) = M, & \gamma(22,2) = M, & \gamma(28,2) = M, \\
\gamma(2,2) = M, & \gamma(11,1) = S, & \gamma(23,1) = M, & \gamma(29,1) = S, \\
\gamma(4,1) = M, & \gamma(11,2) = M, & \gamma(23,2) = M, & \gamma(29,2) = S, \\
\gamma(4,2) = M, & \gamma(12,1) = M, & \gamma(24,1) = M, & \gamma(31,1) = S, \\
\gamma(4,3) = M, & \gamma(12,2) = M, & \gamma(24,2) = M, & \gamma(31,2) = S, \\
\gamma(5,1) = M, & \gamma(13,1) = S, & \gamma(25,1) = M, & \gamma(33,1) = S, \\
\gamma(5,2) = M, & \gamma(13,2) = M, & \gamma(25,2) = M, & \gamma(33,2) = S, \\
\gamma(6,1) = M, & \gamma(14,1) = M, & \gamma(26,1) = M, & \gamma(35,1) = S, \\
\gamma(6,2) = M, & \gamma(14,2) = M, & \gamma(26,2) = M, & \gamma(35,2) = S,
\end{array}$$

and the *num_mod* operation identifies that there are 36 modules in the source SCA, and that the largest number of inputs to any one module in the SCA is 3. It is therefore possible to rewrite the $Create\gamma s$ call as:

$$Create\gamma s\left(\ source\_SCA\ \right) = B\gamma s\left(\begin{array}{l} 36, \\ 3, \\ Get\gamma Eqs(source\_SCA), \\ [] \end{array}\right)$$

Subsequently the call to $B\gamma s$ operation will result in the invocation of the first case (where the module number is greater than 1):

$$B\gamma s\left(\begin{array}{l} 36, \\ 3, \\ eqs, \\ neqs, \end{array}\right) = B\gamma s\left(\begin{array}{l} 35, \\ 3, \\ eqs, \\ \left(B\gamma\left(\begin{array}{l} 3, \\ 36, \\ eqs, \end{array}\right), neqs\right) \end{array}\right)$$

The call to $B\gamma$ expands as:

$$B\gamma\begin{pmatrix}3,\\36,\\eqs,\end{pmatrix}=\begin{pmatrix}Build\gamma\begin{pmatrix}\gamma_0,\\36,\\0,\\M\end{pmatrix},B\gamma Args\begin{pmatrix}3,\\36,\\eqs,\\[]\end{pmatrix}\end{pmatrix}$$

Following the call to $B\gamma Args$ results in the following list:

$$\gamma_0(36,1)=U,$$
$$\gamma_0(36,2)=U,$$
$$\gamma_0(36,3)=U$$

since module 36 is not wired to anything in the source module. The return from $B\gamma$ is the list:

$$\gamma_0(36,0)=M,$$
$$\gamma_0(36,1)=U,$$
$$\gamma_0(36,2)=U,$$
$$\gamma_0(36,3)=U$$

The call to $B\gamma s$ becomes:

$$B\gamma s\begin{pmatrix}35,\\3,\\eqs,\\ \gamma_0(36,0)=M,\\ \gamma_0(36,1)=U,\\ \gamma_0(36,2)=U,\\ \gamma_0(36,3)=U,\end{pmatrix},$$

which in turn would see another call to the $B\gamma$ operation of:

$$B\gamma\begin{pmatrix}3,\\35,\\eqs,\end{pmatrix}=\begin{pmatrix}Build\gamma\begin{pmatrix}\gamma_0,\\35,\\0,\\M\end{pmatrix},B\gamma Args\begin{pmatrix}3,\\35,\\eqs,\\[]\end{pmatrix}\end{pmatrix}$$

When considering the call to $B\gamma Args$ this time, the first result (for argument 3) is still unwired, however the recursive call considers a situation where there are definitions in the original SCA. The result from $B\gamma Args$ this time will be the list:

$$\gamma_0(35,1) = S,$$
$$\gamma_0(35,2) = S,$$
$$\gamma_0(35,3) = U$$

and subsequently the return from $B\gamma$ will be the list:

$$\gamma_0(36,0) = M,$$
$$\gamma_0(36,1) = S,$$
$$\gamma_0(36,2) = S,$$
$$\gamma_0(36,3) = U$$

The overall result of $Create\gamma s$ is:

$$\begin{array}{llll}
\gamma_0(1,0) = M, & \gamma_0(1,1) = M, & \gamma_0(1,2) = M, & \gamma_0(1,3) = M, \\
\gamma_0(2,0) = M, & \gamma_0(2,1) = M, & \gamma_0(2,2) = M, & \gamma_0(2,3) = U, \\
\gamma_0(3,0) = M, & \gamma_0(3,1) = U, & \gamma_0(3,2) = U, & \gamma_0(3,3) = U, \\
\gamma_0(4,0) = M, & \gamma_0(4,1) = M, & \gamma_0(4,2) = M, & \gamma_0(4,3) = M, \\
\gamma_0(5,0) = M, & \gamma_0(5,1) = M, & \gamma_0(5,2) = M, & \gamma_0(5,3) = U, \\
\gamma_0(6,0) = M, & \gamma_0(6,1) = M, & \gamma_0(6,2) = M, & \gamma_0(6,3) = U, \\
\gamma_0(7,0) = M, & \gamma_0(7,1) = M, & \gamma_0(7,2) = M, & \gamma_0(7,3) = U, \\
\gamma_0(8,0) = M, & \gamma_0(8,1) = U, & \gamma_0(8,2) = U, & \gamma_0(8,3) = U, \\
\gamma_0(9,0) = M, & \gamma_0(9,1) = U, & \gamma_0(9,2) = U, & \gamma_0(9,3) = U, \\
\gamma_0(10,0) = M, & \gamma_0(10,1) = M, & \gamma_0(10,2) = M, & \gamma_0(10,3) = U, \\
\gamma_0(11,0) = M, & \gamma_0(11,1) = S, & \gamma_0(11,2) = M, & \gamma_0(11,3) = U, \\
\gamma_0(12,0) = M, & \gamma_0(12,1) = M & \gamma_0(12,2) = M, & \gamma_0(12,3) = U, \\
\gamma_0(13,0) = M, & \gamma_0(13,1) = S, & \gamma_0(13,2) = M, & \gamma_0(13,3) = U, \\
\gamma_0(14,0) = M, & \gamma_0(14,1) = M, & \gamma_0(14,2) = M, & \gamma_0(14,3) = U, \\
\gamma_0(15,0) = M, & \gamma_0(15,1) = S, & \gamma_0(15,2) = M, & \gamma_0(15,3) = U, \\
\gamma_0(16,0) = M, & \gamma_0(16,1) = U, & \gamma_0(16,2) = U, & \gamma_0(16,3) = U, \\
\gamma_0(17,0) = M, & \gamma_0(17,1) = U, & \gamma_0(17,2) = U, & \gamma_0(17,3) = U, \\
\gamma_0(18,0) = M, & \gamma_0(18,1) = U, & \gamma_0(18,2) = U, & \gamma_0(18,3) = U, \\
\gamma_0(19,0) = M, & \gamma_0(19,1) = U, & \gamma_0(19,2) = U, & \gamma_0(19,3) = U,
\end{array}$$

$$
\begin{array}{llll}
\gamma_0(20,0) = M, & \gamma_0(20,1) = U, & \gamma_0(20,2) = U, & \gamma_0(20,3) = U, \\
\gamma_0(21,0) = M, & \gamma_0(21,1) = U, & \gamma_0(21,2) = U, & \gamma_0(21,3) = U, \\
\gamma_0(22,0) = M, & \gamma_0(22,1) = M, & \gamma_0(22,2) = M, & \gamma_0(22,3) = U, \\
\gamma_0(23,0) = M, & \gamma_0(23,1) = M, & \gamma_0(23,2) = M, & \gamma_0(23,3) = U, \\
\gamma_0(24,0) = M, & \gamma_0(24,1) = M, & \gamma_0(24,2) = M, & \gamma_0(24,3) = U, \\
\gamma_0(25,0) = M, & \gamma_0(25,1) = M, & \gamma_0(25,2) = M, & \gamma_0(25,3) = U, \\
\gamma_0(26,0) = M, & \gamma_0(26,1) = M, & \gamma_0(26,2) = M, & \gamma_0(26,3) = U, \\
\gamma_0(27,0) = M, & \gamma_0(27,1) = M, & \gamma_0(27,2) = M, & \gamma_0(27,3) = U, \\
\gamma_0(28,0) = M, & \gamma_0(28,1) = M, & \gamma_0(28,2) = M, & \gamma_0(28,3) = U, \\
\gamma_0(29,0) = M, & \gamma_0(29,1) = S, & \gamma_0(29,2) = S, & \gamma_0(29,3) = U. \\
\gamma_0(30,0) = M, & \gamma_0(30,1) = U, & \gamma_0(30,2) = U, & \gamma_0(30,3) = U, \\
\gamma_0(31,0) = M, & \gamma_0(31,1) = S, & \gamma_0(31,2) = S, & \gamma_0(31,3) = U, \\
\gamma_0(32,0) = M, & \gamma_0(32,1) = U, & \gamma_0(32,2) = U, & \gamma_0(32,3) = U, \\
\gamma_0(33,0) = M, & \gamma_0(33,1) = S, & \gamma_0(33,2) = S, & \gamma_0(33,3) = U, \\
\gamma_0(34,0) = M, & \gamma_0(34,1) = U, & \gamma_0(34,2) = U, & \gamma_0(34,3) = U, \\
\gamma_0(35,0) = M, & \gamma_0(35,1) = S, & \gamma_0(35,2) = S, & \gamma_0(35,3) = U, \\
\gamma_0(36,0) = M, & \gamma_0(36,1) = U, & \gamma_0(36,2) = U, & \gamma_0(36,3) = U, \\
\gamma_0(pc,0) = M &&&
\end{array}
$$

β Transformation

The β-wiring functions from the SCA are transformed into the following Form 1 abstract dSCA β-wiring functions:

$$
\begin{array}{llll}
\beta_0(1,0) = pc, & \beta_0(1,1) = 2, & \beta_0(1,2) = 3, & \beta_0(1,3) = 4, \\
\beta_0(2,0) = pc, & \beta_0(2,1) = 5, & \beta_0(2,2) = 6, & \beta_0(2,3) = \omega, \\
\beta_0(3,0) = pc, & \beta_0(3,1) = \omega, & \beta_0(3,2) = \omega, & \beta_0(3,3) = \omega, \\
\beta_0(4,0) = pc, & \beta_0(4,1) = 7, & \beta_0(4,2) = 8, & \beta_0(4,3) = 9, \\
\beta_0(5,0) = pc, & \beta_0(5,1) = 10, & \beta_0(5,2) = 11, & \beta_0(5,3) = \omega, \\
\beta_0(6,0) = pc, & \beta_0(6,1) = 12, & \beta_0(6,2) = 13, & \beta_0(6,3) = \omega, \\
\beta_0(7,0) = pc, & \beta_0(7,1) = 14, & \beta_0(7,2) = 15, & \beta_0(7,3) = \omega, \\
\beta_0(8,0) = pc, & \beta_0(8,1) = \omega, & \beta_0(8,2) = \omega, & \beta_0(8,3) = \omega, \\
\beta_0(9,0) = pc, & \beta_0(9,1) = \omega, & \beta_0(9,2) = \omega, & \beta_0(9,3) = \omega, \\
\beta_0(10,0) = pc, & \beta_0(10,1) = 22, & \beta_0(10,2) = 16, & \beta_0(10,3) = \omega, \\
\beta_0(11,0) = pc, & \beta_0(11,1) = 9, & \beta_0(11,2) = 17, & \beta_0(11,3) = \omega, \\
\beta_0(12,0) = pc, & \beta_0(12,1) = 22, & \beta_0(12,2) = 18, & \beta_0(12,3) = \omega, \\
\beta_0(13,0) = pc, & \beta_0(13,1) = 9, & \beta_0(13,2) = 19, & \beta_0(13,3) = \omega, \\
\beta_0(14,0) = pc, & \beta_0(14,1) = 22, & \beta_0(14,2) = 20, & \beta_0(14,3) = \omega,
\end{array}
$$

$$
\begin{array}{llll}
\beta_0(15,0) = pc, & \beta_0(15,1) = 9, & \beta_0(15,2) = 21, & \beta_0(15,3) = \omega, \\
\beta_0(16,0) = pc, & \beta_0(16,1) = \omega, & \beta_0(16,2) = \omega, & \beta_0(16,3) = \omega, \\
\beta_0(17,0) = pc, & \beta_0(17,1) = \omega, & \beta_0(17,2) = \omega, & \beta_0(17,3) = \omega, \\
\beta_0(18,0) = pc, & \beta_0(18,1) = \omega, & \beta_0(18,2) = \omega, & \beta_0(18,3) = \omega, \\
\beta_0(19,0) = pc, & \beta_0(19,1) = \omega, & \beta_0(19,2) = \omega, & \beta_0(19,3) = \omega, \\
\beta_0(20,0) = pc, & \beta_0(20,1) = \omega, & \beta_0(20,2) = \omega, & \beta_0(20,3) = \omega, \\
\beta_0(21,0) = pc, & \beta_0(21,1) = \omega, & \beta_0(21,2) = \omega, & \beta_0(21,3) = \omega, \\
\beta_0(22,0) = pc, & \beta_0(22,1) = 23, & \beta_0(22,2) = 24, & \beta_0(22,3) = \omega, \\
\beta_0(23,0) = pc, & \beta_0(23,1) = 25, & \beta_0(23,2) = 26, & \beta_0(23,3) = \omega, \\
\beta_0(24,0) = pc, & \beta_0(24,1) = 27, & \beta_0(24,2) = 28, & \beta_0(24,3) = \omega, \\
\beta_0(25,0) = pc, & \beta_0(25,1) = 29, & \beta_0(25,2) = 30, & \beta_0(25,3) = \omega, \\
\beta_0(26,0) = pc, & \beta_0(26,1) = 31, & \beta_0(26,2) = 32, & \beta_0(26,3) = \omega, \\
\beta_0(27,0) = pc, & \beta_0(27,1) = 33, & \beta_0(27,2) = 34, & \beta_0(27,3) = \omega, \\
\beta_0(28,0) = pc, & \beta_0(28,1) = 35, & \beta_0(28,2) = 36, & \beta_0(28,3) = \omega, \\
\beta_0(29,0) = pc, & \beta_0(29,1) = 1, & \beta_0(29,2) = 2, & \beta_0(29,3) = \omega. \\
\beta_0(30,0) = pc, & \beta_0(30,1) = \omega, & \beta_0(30,2) = \omega, & \beta_0(30,3) = \omega, \\
\beta_0(31,0) = pc, & \beta_0(31,1) = 3, & \beta_0(31,2) = 4, & \beta_0(31,3) = \omega, \\
\beta_0(32,0) = pc, & \beta_0(32,1) = \omega, & \beta_0(32,2) = \omega, & \beta_0(32,3) = \omega, \\
\beta_0(33,0) = pc, & \beta_0(33,1) = 5, & \beta_0(33,2) = 6, & \beta_0(33,3) = \omega, \\
\beta_0(34,0) = pc, & \beta_0(34,1) = \omega, & \beta_0(34,2) = \omega, & \beta_0(34,3) = \omega, \\
\beta_0(35,0) = pc, & \beta_0(35,1) = 7, & \beta_0(35,2) = 8, & \beta_0(35,3) = \omega, \\
\beta_0(36,0) = pc, & \beta_0(36,1) = \omega, & \beta_0(36,2) = \omega, & \beta_0(36,3) = \omega, \\
\beta_0(pc,1) = pc & & &
\end{array}
$$

Delay Function Transformation (δ)

The delay functions are transformed such that for $i = 1, \ldots, 36$ and $j = 0, \ldots, 3$:

$$\delta_{i,j,0}(t, a, x) = t - 1$$

and the new delay function for the program counter modules is created as:

$$\delta_{pc,0,0}(t, a, x) = t - 1$$

Initial State Equation Transformation

The transformed Initial State equations for the Form 1 abstract dSCA are a copy of the SCA Initial State equations with the addition of an Initial State equations for the

programme counter. The resultant Initial State equations are therefore given as:

$$
\begin{array}{lll}
V_1(0,a,x) = stay & V_2(0,a,x) = true & V_3(0,a,x) = stay \\
V_4(0,a,x) = up & V_5(0,a,x) = true & V_6(0,a,x) = false \\
V_7(0,a,x) = false & V_8(0,a,x) = down & V_9(0,a,x) = up \\
V_{10}(0,a,x) = true & V_{11}(0,a,x) = true & V_{12}(0,a,x) = false \\
V_{13}(0,a,x) = false & V_{14}(0,a,x) = false & V_{15}(0,a,x) = true \\
V_{16}(0,a,x) = false & V_{17}(0,a,x) = 90 & V_{18}(0,a,x) = true \\
V_{19}(0,a,x) = 0 & V_{20}(0,a,x) = true & V_{21}(0,a,x) = 0 \\
V_{22}(0,a,x) = false & V_{23}(0,a,x) = false & V_{24}(0,a,x) = false \\
V_{25}(0,a,x) = false & V_{26}(0,a,x) = false & V_{27}(0,a,x) = false \\
V_{28}(0,a,x) = false & V_{29}(0,a,x) = 0 & V_{30}(0,a,x) = 0 \\
V_{31}(0,a,x) = 0 & V_{32}(0,a,x) = 0 & V_{33}(0,a,x) = 0 \\
V_{34}(0,a,x) = 0 & V_{35}(0,a,x) = 0 & V_{36}(0,a,x) = 0 \\
V_{pc}(0,a,x) = 0 & &
\end{array}
$$

State Transition Equation Transformation

Transforming the State Transition equations commences with a call to the $CreateSTVFs$ operation:

$$
CreateSTVFs(Source\_SCA) = BSTs \left(\begin{array}{l}
GetEqIV(Source\_SCA), \\
[], \\
Create\beta s\left(Get\beta Ops(Source\_SCA)\right), \\
Create\gamma s\left(Get\gamma Ops(Source\_SCA)\right), \\
Create\delta s\left(Get\delta Ops(Source\_SCA)\right),
\end{array} \right)
$$

where the last three arguments to the call to the $BSTs$ operations are the lists obtained above and the first argument is the list of State Transition Equations extracted from the SCA algebraic specification.

Consider the call to $BSTs$, this will initially result in a recursive call to itself as follows:

$$
BSTs \left(\begin{array}{l}
(e, eqs), \\
[], \\
\beta s, \\
\gamma s, \\
\delta s
\end{array} \right) = BSTs \left(\begin{array}{l}
eqs, \\
(BST(e, \beta s, \gamma s, \delta s), []), \\
\beta s, \\
\gamma s, \\
\delta s
\end{array} \right)
$$

where e is the State Transition equation:

$$V_1(t+1,a,x) = cond(V_2(t,a,x), V_3(t,a,x), V_4(t,a,x))$$

The call to BST within this definition is as follows:

$$BST\begin{pmatrix} e, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = CreateVF\left(\begin{array}{l} RetTerm(e,1), \\ \left(\begin{array}{lll} & m_1' = 0, & \\ cond & rewire & \begin{pmatrix} RetTerm(e,2), \\ newVFTerm, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, \\ & null & \end{array}\right) \end{array}\right)$$

which can be rewritten as:

$$BST\begin{pmatrix} e, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = CreateVF\left(\begin{array}{l} V_1(t+1,a,x), \\ \left(\begin{array}{lll} & V_{pc}(t,a,x) = 0, & \\ cond & rewire & \begin{pmatrix} cond\begin{pmatrix} V_2(t,a,x), \\ V_3(t,a,x), \\ V_4(t,a,x) \end{pmatrix}, \\ & & 1, \\ & & \beta s, \\ & & \gamma s, \\ & & \delta s \end{pmatrix}, \\ & null & \end{array}\right) \end{array}\right)$$

the call to *rewire* is simplified as:

$$rewire\begin{pmatrix} cond\begin{pmatrix} V_2(t,a,x), \\ V_3(t,a,x), \\ V_4(t,a,x) \end{pmatrix}, \\ 1, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = cond\begin{pmatrix} V_2(t,a,x), \\ V_3(t,a,x), \\ V_4(t,a,x) \end{pmatrix}$$

and therefore the call to BST can be rewritten as:

$$BST\begin{pmatrix} e, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = CreateVF\begin{pmatrix} V_1(t+1,a,x), \\ \begin{pmatrix} cond\begin{pmatrix} V_{pc}(t,a,x)=0, \\ cond\begin{pmatrix} V_2(t,a,x), \\ V_3(t,a,x), \\ V_4(t,a,x) \end{pmatrix}, \\ null \end{pmatrix} \end{pmatrix} \end{pmatrix}$$

finally resulting in:

$$V_1(t+1,a,x) = cond\begin{pmatrix} V_{pc}(t,a,x)=0, \\ cond\begin{pmatrix} V_2(t,a,x), \\ V_3(t,a,x), \\ V_4(t,a,x) \end{pmatrix}, \\ null \end{pmatrix}$$

or written in a more natural form:

$$V_1(t+1,a,x) = cond(V_2(t,a,x), V_3(t,a,x), V_4(t,a,x)) \text{ if } \mathrm{V_{pc}(t,a,x)} = 0$$

Finally, the recursive part of the $BSTs$ operation produces the State Transition equation for the program counter of (written in a more natural form):

$$V_1(t+1,a,x) = mod(add(V_{pc}(t,a,x),1),1) \text{ if } \mathrm{V_{pc}(t,a,x)} = 0$$

The complete list of transformed, and simplified, State Transition equations are as follows:

$$
\begin{array}{ll}
V_1(t+1,a,x) = cond(V_2(t,a,x), V_3(t,a,x), V_4(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_2(t+1,a,x) = or(V_5(t,a,x), V_6(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_3(t+1,a,x) = start & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_4(t+1,a,x) = cond(V_7(t,a,x), V_8(t,a,x), V_9(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_5(t+1,a,x) = and(V_{10}(t,a,x), V_{11}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_6(t+1,a,x) = and(V_{12}(t,a,x), V_{13}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_7(t+1,a,x) = and(V_{14}(t,a,x), V_{15}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_8(t+1,a,x) = down & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_9(t+1,a,x) = up & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{10}(t+1,a,x) = eq(V_{22}(t,a,x), V_{16}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{11}(t+1,a,x) = eq(a_9(t), V_{17}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{12}(t+1,a,x) = eq(V_{22}(t,a,x), V_{18}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{13}(t+1,a,x) = eq(a_9(t), V_{19}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{14}(t+1,a,x) = eq(V_{22}(t,a,x), V_{20}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{15}(t+1,a,x) = gt(a_9(t), V_{21}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{16}(t+1,a,x) = false & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{17}(t+1,a,x) = 90 & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{18}(t+1,a,x) = true & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{19}(t+1,a,x) = 0 & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{20}(t+1,a,x) = true & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{21}(t+1,a,x) = 0 & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{22}(t+1,a,x) = or(V_{23}(t,a,x), V_{24}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{23}(t+1,a,x) = or(V_{25}(t,a,x), V_{26}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{24}(t+1,a,x) = or(V_{27}(t,a,x), V_{28}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{25}(t+1,a,x) = gt(V_{29}(t,a,x), V_{30}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{26}(t+1,a,x) = gt(V_{31}(t,a,x), V_{32}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{27}(t+1,a,x) = gt(V_{33}(t,a,x), V_{34}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{28}(t+1,a,x) = gt(V_{35}(t,a,x), V_{36}(t,a,x)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{29}(t+1,a,x) = sub(a_1(t), a_2(t)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{30}(t+1,a,x) = 0 & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{31}(t+1,a,x) = sub(a_3(t), a_4(t)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{32}(t+1,a,x) = 0 & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{33}(t+1,a,x) = sub(a_5(t), a_6(t)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{34}(t+1,a,x) = 0 & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{35}(t+1,a,x) = sub(a_7(t), a_8(t)) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{36}(t+1,a,x) = 0 & \text{if } \mathrm{V_{pc}(t,a,x)} = 0, \\
V_{pc}(t+1,a,x) = mod(add(V_{pc}(t,a,x), 1), 1) & \text{if } \mathrm{V_{pc}(t,a,x)} = 0
\end{array}
$$

The complete Form 1 abstract dSCA created from the transformation of the SCA at Appendix B is shown in Appendix C.

10.4 Correctness of Example

The generated Form 1 abstract dSCA created from transforming the SCA can be seen to be the same as the Form 1 abstract dSCA given in Chapter 8.3 - the semantic proof of correctness given in that chapter shows that this abstract dSCA is a correct implementation of a solution to the GRCP.

The notion that the global behaviour of SCA abstracts that of the abstract form 1 dSCA is now formalised. Let V_{src} and V_{tgt} be the global state functions determined from the channel state functions of these 2 SCAs.

Conjecture It is believed that the following diagram commutes:

$$\begin{array}{ccccc} T_{tgt} & \times [T_{tgt} \rightarrow A_{tgt}]^{In_{tgt}} \times & A_{tgt}^{Ch_{tgt}} & \xrightarrow{V_{tgt}} & A_{tgt}^{Ch_{tgt}} \\ \uparrow \lambda & \uparrow \theta & \uparrow \phi & & \uparrow \phi \\ Start_{\lambda} & \times [T_{src} \rightarrow A_{src}]^{In_{src}} \times & A_{src}^{Ch_{src}} & \xrightarrow{V_{src}} & A_{src}^{Ch_{src}} \end{array}$$

We have seen from the definition of the correctness of transformation that this is true. Given the construction of appropriate mappings for:

- spaces;
- clocks;
- global states; and
- input streams.

We rely on Theorem 10.2.1 for proof of correctness. A quick examination of the specifications for the SCA and Form 1 abstract dSCA in the appendices demonstrates the theorem holds.

10.5 Concluding Comments

This chapter has demonstrated the techniques required for mapping a SCA to an abstract dSCA with a defining shape that represents the SCA. Using the SCA solution to the GRC Problem, the transformation has been demonstrated by producing an abstract dSCA representation of the GRC Problem.

10.6 Sources

The definition of the transformation and the walk through of the example is all my own work.

Chapter 11

Abstract dSCA to abstract dSCA

Purpose of Transformation
To transform an abstract dSCA with a defining shape of $\nabla = (n_1, m_1)$ *to an abstract dSCA with defining shape* $\nabla = (n_2, m_2)$ *using a mapping function* Ξ *which shows how operations at program counter value* m_1 *on module* n_1 *in Network* N_1 *are transposed to execute at program counter value* m_2 *on module* n_2 *in Network* N_2.

11.1 Process

This chapter highlights the key points in the process of transforming an abstract dSCA with defining shape $\nabla = (n_1, m_1)$ to an abstract dSCA with an defining shape of $\nabla = (n_2, m_2)$. Full details of this transformation can be found in G.

Transformations are required for the following equation lists within a supplied abstract SCA algebraic specification are covered:

1. Wiring Functions;

2. Delay Functions;

3. Initial State Equations; and

4. State Transition Equations.

After discussing the necessary transformations they are used to transform the abstract dSCA produced in the last chapter to an abstract dSCA with defining shape of $\nabla = (1, k)$. Subsequently the correctness of the transformed Form 2 abstract dSCA is discussed.

11.1.1 Prerequisites

- The source network, N_1 has $k_1 > 1$ modules and $Max_{n_1} > 0$ component specifications in its modules definitions;

- The object network, N_2 has $k_2 > 1$ modules and $Max_{n_2} > 0$ component specifications in its modules definitions;

- The defining size of N_2 must be equal to or greater than the defining size of N_1, i.e. $\triangle(N_2) \geq \triangle(1)$;

- There exists the total mapping, Ξ given as:

$$\Xi : \mathbb{N}_{k_1} \times \mathbb{N}_{pc_1} \rightarrow \mathbb{N}_{k_2} \times \mathbb{N}_{pc_2}$$

  that maps modules and execution orders of N_1 to modules and execution orders of N_2; and

- There exists the inverse mapping Ξ^{-1}, given as:

$$\Xi^{-1} : \mathbb{N}_{k_2} \times \mathbb{N}_{pc_2} \rightsquigarrow \mathbb{N}_{k_1} \times \mathbb{N}_{pc_1}$$

  (Note that this mapping may not be total, since some functional components of N_2 may be the undefined operation used to ensure synchronicity of the network).

11.1.2 Mapping Function

The provision of a mapping function is a fundamental prerequisite before this transformation can occur. Its purpose is to provide a total mapping between when a particular function executed on a particular module in the source network and what module and when it will execute on the target network. It is a simple list of equations containing two pairs:

$$(i_1, pc\_val_1) = (i_2, pc\_val_2)$$

and must be defined for all values $i_1 \in \mathbb{N}\_k_1$ of the k-module source abstract dSCA and $pc\_val_1 \in \{0, \ldots, Max_{n_1} - 1\}$. The mapping is denoted as Ξ, and has the (partial) inverse Ξ^{-1}. There is no need to map the program counter module.

11.1.3 Wiring Functions

Unlike the previous transformation, wiring functions will alter values radically to provide the dynamic retiming and structure necessary to support a re-shaped abstract dSCA. The process of generating the wiring functions is quite simplistic and so this thesis will restrict itself to an informal demonstration (a more formal description is given in G).

γ-wiring Operations

Consider the source abstract dSCA γ-wiring function:

$$\gamma_{pc\_val_1}(i_1, j_1) = z_1$$

the corresponding target abstract dSCA γ-wiring function will be:

$$\gamma_{pc\_val_2}(i_2, j_2) = z_2$$

where $j_1 = j_2$, and $\Xi\,(i_1, pc\_val_1) = (i_2, pc\_val_2)$

The *informal* process of generating target abstract dSCA γ-wiring functions is to walk the structure of the target architecture creating wiring functions for all modules at all values of the program counter for the number of inputs to each module.

- For each module m_i where $i \in \mathbb{N}_{k_2}$ and $i > 0$:
  - For each *pc_val* where $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$:
    - For the o^{th} argument of each module create:

$$\gamma'_{pc\_val}(i, 0) = M$$

    - For each argument where $j \in \{1, \ldots, n_2(i)\}$ create a new γ-wiring function

$$\gamma'_{pv\_val}(i, j) = \begin{cases} Value\ from\ source & \text{if } \Xi^{-1}(i, j) \downarrow \\ U & \text{otherwise} \end{cases}$$

      with the intended meaning that the undefined connection is given if the inverse mapping is not defined, otherwise the appropriate value from the source network is used.
- For module 0 create Max_{N_2} γ-wiring functions wiring m_0 back to itself.

β-wiring Operations

Consider the source abstract dSCA β-wiring function:

$$\beta_{pc\_val_1}(i_1, j_1) = z_1$$

the corresponding target abstract dSCA β-wiring function will be:

$$\beta_{pc\_val_2}(i_2, j_2) = z_2$$

where $j_1 = j_2$ and $\Xi(i_1, pc\_val_1) = (i_2, pc\_val_2)$ The *informal* process of generating target abstract dSCA β-wiring functions is to walk the structure of the target architecture creating wiring functions for all modules at all values of the program counter for the number of inputs to each module:

- For each module m_i where $i \in \mathbb{N}_{k_2}$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$:
    * For the o^{th} argument of each module create:

$$\beta'_{pc\_val}(i, 0) = M$$

    * For each argument where $j \in \{1, \ldots, n_2(i)\}$ create a new β-wiring function

$$\beta'_{pv\_val}(i, j) = \begin{cases} Value\ from\ source & \text{if } \Xi^{-1}(i, j) \downarrow \\ \omega & \text{otherwise} \end{cases}$$

      with the intended meaning that the undefined index is given if the inverse mapping is not defined, otherwise the appropriate value from the source network is used.

- For module 0 create Max_N β-wiring functions to wire m_0 back to itself.

11.1.4 Delay Functions

The delay functions for the source and target abstract dSCA are of the same format, however the derivation of the delay is more complicated than the simple generation of the wiring functions, and thus a more detailed explanation of the derivation is given.

In both networks, it is the intention of the delay function to indicate the time delay between now and the time the result was calculated. In the source abstract dSCA this is given by the defined delay function. For the object abstract dSCA this value needs to be derived from the data available.

Informally, target abstract dSCA functions are produced as follows:

- For each module m_i where $i \in \mathbb{N}_{k_2}$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$:

* We define, for the 0^{th} argument, the unit delay:

$$\delta'_{i,0,pc\_val}(t,a,x) = t-1$$

* For each argument where $j \in \{1,\ldots,n_2(i)\}$ create a new δ-wiring function

$$\delta_{pc\_val}(i,j) = \begin{cases} t - new\_value & \text{if } (\Xi^{-1}(i,pc\_val)) \downarrow \wedge \\ & (\gamma_{pc\_val}(i,j) = M) \\ t - new\_value & \text{if} \\ t-1 & \text{otherwise} \end{cases}$$

- For module 0 create Max_N delay operations of unit length delay to represent the wiring of m_0 back to itself.

The usual recursive functions are defined to walk the structure of the new abstract dSCA, but of particular interest is how the creation of a new *delta* function for particular values of *pc_val*, i and j. The $B\delta$ operation, which is responsible for creating the new delay function for the arg_val^{th} (j^{th}) argument of module *mod_val* at program counter *pc_val*, is called and it is given as:

$$\begin{aligned} B\delta: \; & N^3 \times \delta dSCAEqList \times \gamma dSCAEqList \times \beta dSCAEqList \times \\ & N^2 \times MapEqList^2 \rightarrow \delta dSCAEquation \end{aligned}$$

To provide a definition of $B\delta$ the new value of the delay needs to be generated from the existing knowledge of the two abstract dSCAs. To understand what the delay should be, an understanding of what the module links to is required. If the wiring is to a source, or is unconnected, then the unit delay is generated. This case is identified by considering the target abstract dSCA β-wiring functions, thus the first definition

is given as:

$$B\delta \begin{pmatrix} mod\_val, \\ arg\_val, \\ pc\_val, \\ old\delta s \\ old\gamma s, \\ new\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} = Build\delta \begin{pmatrix} mod\_val, \\ arg\_val, \\ pc\_val, \\ t-1 \end{pmatrix} \; if \; cond_1$$

where:

$$cond_1 = \begin{pmatrix} RetTerm \begin{pmatrix} GetEl \begin{pmatrix} old\gamma s, \\ pc\_val, \\ mod\_val, \\ arg\_val \end{pmatrix}, 2 \end{pmatrix} \neq M \end{pmatrix}$$

In the situation where this condition is not true, i.e. the wiring under consideration is to another module, then the value of the new delay function needs to be calculated. To calculate the new value, the following process is followed:

1. Find the module and program counter value in the source abstract dSCA that relates to the current module and program counter value in the target abstract dSCA, using the inverse mapping function;

2. Identify the module in the source abstract dSCA that produces the value we are interested in from the β-wiring function;

3. Identify the program counter value in the source abstract dSCA that the value we are interested in is calculated from the delay functions;

4. Find the module and program counter in the target dSCA that produces the value we are interested in, using the mapping function; and

5. Calculate the delay between the current value of the program counter and the program counter value from (4).

The module and program counter in the source abstract dSCA is given directly by the inverse mapping function:

$$\Xi^{-1}(mod\_val_2, pc\_val_2) = (mod\_val_1, pc\_val_1)$$

The position of arguments in the functional specification cannot change in the transformation. Thus if arg_val is the argument number under consideration in the target abstract dSCA, then it will also be in the source abstract dSCA. This fact and the β-wiring function in the source abstract dSCA are used to determine the module that produces the value for that argument, in the source SCA:

$$mod\_val_1^{res} = \beta_{pc\_val_1}(mod\_val_1, arg\_val)$$

Using the delay function from the source dSCA, the value of the program counter that the result was calculated at can be determined. It will be the current source program counter value minus the delay value for this argument modulus the value of Max_N in the source abstract dSCA:

$$pc\_val_1^{res} = (pc\_val_1 - (t - \delta^{src}_{mod\_val_1, arg\_val, pc\_val_1}(t, a, x))) \bmod Max_{N_1}$$

It is now possible to determine the value of the program counter in the target abstract dSCA by applying the mapping function to the values $pc\_val_1^{res}$ just determined, and $mod\_val_1^{res}$, and taking the second element of the returned tuple:

$$pc\_val_2^{res} = snd(\Xi(mod\_val_1^{res}, pc\_val_1^{res}))$$

The value of the delay can be worked out from the difference between the program counter in the target abstract dSCA now, and the value of $pc\_val_2^{res}$:

$$(pc\_val - pc\_val_2^{res}) \bmod Max_N^2$$

$B\delta$ is defined for this case as:

$$B\delta \begin{pmatrix} mod\_val, \\ arg\_val, \\ pc\_val, \\ old\delta s \\ old\gamma s, \\ old\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} = Build\delta \begin{pmatrix} mod\_val, \\ arg\_val, \\ pc\_val, \\ t - ((pc\_val - pc\_val_{tgt}^{res}) \bmod Max_N^{tgt}) \end{pmatrix}$$

and:

$$pc\_val_{tgt}^{res} = snd \left(RetTerm \left(GetEl \begin{pmatrix} \Xi, \\ mod\_val_{src}^{res}, \\ pc\_val_{src}^{res} \end{pmatrix}, 2 \right) \right)$$

with:

$$mod\_val_{src}^{res} = fst \left(RetTerm \left(GetEl \begin{pmatrix} old\beta s, \\ pc\_val_{src}, \\ mod\_val_{src}, \\ arg\_val \end{pmatrix}, 2 \right) \right)$$

and:

$$pc\_val_{src}^{res} = pc\_val_{src} - \left(t - GetEl \begin{pmatrix} old\delta s, \\ mod\_val_{src}, \\ arg\_val, \\ pc\_val_{src} \end{pmatrix} \right) \bmod Max_N^{src}$$

where mod_val_{src} and pc_val_{src} are:

$$pc\_val_{src} = snd\left(RetTerm\left(GetEl\left(\begin{array}{l}\Xi^{-1},\\ mod\_val,\\ pc\_val\end{array}\right),2\right)\right)$$

$$mod\_val_{src} = fst\left(RetTerm\left(GetEl\left(\begin{array}{l}\Xi^{-1},\\ mod\_val,\\ pc\_val\end{array}\right),2\right)\right)$$

11.1.5 Initial State Equations

Consider the target abstract dSCA module m_i, its Initial State equations, will be of the form:

$$\begin{aligned} V_i(0,a,x) &= x_{i,0}\\ V_i(0,a,x) &= x_{i,1}\\ &\vdots\\ V_i(0,a,x) &= x_{i,Max_{N_2}-1}\end{aligned}$$

where each value x_{i,pc_val}, where $pc\_val = 0, 1, \ldots, Max_{N_2} - 1$, will either be the undefined element, or will come from some particular module and value of the source abstract dSCA program counter. Values of the source program counter and module are given directly from the mapping function, Ξ.

Informally, the set of Initial State equations is created as follows:

- For each module m_i where $i \in \mathbb{N}_{k_2}$ and $i > 0$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$ create a new Initial State equation:

$$V_i(pc\_val, a, x) = \begin{cases} new\_value & \text{if } \Xi^{-1}(i, pc))\downarrow \\ u & \text{otherwise}\end{cases}$$

- For m_0, the program counter:

– For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$ create a new Initial State equation:

$$V_0(pc\_val, a, x) = (pc\_val + 1) \bmod Max_{N_2}$$

The usual set of operations to perform a recursive walk of the new structure are given, resulting in a call to the $BIVpc$ operation, where the key work is done in this transformation component. It is defined by two cases, the first representing the case where the program counter is greater than zero and the second case is where the program counter is zero.

The first case is defined for two situations, where the inverse mapping is defined (in which case a new equation is created from values in the source abstract dSCA) and where it is not (in which case an equation is created that returns the undefined value u in the appropriate parts of the initial state vector):

$$BIVpc\begin{pmatrix} pc, \\ i, \\ oeqs, \\ neqs, \\ \Xi^{-1} \end{pmatrix} = \begin{cases} BIVpc\begin{pmatrix} pc-1, \\ i, \\ \begin{pmatrix} new\_val, \\ neqs \end{pmatrix}, \\ \Xi^{-1} \end{pmatrix} & \text{if } \Xi^{-1}(i, pc) \downarrow \\ BIVpc\begin{pmatrix} pc-1, \\ i, \\ \begin{pmatrix} BuildIV\begin{pmatrix} i, \\ pc, \\ u \end{pmatrix}, \\ neqs \end{pmatrix}, \\ \Xi^{-1} \end{pmatrix} & \text{if } \Xi^{-1}(i, pc) \uparrow \end{cases}$$

where:

$$new\_val = BuildIV\begin{pmatrix} i, \\ pc, \\ RetTerm\left(GetEl\left(oeqs, RetTerm(GetEl(\Xi^{-1}, i, pc), 2)\right), 2\right) \end{pmatrix}$$

The second case of $BIVpc$, where the program counter is zero is the simple case of creating the equation for that value of the program counter and appending it to the list of already generated Initial State equations:

$$BIVpc\begin{pmatrix} 0, \\ i, \\ oeqs, \\ neqs, \\ \Xi^{-1} \end{pmatrix} = \begin{pmatrix} BuildIV\begin{pmatrix} i, \\ RetTerm\begin{pmatrix} GetEl(oeqs, \Xi^{-1}(i,0)), \\ 2 \end{pmatrix} \end{pmatrix}, \\ neqs \end{pmatrix}$$

11.1.6 State Transition Equations

Consider the target abstract dSCA module m_i, its State Transition equations, will be of the form:

$$V_i(t+1,a,x) = \begin{cases} f_{i,0}(\ldots) & \text{if } pc = 0 \\ \ldots & \\ f_{i,Max_N-1}(\ldots) & \text{if } pc = Max_N - 1 \end{cases}$$

where each functional specification component f_{i,pc_val}, for values of $pc\_val = 0, 1, \ldots, Max_N - 1$, will either be the undefined element, or will be the component specification extracted from some particular module and value of the source abstract dSCA program counter in the source abstract dSCA. In a similar manner to creating the Initial State equations, values of the program counter and module number in the source abstract dSCA for values in the target abstract dSCA are provided by the inverse mapping function, Ξ^{-1}.

Informally, the set of State Transition equations is created as follows:

- For each module m_i where $i \in \mathbb{N}_{k_2}$ and $i > 0$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$ in abstract dSCA extract and rewire the relevant functional specifications from the source abstract dSCA, if one exists, otherwise use the undefined constant u.

- Create a new State Transition equation from the previous result.

- For m_0, the program counter:

  - Create the program counter State Transition equation:

$$V_{pc}(t+1,a,x) = \begin{cases} mod(add(V_{pc}(t,a,x),1),Max_N) & \text{if } V_{pc}(t-1,a,x) = 0 \\ \vdots & \\ mod(add(V_{pc}(t,a,x),1),Max_N) & \vdots \end{cases}$$

The pattern for transformation is a familiar one of recursion over the structure of the target abstract dSCA. There are a couple of key functions that need to be explained in some more detail.

Consider that the VFOPDef term of a Value Function equation for an abstract dSCA is of the form:

$$f_i(pc,\ldots) = \begin{cases} f_{i,0}(\ldots) & \text{if } pc = 0 \\ f_{i,1}(\ldots) & \text{if } pc = 1 \\ \vdots & \\ f_{i,Max_N-1}(\ldots) & \text{if } pc = Max_N - 1 \end{cases}$$

It has already been noted that this is a convenient syntactic way of writing the conditional. If written according to the machine algebra, M_A, it would appear as:

$$f_i(pc,\ldots) = cond\left(\begin{array}{l} pc = 0, \\ f_{i,0}(\ldots), \\ cond\left(\begin{array}{l} pc = 1, \\ f_{i,1}(\ldots), \\ cond\left(\ldots, cond\left(\begin{array}{l} pc = Max_N - 1, \\ f_{i,Max_N-1}(\ldots), \\ null \end{array}\right)\ldots\right) \end{array}\right) \end{array}\right)$$

It is this second form that is used to select the component specification based on a particular value of the program counter. To do so, the operation $GetFn$ is introduced:

$$GetFn : VFOpDefTerm \times N \rightarrow Term$$

and is defined recursively over the structure of the VFOpDef term definition:

$$GetFn(cond(a,b,c),0) = b$$
$$GetFn(cond(a,b,c),pc_{req}) = GetFn(c,pc_{req}-1)$$

To generate a target abstract dSCA State Transition equation for a module a list of the appropriate VFOpDef Terms, selected from the source abstract dSCA by means of the inverse mapping function Ξ^{-}, the *GetEl* operation for STEqList specifications and the *GetFn* operation defined above are used. Consider module m_i in the target abstract dSCA, at program counter value *pc_val* it is defined to be executing either the:

1. VFOpDef term in module $fst(\Xi^{-1}(i,pc\_val))$ at the source program counter value $snd(\Xi^{-1}(i,pc\_val))$ in the source abstract dSCA, if the mapping is defined; or

2. the output u, if the mapping is undefined.

The NST operation is introduced to determine which case is under consideration, and it is given as:

$$NST : N^2 \times dSCASTVEqList^2 \times MapEqList \rightarrow VFOpDefList$$

and recurses over the program counter values to produce a list of VFOpDef terms that are used for the definition of the State Transition equation for a particular module. It is defined:

$$NST\begin{pmatrix} pc\_val, \\ mod\_val, \\ neqs, \\ oeqs, \\ \Xi^{-1} \end{pmatrix} = NST\begin{pmatrix} pc\_val-1, \\ mod\_val, \\ \begin{pmatrix} neqs, Extract\begin{pmatrix} oeqs, \\ mod\_val_{src}, \\ pc\_val_{src} \end{pmatrix} \end{pmatrix}, \\ oeqs, \\ \Xi^{-1} \end{pmatrix}$$

where:

$$mod\_val_{src} = fst\left(RetTerm\left(GetEl\left(\begin{array}{l}\Xi^{-1},\\ mod\_val,\\ pc\_val\end{array}\right),2\right)\right)$$

and:

$$pc\_val_{src} = snd\left(RetTerm\left(GetEl\left(\begin{array}{l}\Xi^{-1},\\ mod\_val,\\ pc\_val\end{array}\right),2\right)\right)$$

The *Extract* function used in the above definition is given as:

$$Extract : dSCASTVEqList \times N^2 \rightarrow VFOpDefTerm$$

and is defined as:

$$Extract\left(\begin{array}{l}oeqs,\\ mod\_val,\\ pc\_val\end{array}\right) = GetFn\left(\begin{array}{l}GetEl(oeqs, mod\_val),\\ pc\_val\end{array}\right)$$

The second case of the NST operation is defined as returning the list of VFOpDef terms constructed by appending the value for the program counter at 0 to those VFOpDef terms already obtained.

To complete the generation of a State Transition equation for module m_{mod_val} in the target dSCA the list of rewired VFOpDef terms must be turned into the component specification. This is done using the $NewST$ operation, given as:

$$NewST : VFOpDefList \times N \rightarrow VFOpDef$$

which takes the list of VFOpDef terms (which has the VFOpDef term corresponding to $pc = Max_N - 1$ at the head and the VFOpDef term corresponding to $pc = 0$ at the end) and recurses down the list producing the appropriate target dSCA VFOpDef term. For the recursive case it is defined as:

$$NewST\left(\begin{array}{l}(e, es),\\ pc\_val,\\ neqs\end{array}\right) = NewST\left(\begin{array}{l}es,\\ pc\_val - 1,\\ cond(V_{pc}(t, a, x) = pc\_val, e, neqs)\end{array}\right)$$

and the base case is defined:

$$NewST\begin{pmatrix} e, \\ pc\_val, \\ neqs \end{pmatrix} = cond(V_{pc}(t,a,x) = pc\_val, e, neqs)$$

11.1.7 Transformation Process

Each of the operations above need to be coordinated together so that a new abstract dSCA can be created by transforming the source abstract dSCA. The $Create\_adSCA$ operation is provided to do this, it is given as:

$$Transform : adSCAAlgebra \times N^2 \times MapEqList^2 \rightarrow adSCAAlgebra$$

The operation takes the source abstract dSCA and the defining shape of the target abstract dSCA together with the mapping and inverse mapping functions. It is defined for

$$VFOp = \begin{pmatrix} V_0 : T \times M_A^n \times M_A^k \rightarrow M_A, \\ \vdots, \\ V_k : T \times M_A^n \times M_A^k \rightarrow M_A \end{pmatrix}$$

$$\delta Op = \begin{pmatrix} \delta_{0,0,0} : T \times M_A^n \times M_A^k \rightarrow T, \\ \vdots, \\ \delta_{i,j,0} : T \times M_A^n \times M_A^k \rightarrow T \end{pmatrix}$$

and:

$$j = Get\_MaxA(Src\_SCA)$$
$$n = num\_inp(Src\_SCA)$$

as:

$$Transform\left(\begin{array}{l} SCA_{src}, \\ k, \\ Max_N, \\ \Xi, \\ \Xi^{-1} \end{array}\right) = CreateadSCA\left(\begin{array}{l} GetName(SCA_{src}), \\ adSCAAlgebra, \\ [], \\ [], \\ VFOp, \\ \gamma_0 : N^2 \rightarrow \{M, S, U\}, \\ \beta_0 : N^2 \rightarrow N, \\ \delta Op, \\ Create\gamma s\left(\begin{array}{l} SCA_{src}, \\ k, \\ Max_N, \\ \Xi^{-1} \end{array}\right), \\ Create\beta s\left(\begin{array}{l} SCA_{src}, \\ k, \\ Max_N, \\ \Xi^{-1} \end{array}\right), \\ Create\delta s\left(\begin{array}{l} SCA_{src}, \\ k, \\ Max_N, \\ \Xi, \\ \Xi^{-1}, \end{array}\right), \\ CreateIVs\left(\begin{array}{l} SCA_{src}, \\ k, \\ Max_N, \\ \Xi^{-1} \end{array}\right), \\ CreateSTs\left(\begin{array}{l} SCA_{src}, \\ k, \\ Max_N, \\ \Xi^{-1}, \\ \Xi \end{array}\right) \end{array}\right)$$

It is not intended to bring together all the operations defined in this chapter into a written down specification in this thesis for reasons of brevity. If this was to be performed, then it would appear similar to the algebraic specification provided for the SCA to abstract dSCA transformation in Appendix F.

11.2 Correctness

Theorem 11.2.1. *The transformation of a Form 1 abstract dSCA to a Form 2 abstract dSCA preserves correctness.*

The Form 1 abstract dSCA and transformed result, the Form 2 abstract dSCA,

exist in a hierarchy and it is possible to show that the transformation is correct by considering Poole, Holden and Tucker's work on hierarchy of Spatially Expanded Systems.

Let N_{dSCA1} be a $\mathbb{N}_k^{dSCA1} > 1$ module source Form1 abstract dSCA network with $n^{dSCA1} > 0$ sources processing data from a set M_A^{dSCA1} against a global clock T^{dSCA1}

Let N_{dSCA2} be a $\mathbb{N}_k^{dSCA2} > 1$ module Form 2 abstract dSCA network with $n^{dSCA2} > 0$ sources processing data from a set M_A^{dSCA2} against a global clock T^{dSCA2} as generated from N_{dSCA1} using the abstract dSCA to abstract dSCA transformation.

Poole, Holden and Tucker claimed that if it was possible to generate appropriate mappings and show the following diagram commutes then the two spatially expanded systems under consideration were correct with respect to each other.

$$\begin{array}{ccccc}
T_{dSCA1} & \times[T_{dSCA1} \rightarrow M_{A_{dSCA1}}]^{In_{dSCA1}} \times & M_{A_{dSCA1}}^{Ch_{dSCA1}} & \xrightarrow{V_{dSCA1}} & M_{A_{dSCA1}}^{Ch_{dSCA1}} \\
\uparrow \lambda & \uparrow \theta & \uparrow \phi & & \uparrow \phi \\
Start_{\lambda} & \times[T_{dSCA2} \rightarrow M_{A_{dSCA2}}]^{In_{dSCA2}} \times & M_{A_{dSCA2}}^{Ch_{dSCA2}} & \xrightarrow{V_{dSCA2}} & M_{A_{dSCA2}}^{Ch_{dSCA2}}
\end{array}$$

Mappings are needed for four areas:

- spaces;
- clocks;
- global states; and
- input streams.

The mappings are defined as follows:

Spaces. Spaces (modules) in the two networks differ, this is the point of the transformation, however, for modules m_i where $i \in \mathbb{N}_k^{dSCA2}$ the inverse mapping function

provides the necessary details. Thus it is appropriate to define the respacing operation $\pi : I_{N_{dSCA2}} \times \mathbb{N}_{Max_{N_{dSCA2}}} \rightarrow I_{N_{dSCA1}} \times \mathbb{N}_{Max_{N_{dSCA1}}}$ as:

$$\pi(i, pc) = \Xi^{-1}(i, pc)$$

Clocks. There exists a timing abstraction between the networks which is clearly given by the relationship between the values $Max_{N_{dSCA2}}$ and $Max_{N_{dSCA1}}$. The retiming between clocks T^{dSCA2} and T^{dSCA1} is the retiming $\lambda : T^{SCA} \rightarrow T^{dSCA}$, where for $t \in T^{dSCA2}$ it can be appropriately defined as:

$$\lambda(t) = \left\lfloor \frac{t}{\left(\frac{Max_{N_{dSCA2}}}{Max_{N_{dSCA1}}}\right)} \right\rfloor$$

Input Streams. There are no data abstractions required for inputs since these are not altered by the transformation. However there is a temporal abstraction, which matches the above retiming. Thus is it appropriate to define the input stream abstraction $\theta : [T^{dSCA2} \rightarrow M_{A_{dSCA2}}]^{n^{SCA2}} \rightarrow [T^{dSCA1} \rightarrow M_{A_{dSCA1}}]^{n^{dSCA1}}$ as the operation:

$$\begin{aligned} \theta(a)(t) &= a(\lambda(t)) \\ &= a(s) \end{aligned}$$

Global States. It is defined in the transformation that the carrier data set for source abstract dSCA and target abstract dSCA are the same, M_A. Thus there is no data abstraction required for consideration.

There is though an alterations of channels between the two modules based on the inverse mapping function identified. We therefore consider the state abstraction map $\phi : M_{A_{dSCA2}}^{Ch_{dSCA2}} \rightarrow M_{A_{dSCA1}}^{Ch_{dSCA1}}$ for all states $s \in M_{A_{dSCA2}}^{Ch_{dSCA2}}$ to be defined as follows, for $i \in \mathbb{N}_k^{dSCA2}$:

$$\phi(s)(i) = s(i)$$

Conjecture Given this set of mappings it is believed that the diagram above commutes, and proof of such is done in a similar manner as for Theorem 10.2.1.

11.3 Generalised Railroad Crossing Problem as a single processor Abstract dSCA

Now this thesis will consider the transformation of the (source) Form 1 abstract dSCA from the previous chapter, which has a defining shape of $\nabla = (k, 1)$, to the (target) Form 2 abstract dSCA with a defining shape of $\nabla = (1, k)$. The following example is based on the full definition of transformation given in Appendix G, as highlighted in the previous section.

Before walking through the processes of transformation, the prerequisites are reviewed:

- the source abstract dSCA has $36 + 1$ modules;
- each module in the source abstract dSCA has $Max_N^{src} = 1$ component specifications;
- the target abstract dSCA has $1 + 1$ modules;
- each module in the target abstract dSCA has $Max_N^{tgt} = 36$ component specifications;
- the defining size of the source abstract dSCA is $\triangle_{src} = 36$; and
- the defining size of the target abstract dSCA is $\triangle_{src} = 36$, thus $\triangle_{tgt} \geq \triangle_{src}$.

Thus, with the exception of a mapping between the source and target abstract dSCAs, the prerequisites are met. There are many possibilities for producing a mapping function, and of interest is the development of an automatic method for producing a mapping which results in a cyclic consistent abstract dSCA.

11.3.1 Automating the Generation of the Mapping Function

Recall the definition of a cycle consistent abstract dSCA:

> "If for all values of the program counter it can be shown that functions that calculate inputs to other modules execute at program counter values greater than the value of the program counter when the module that uses those input values executes, then the abstract dSCA is said to be cycle consistent"

The generation of a mapping between a source abstract dSCA with a defining shape of $\nabla = (k, 1)$ to a target abstract dSCA with a defining shape of $\nabla = (1, k)$ can be automated, if the following conditions are true:

- V_{out} contains only one module;
- $Max_N^{src} = 1$;
- There are no loops in the network; and
- The modules are (re-)numbered in a breadth first manner from V_{out}.

Figure 11.1 shows an abstract dSCA network that meets such conditions.

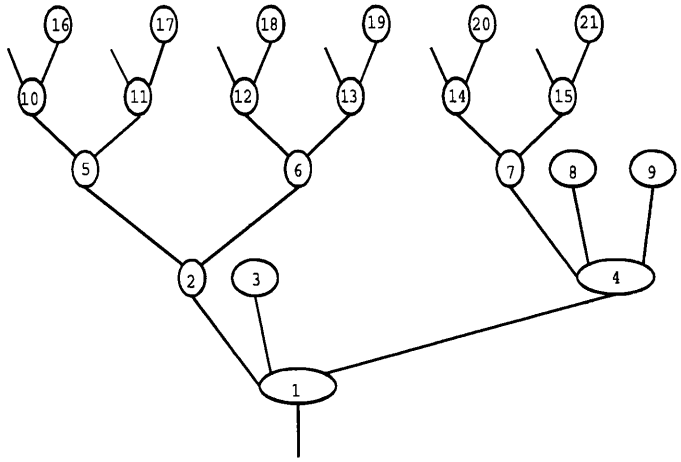

Figure 11.1: Numbered abstract dSCA network

The generation of the mapping for such a network is the simple process of walking the network in a breadth first manner. It can be seen by inspecting Figure 11.1 that an

algorithm of this type ensures that no module executes after the modules generating its inputs have executed - thus the resultant abstract dSCA is cycle consistent.

Using such an algorithm on the source abstract dSCA implementation of the GRCP, generates the following mapping function definition:

$\Xi(1,0) = (1,0)$	$\Xi(10,0) = (1,9)$	$\Xi(19,0) = (1,18)$	$\Xi(28,0) = (1,27)$
$\Xi(2,0) = (1,1)$	$\Xi(11,0) = (1,10)$	$\Xi(20,0) = (1,19)$	$\Xi(29,0) = (1,28)$
$\Xi(3,0) = (1,2)$	$\Xi(12,0) = (1,11)$	$\Xi(21,0) = (1,20)$	$\Xi(30,0) = (1,29)$
$\Xi(4,0) = (1,3)$	$\Xi(13,0) = (1,12)$	$\Xi(22,0) = (1,21)$	$\Xi(31,0) = (1,30)$
$\Xi(5,0) = (1,4)$	$\Xi(14,0) = (1,13)$	$\Xi(23,0) = (1,22)$	$\Xi(32,0) = (1,31)$
$\Xi(6,0) = (1,5)$	$\Xi(15,0) = (1,14)$	$\Xi(24,0) = (1,23)$	$\Xi(33,0) = (1,32)$
$\Xi(7,0) = (1,6)$	$\Xi(16,0) = (1,15)$	$\Xi(25,0) = (1,24)$	$\Xi(34,0) = (1,33)$
$\Xi(8,0) = (1,7)$	$\Xi(17,0) = (1,16)$	$\Xi(26,0) = (1,25)$	$\Xi(35,0) = (1,34)$
$\Xi(9,0) = (1,8)$	$\Xi(18,0) = (1,17)$	$\Xi(27,0) = (1,26)$	$\Xi(36,0) = (1,35)$

The corresponding inverse mapping function, Ξ^{-1}, is subsequently defined as:

$\Xi^{-1}(1,0) = (1,0)$	$\Xi^{-1}(1,9) = (10,0)$	$\Xi^{-1}(1,18) = (19,0)$	$\Xi^{-1}(1,27) = (28,0)$
$\Xi^{-1}(1,1) = (2,0)$	$\Xi^{-1}(1,10) = (11,0)$	$\Xi^{-1}(1,19) = (20,0)$	$\Xi^{-1}(1,28) = (29,0)$
$\Xi^{-1}(1,2) = (3,0)$	$\Xi^{-1}(1,11) = (12,0)$	$\Xi^{-1}(1,20) = (21,0)$	$\Xi^{-1}(1,29) = (30,0)$
$\Xi^{-1}(1,3) = (4,0)$	$\Xi^{-1}(1,12) = (13,0)$	$\Xi^{-1}(1,21) = (22,0)$	$\Xi^{-1}(1,30) = (31,0)$
$\Xi^{-1}(1,4) = (5,0)$	$\Xi^{-1}(1,13) = (14,0)$	$\Xi^{-1}(1,22) = (23,0)$	$\Xi^{-1}(1,31) = (32,0)$
$\Xi^{-1}(1,5) = (6,0)$	$\Xi^{-1}(1,14) = (15,0)$	$\Xi^{-1}(1,23) = (24,0)$	$\Xi^{-1}(1,32) = (33,0)$
$\Xi^{-1}(1,6) = (7,0)$	$\Xi^{-1}(1,15) = (16,0)$	$\Xi^{-1}(1,24) = (25,0)$	$\Xi^{-1}(1,33) = (34,0)$
$\Xi^{-1}(1,7) = (8,0)$	$\Xi^{-1}(1,16) = (17,0)$	$\Xi^{-1}(1,25) = (26,0)$	$\Xi^{-1}(1,34) = (35,0)$
$\Xi^{-1}(1,8) = (9,0)$	$\Xi^{-1}(1,17) = (18,0)$	$\Xi^{-1}(1,26) = (27,0)$	$\Xi^{-1}(1,35) = (36,0)$

It is possible that there are other methods for producing cycle consistent dSCAs, this is the only one that we have considered.

γ-wiring function

The target network has $1+1$ modules and 36 component specifications for each module. Each module except the program counter will have a maximum of $3+1$ arguments. The *Createγs* operation will therefore be called as follows:

$$Create\gamma s \left(\begin{array}{l} Source\_SCA, \\ 1, \\ 36, \\ 4, \\ \Xi^{-1} \end{array} \right)$$

which would expand, according to its definition, to the call to $B\gamma s$ of:

$$Create\gamma s \begin{pmatrix} Source\_SCA, \\ 1, \\ 36, \\ 4, \\ \Xi^{-1} \end{pmatrix} = B\gamma s \begin{pmatrix} 1, \\ Get\gamma s(Source\_SCA), \\ [], \\ 36, \\ 4, \\ \Xi^{-1} \end{pmatrix}$$

The right hand side of this definition will result in a recursive call to $B\gamma s$ as well as a call to the $B\gamma pc$ operation. Considering the call to $B\gamma pc$ first, it can be seen below that the value of Max_N is decremented by one in preparation for the recursive nature of $B\gamma pc$ and are supplying the empty list for the new values of γ-wiring functions to be added to:

$$B\gamma s \begin{pmatrix} 1, \\ old\gamma s, \\ [], \\ 36, \\ 4, \\ \Xi^{-1} \end{pmatrix} = B\gamma s \begin{pmatrix} 0, \\ old\gamma s, \\ \begin{pmatrix} B\gamma pc \begin{pmatrix} 35, \\ old\gamma s, \\ [], \\ 1, \\ 4, \\ \Xi^{-1} \end{pmatrix}, [] \end{pmatrix}, \\ 36, \\ 4, \\ \Xi^{-1} \end{pmatrix}$$

The $B\gamma pc$ call expands as:

$$B\gamma pc\begin{pmatrix} 35, \\ old\gamma s, \\ [], \\ 1, \\ 4, \\ \Xi^{-1} \end{pmatrix} = B\gamma pc\begin{pmatrix} 34, \\ old\gamma s, \\ \begin{pmatrix} B\gamma arg\begin{pmatrix} 3, \\ old\gamma s, \\ [], \\ 1, \\ 35, \\ \Xi^{-1} \end{pmatrix}, [] \end{pmatrix}, \\ 1, \\ 4, \\ \Xi^{-1} \end{pmatrix}$$

Taking a look at the call to the $B\gamma arg$ operation, it can be seen to recurse over all the arguments, $arg\_val \in \{0, 1, 2, 3\}$, and it can be ascertained that:

$$\gamma^{src}_{snd(\Xi^{-1}(1,35))}(fst(\Xi^{-1}(1, 35)), arg\_val) \uparrow$$

or written to remove the inverse mapping function:

$$\gamma^{src}_{0}(36, arg\_val) \uparrow$$

these values are to be expected, since module 36 in the source abstract dSCA simply provides a constant and has no inputs. The list of γ-wiring functions returned for the 4 arguments of module 1 at $pc\_val = 35$ are:

$$\gamma_{35}(1, 3) = U,$$
$$\gamma_{35}(1, 2) = U,$$
$$\gamma_{35}(1, 1) = U,$$
$$\gamma_{35}(1, 0) = M$$

The recursive call to $Build\gamma pc$ will expand to:

$$B\gamma pc\left(\begin{array}{l} 34, \\ old\gamma s, \\ \left(\begin{array}{l} \gamma_{35}(1,3)=U, \\ \gamma_{35}(1,2)=U, \\ \gamma_{35}(1,1)=U, \\ \gamma_{35}(1,0)=M \end{array}\right), \\ 1, \\ 4, \\ \Xi^{-1} \end{array}\right) = B\gamma pc\left(\begin{array}{l} 33, \\ old\gamma s, \\ \left(\begin{array}{l} B\gamma arg\left(\begin{array}{l} 3, \\ old\gamma s, \\ [], \\ 1, \\ 34, \\ \Xi^{-1} \end{array}\right), \\ \left(\begin{array}{l} \gamma_{35}(1,3)=U, \\ \gamma_{35}(1,2)=U, \\ \gamma_{35}(1,1)=U, \\ \gamma_{35}(1,0)=M \end{array}\right) \end{array}\right), \\ 1, \\ 4, \\ \Xi^{-1} \end{array}\right)$$

The call to $B\gamma arg$ in this case is more productive since $\Xi^1(1,34) = (35,0)$, and module 35 in the source abstract dSCA is wired to two inputs. The expansion of the $B\gamma arg$ call is:

$$B\gamma arg\left(\begin{array}{l} 3, \\ old\gamma s, \\ new\gamma s, \\ 1, \\ 34, \\ \Xi^{-1} \end{array}\right) = B\gamma arg\left(\begin{array}{l} 2, \\ old\gamma s, \\ \left(\begin{array}{l} B\gamma\left(\begin{array}{l} 1, \\ 3, \\ 34, \\ old\gamma s, \\ \Xi^{-1} \end{array}\right), neqs \end{array}\right), \\ 1, \\ 34, \\ \Xi^{-1} \end{array}\right)$$

The first call to $B\gamma$ will instigate the case where there is no corresponding γ-wiring function in the source abstract dSCA and so will produce a wiring to the undefined module:

$$B\gamma\begin{pmatrix}1,\\3,\\34,\\old\gamma s,\\\Xi^{-1}\end{pmatrix} = Build\gamma\begin{pmatrix}\gamma_{34},\\1,\\3,\\U\end{pmatrix}$$
$$= \gamma_{34}(1,3) = U$$

The recursive call to $B\gamma arg$ will be:

$$B\gamma arg\begin{pmatrix}2,\\old\gamma s,\\new\gamma s,\\1,\\34,\\\Xi^{-1}\end{pmatrix} = B\gamma arg\begin{pmatrix}1,\\old\gamma s,\\\begin{pmatrix}B\gamma\begin{pmatrix}1,\\2,\\34,\\old\gamma s,\\\Xi^{-1}\end{pmatrix},neqs\end{pmatrix},\\1,\\34,\\\Xi^{-1}\end{pmatrix}$$

This time the call to $B\gamma$ will be the case where the mapping exists, and there is a corresponding γ-wiring function in the source abstract dSCA. In this situation $B\gamma$

will expand as:

$$B\gamma \begin{pmatrix} 1, \\ 2, \\ 34, \\ old\gamma s, \\ \Xi^{-1} \end{pmatrix} = Build\gamma \begin{pmatrix} \gamma_{34}, \\ 1, \\ 2, \\ RetTerm \left(GetEl \begin{pmatrix} old\gamma s, \\ i_{old}, \\ 2, \\ pc_{old} \end{pmatrix}, 2 \right) \end{pmatrix}$$

where:

$$\begin{aligned} i_{old} &= fst(RetTerm(GetEl(\Xi^{-1}, 1, 34)), 2) \\ &= fst(RetTerm(\Xi^{-1}(1, 34) = (35, 0)), 2) \\ &= fst(35, 0) \\ &= 35 \end{aligned}$$

and:

$$\begin{aligned} pc_{old} &= sndfst(RetTerm(GetEl(\Xi^{-1}, 1, 34)), 2) \\ &= snd(RetTerm(\Xi^{-1}(1, 34) = (35, 0)), 2) \\ &= snd(35, 0) \\ &= 0 \end{aligned}$$

the call to $B\gamma$ can be rewritten as:

$$
\begin{aligned}
B\gamma\begin{pmatrix} 1, \\ 2, \\ 34, \\ old\gamma s, \\ \Xi^{-1} \end{pmatrix} &= Build\gamma\begin{pmatrix} \gamma_{34}, \\ 1, \\ 2, \\ RetTerm\left(GetEl\begin{pmatrix} old\gamma s, \\ 35, \\ 2, \\ 0 \end{pmatrix}, 2\right) \end{pmatrix} \\
&= Build\gamma\begin{pmatrix} \gamma_{34}, \\ 1, \\ 2, \\ RetTerm\left(\gamma_0(35,2) = S\right), 2) \end{pmatrix} \\
&= Build\gamma\begin{pmatrix} \gamma_{34}, \\ 1, \\ 2, \\ S \end{pmatrix} \\
&= (\gamma_{34}(1,2) = S)
\end{aligned}
$$

Similarly the remainder of the γ-wiring functions for module 1 at program counter value 34 are determined, providing the following list:

$$
\begin{aligned}
\gamma_{34}(1,3) &= U, \\
\gamma_{34}(1,2) &= S, \\
\gamma_{34}(1,1) &= S, \\
\gamma_{34}(1,0) &= M
\end{aligned}
$$

The process continues through the values of the program counters and modules, until the call to $B\gamma s$ where the module number is 0 is reached. This will result in:

$$B\gamma s \begin{pmatrix} 0, \\ old\gamma s, \\ new\gamma s, \\ 35, \\ 4, \\ \Xi^{-1} \end{pmatrix} = \left(Build\gamma \begin{pmatrix} \gamma_0, \\ 0, \\ 0, \\ M \end{pmatrix}, \ldots, Build\gamma \begin{pmatrix} \gamma_{35}, \\ 0, \\ 0, \\ M \end{pmatrix}, new\gamma s \right)$$

The complete list of γ-wiring functions generated from this process are listed below:

$$\begin{array}{llll}
\gamma_0(pc,0) = M, & \gamma_9(pc,0) = M, & \gamma_{18}(pc,0) = M, & \gamma_{27}(pc,0) = M, \\
\gamma_1(pc,0) = M, & \gamma_{10}(pc,0) = M, & \gamma_{19}(pc,0) = M, & \gamma_{28}(pc,0) = M, \\
\gamma_2(pc,0) = M, & \gamma_{11}(pc,0) = M, & \gamma_{20}(pc,0) = M, & \gamma_{29}(pc,0) = M, \\
\gamma_3(pc,0) = M, & \gamma_{12}(pc,0) = M, & \gamma_{21}(pc,0) = M, & \gamma_{30}(pc,0) = M, \\
\gamma_4(pc,0) = M, & \gamma_{13}(pc,0) = M, & \gamma_{22}(pc,0) = M, & \gamma_{31}(pc,0) = M, \\
\gamma_5(pc,0) = M, & \gamma_{14}(pc,0) = M, & \gamma_{23}(pc,0) = M, & \gamma_{32}(pc,0) = M, \\
\gamma_6(pc,0) = M, & \gamma_{15}(pc,0) = M, & \gamma_{24}(pc,0) = M, & \gamma_{33}(pc,0) = M, \\
\gamma_7(pc,0) = M, & \gamma_{16}(pc,0) = M, & \gamma_{25}(pc,0) = M, & \gamma_{34}(pc,0) = M, \\
\gamma_8(pc,0) = M, & \gamma_{17}(pc,0) = M, & \gamma_{26}(pc,0) = M, & \gamma_{35}(pc,0) = M, \\
\gamma_0(1,0) = M, & \gamma_0(1,1) = M, & \gamma_0(1,2) = M, & \gamma_0(1,3) = M, \\
\gamma_1(1,0) = M, & \gamma_1(1,1) = M, & \gamma_1(1,2) = M, & \gamma_1(1,3) = U, \\
\gamma_2(1,0) = M, & \gamma_2(1,1) = U, & \gamma_2(1,2) = U, & \gamma_2(1,3) = U, \\
\gamma_3(1,0) = M, & \gamma_3(1,1) = M, & \gamma_3(1,2) = M, & \gamma_3(1,3) = M, \\
\gamma_4(1,0) = M, & \gamma_4(1,1) = M, & \gamma_4(1,2) = M, & \gamma_4(1,3) = U, \\
\gamma_5(1,0) = M, & \gamma_5(1,1) = M, & \gamma_5(1,2) = M, & \gamma_5(1,3) = U, \\
\gamma_6(1,0) = M, & \gamma_6(1,1) = M, & \gamma_6(1,2) = M, & \gamma_6(1,3) = U, \\
\gamma_7(1,0) = M, & \gamma_7(1,1) = U, & \gamma_7(1,2) = U, & \gamma_7(1,3) = U, \\
\gamma_8(1,0) = M, & \gamma_8(1,1) = U, & \gamma_8(1,2) = U, & \gamma_8(1,3) = U, \\
\gamma_9(1,0) = M, & \gamma_9(1,1) = M, & \gamma_9(10,2) = M, & \gamma_9(1,3) = U, \\
\gamma_{10}(1,0) = M, & \gamma_{10}(1,1) = S, & \gamma_{10}(1,2) = M, & \gamma_{10}(1,3) = U, \\
\gamma_{11}(1,0) = M, & \gamma_{11}(1,1) = M & \gamma_{11}(1,2) = M, & \gamma_{11}(1,3) = U, \\
\gamma_{12}(1,0) = M, & \gamma_{12}(1,1) = S, & \gamma_{12}(1,2) = M, & \gamma_{12}(1,3) = U, \\
\gamma_{13}(1,0) = M, & \gamma_{13}(1,1) = M, & \gamma_{13}(1,2) = M, & \gamma_{13}(1,3) = U,
\end{array}$$

$$\begin{array}{llll}
\gamma_{14}(1,0)=M, & \gamma_{14}(1,1)=S, & \gamma_{14}(1,2)=M, & \gamma_{14}(1,3)=U,\\
\gamma_{15}(1,0)=M, & \gamma_{15}(1,1)=U, & \gamma_{15}(1,2)=U, & \gamma_{15}(1,3)=U,\\
\gamma_{16}(1,0)=M, & \gamma_{16}(1,1)=U, & \gamma_{16}(1,2)=U, & \gamma_{16}(1,3)=U,\\
\gamma_{17}(1,0)=M, & \gamma_{17}(1,1)=U, & \gamma_{17}(1,2)=U, & \gamma_{17}(1,3)=U,\\
\gamma_{18}(1,0)=M, & \gamma_{18}(1,1)=U, & \gamma_{18}(1,2)=U, & \gamma_{18}(1,3)=U,\\
\gamma_{19}(1,0)=M, & \gamma_{19}(1,1)=U, & \gamma_{19}(1,2)=U, & \gamma_{19}(1,3)=U,\\
\gamma_{20}(1,0)=M, & \gamma_{20}(1,1)=U, & \gamma_{20}(1,2)=U, & \gamma_{20}(1,3)=U,\\
\gamma_{21}(1,0)=M, & \gamma_{21}(1,1)=M, & \gamma_{21}(1,2)=M, & \gamma_{21}(1,3)=U,\\
\gamma_{22}(1,0)=M, & \gamma_{22}(1,1)=M, & \gamma_{22}(1,2)=M, & \gamma_{22}(1,3)=U,\\
\gamma_{23}(1,0)=M, & \gamma_{23}(1,1)=M, & \gamma_{23}(1,2)=M, & \gamma_{23}(1,3)=U,\\
\gamma_{24}(1,0)=M, & \gamma_{24}(1,1)=M, & \gamma_{24}(1,2)=M, & \gamma_{24}(1,3)=U,\\
\gamma_{25}(1,0)=M, & \gamma_{25}(1,1)=M, & \gamma_{25}(1,2)=M, & \gamma_{25}(1,3)=U,\\
\gamma_{26}(1,0)=M, & \gamma_{26}(1,1)=M, & \gamma_{26}(1,2)=M, & \gamma_{26}(1,3)=U,\\
\gamma_{27}(1,0)=M, & \gamma_{27}(1,1)=M, & \gamma_{27}(1,2)=M, & \gamma_{27}(1,3)=U,\\
\gamma_{28}(1,0)=M, & \gamma_{28}(1,1)=S, & \gamma_{28}(1,2)=S, & \gamma_{28}(1,3)=U.\\
\gamma_{29}(1,0)=M, & \gamma_{29}(1,1)=U, & \gamma_{29}(1,2)=U, & \gamma_{29}(1,3)=U,\\
\gamma_{30}(1,0)=M, & \gamma_{30}(1,1)=S, & \gamma_{30}(1,2)=S, & \gamma_{30}(1,3)=U,\\
\gamma_{31}(1,0)=M, & \gamma_{31}(1,1)=U, & \gamma_{31}(1,2)=U, & \gamma_{31}(1,3)=U,\\
\gamma_{32}(1,0)=M, & \gamma_{32}(1,1)=S, & \gamma_{32}(1,2)=S, & \gamma_{32}(1,3)=U,\\
\gamma_{33}(1,0)=M, & \gamma_{33}(1,1)=U, & \gamma_{33}(1,2)=U, & \gamma_{33}(1,3)=U,\\
\gamma_{34}(1,0)=M, & \gamma_{34}(1,1)=S, & \gamma_{34}(1,2)=S, & \gamma_{34}(1,3)=U,\\
\gamma_{35}(1,0)=M, & \gamma_{35}(1,1)=U, & \gamma_{35}(1,2)=U, & \gamma_{35}(1,3)=U
\end{array}$$

β-wiring functions

The transformation of β-wiring functions results in the following:

$$\begin{array}{llll}
\beta_0(pc,0)=pc, & \beta_9(pc,0)=pc, & \beta_{18}(pc,0)=pc, & \beta_{27}(pc,0)=pc,\\
\beta_1(pc,0)=pc, & \beta_{10}(pc,0)=pc, & \beta_{19}(pc,0)=pc, & \beta_{28}(pc,0)=pc,\\
\beta_2(pc,0)=pc, & \beta_{11}(pc,0)=pc, & \beta_{20}(pc,0)=pc, & \beta_{29}(pc,0)=pc,\\
\beta_3(pc,0)=pc, & \beta_{12}(pc,0)=pc, & \beta_{21}(pc,0)=pc, & \beta_{30}(pc,0)=pc,\\
\beta_4(pc,0)=pc, & \beta_{13}(pc,0)=pc, & \beta_{22}(pc,0)=pc, & \beta_{31}(pc,0)=pc,\\
\beta_5(pc,0)=pc, & \beta_{14}(pc,0)=pc, & \beta_{23}(pc,0)=pc, & \beta_{32}(pc,0)=pc,\\
\beta_6(pc,0)=pc, & \beta_{15}(pc,0)=pc, & \beta_{24}(pc,0)=pc, & \beta_{33}(pc,0)=pc,\\
\beta_7(pc,0)=pc, & \beta_{16}(pc,0)=pc, & \beta_{25}(pc,0)=pc, & \beta_{34}(pc,0)=pc,\\
\beta_8(pc,0)=pc, & \beta_{17}(pc,0)=pc, & \beta_{26}(pc,0)=pc, & \beta_{35}(pc,0)=pc,\\
\beta_0(1,0)=pc, & \beta_0(1,1)=1, & \beta_0(1,2)=1, & \beta_0(1,3)=1,\\
\beta_1(1,0)=pc, & \beta_1(1,1)=1, & \beta_1(1,2)=1, & \beta_1(1,3)=\omega,\\
\beta_2(1,0)=pc, & \beta_2(1,1)=\omega, & \beta_2(1,2)=\omega, & \beta_2(1,3)=\omega,\\
\beta_3(1,0)=pc, & \beta_3(1,1)=1, & \beta_3(1,2)=1, & \beta_3(1,3)=1,\\
\beta_4(1,0)=pc, & \beta_4(1,1)=1, & \beta_4(1,2)=1, & \beta_4(1,3)=\omega,\\
\beta_5(1,0)=pc, & \beta_5(1,1)=1, & \beta_5(1,2)=1, & \beta_5(1,3)=\omega,\\
\beta_6(1,0)=pc, & \beta_6(1,1)=1, & \beta_6(1,2)=1, & \beta_6(1,3)=\omega,\\
\beta_7(1,0)=pc, & \beta_7(1,1)=\omega, & \beta_7(1,2)=\omega, & \beta_7(1,3)=\omega,\\
\beta_8(1,0)=pc, & \beta_8(1,1)=\omega, & \beta_8(1,2)=\omega, & \beta_8(1,3)=\omega,
\end{array}$$

$$
\begin{array}{llll}
\beta_9(1,0)=pc, & \beta_9(1,1)=1, & \beta_9(1,2)=1, & \beta_9(1,3)=\omega, \\
\beta_{10}(1,0)=pc, & \beta_{10}(1,1)=9, & \beta_{10}(1,2)=1, & \beta_{10}(1,3)=\omega, \\
\beta_{11}(1,0)=pc, & \beta_{11}(1,1)=1, & \beta_{11}(1,2)=1, & \beta_{11}(1,3)=\omega, \\
\beta_{12}(1,0)=pc, & \beta_{12}(1,1)=9, & \beta_{12}(1,2)=1, & \beta_{12}(1,3)=\omega, \\
\beta_{13}(1,0)=pc, & \beta_{13}(1,1)=1, & \beta_{13}(1,2)=1, & \beta_{13}(1,3)=\omega, \\
\beta_{14}(1,0)=pc, & \beta_{14}(1,1)=9, & \beta_{14}(1,2)=1, & \beta_{14}(1,3)=\omega, \\
\beta_{15}(1,0)=pc, & \beta_{15}(1,1)=\omega, & \beta_{15}(1,2)=\omega, & \beta_{15}(1,3)=\omega, \\
\beta_{16}(1,0)=pc, & \beta_{16}(1,1)=\omega, & \beta_{16}(1,2)=\omega, & \beta_{16}(1,3)=\omega, \\
\beta_{17}(1,0)=pc, & \beta_{17}(1,1)=\omega, & \beta_{17}(1,2)=\omega, & \beta_{17}(1,3)=\omega, \\
\beta_{18}(1,0)=pc, & \beta_{18}(1,1)=\omega, & \beta_{18}(1,2)=\omega, & \beta_{18}(1,3)=\omega, \\
\beta_{19}(1,0)=pc, & \beta_{19}(1,1)=\omega, & \beta_{19}(1,2)=\omega, & \beta_{19}(1,3)=\omega, \\
\beta_{20}(1,0)=pc, & \beta_{20}(1,1)=\omega, & \beta_{20}(1,2)=\omega, & \beta_{20}(1,3)=\omega, \\
\beta_{21}(1,0)=pc, & \beta_{21}(1,1)=1, & \beta_{21}(1,2)=1, & \beta_{21}(1,3)=\omega, \\
\beta_{22}(1,0)=pc, & \beta_{22}(1,1)=1, & \beta_{22}(1,2)=1, & \beta_{22}(1,3)=\omega, \\
\beta_{23}(1,0)=pc, & \beta_{23}(1,1)=1, & \beta_{23}(1,2)=1, & \beta_{23}(1,3)=\omega, \\
\beta_{24}(1,0)=pc, & \beta_{24}(1,1)=1, & \beta_{24}(1,2)=1, & \beta_{24}(1,3)=\omega, \\
\beta_{25}(1,0)=pc, & \beta_{25}(1,1)=1, & \beta_{25}(1,2)=1, & \beta_{25}(1,3)=\omega, \\
\beta_{26}(1,0)=pc, & \beta_{26}(1,1)=1, & \beta_{26}(1,2)=1, & \beta_{26}(1,3)=\omega, \\
\beta_{27}(1,0)=pc, & \beta_{27}(1,1)=1, & \beta_{27}(1,2)=1, & \beta_{27}(1,3)=\omega, \\
\beta_{28}(1,0)=pc, & \beta_{28}(1,1)=1, & \beta_{28}(1,2)=2, & \beta_{28}(1,3)=\omega. \\
\beta_{29}(1,0)=pc, & \beta_{29}(1,1)=\omega, & \beta_{29}(1,2)=\omega, & \beta_{29}(1,3)=\omega, \\
\beta_{30}(1,0)=pc, & \beta_{30}(1,1)=3, & \beta_{30}(1,2)=4, & \beta_{30}(1,3)=\omega, \\
\beta_{31}(1,0)=pc, & \beta_{31}(1,1)=\omega, & \beta_{31}(1,2)=\omega, & \beta_{31}(1,3)=\omega, \\
\beta_{32}(1,0)=pc, & \beta_{32}(1,1)=5, & \beta_{32}(1,2)=6, & \beta_{32}(1,3)=\omega, \\
\beta_{33}(1,0)=pc, & \beta_{33}(1,1)=\omega, & \beta_{33}(1,2)=\omega, & \beta_{33}(1,3)=\omega, \\
\beta_{34}(1,0)=pc, & \beta_{34}(1,1)=7, & \beta_{34}(1,2)=8, & \beta_{34}(1,3)=\omega, \\
\beta_{35}(1,0)=pc, & \beta_{35}(1,1)=\omega, & \beta_{35}(1,2)=\omega, & \beta_{35}(1,3)=\omega, \\
\beta_0(pc,1)=pc &
\end{array}
$$

Delay Functions

The initial call to create the delay functions is to the $Create\delta$ operation:

$$
Create\delta s \left(\begin{array}{l} Source\_SCA, \\ 1, \\ 36, \\ 4, \\ \Xi, \\ \Xi^{-1}, \end{array} \right)
$$

this expands to:

$$Create\delta s\begin{pmatrix} Source\_SCA, \\ 1, \\ 36, \\ 4, \\ \Xi, \\ \Xi^{-1}, \end{pmatrix} = B\delta s\begin{pmatrix} 1, \\ Get\delta Eqs(Source\_SCA), \\ [], \\ 4, \\ Get\gamma Eqs(Source\_SCA), \\ Create\beta s\begin{pmatrix} Source\_SCA, \\ 1, \\ 36, \\ 4, \\ \Xi^{-1} \end{pmatrix}, \\ GetMax_N(Source\_SCA), \\ 36, \\ \Xi, \\ \Xi^{-1}, \end{pmatrix}$$

the β-wiring functions are given in the previous section, and the operations $Get\gamma Eqs$, $Get\delta Eqs$ and $GetMax_N$ are defined in the SCA specification. The first call to $B\delta s$ is the recursive case:

$$B\delta s\begin{pmatrix} 1, \\ old\delta s, \\ [], \\ 4, \\ old\gamma s, \\ new\beta s, \\ 1, \\ 36, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} = B\delta s\begin{pmatrix} 0, \\ old\delta s, \\ \begin{pmatrix} B\delta pc\begin{pmatrix} 35, \\ old\delta s, \\ [], \\ 1, \\ 4, \\ old\gamma s, \\ new\beta s, \\ 1, \\ 36, \\ \Xi^{-1}, \\ \Xi \end{pmatrix}, [] \end{pmatrix}, \\ 4, old\gamma s, new\beta s, \\ 1, 36, \Xi^{-1}, \Xi \end{pmatrix},$$

This involves a recursive call to itself, and a call to the $B\delta pc$ operation to produce the delay functions for the values of the program counter:

$$B\delta pc\left(\begin{array}{l}35,\\ old\delta s,\\ new\delta s,\\ 1,\\ 4,\\ old\gamma s,\\ new\beta s,\\ 1,\\ 36,\\ \Xi^{-1},\\ \Xi\end{array}\right) = B\delta pc\left(\begin{array}{l}34,\\ old\delta s,\\ \left(\begin{array}{l}\\ \\ B\delta arg\end{array}\left(\begin{array}{l}3,\\ old\delta s,\\ [],\\ 1,\\ 35,\\ old\gamma s,\\ new\beta s,\\ 1,\\ 36,\\ \Xi^{-1},\\ \Xi\end{array}\right),new\delta s\right),\\ 1,4,old\gamma s,new\beta s,\\ 1,36,\Xi^{-1},\Xi\end{array}\right),$$

Building the delay function for the arguments of module 1 at program counter value 35 would result in 4 unit delay functions since module 36 in the source abstract dSCA is not wired to anything.

The next recursive call is:

$$B\delta pc\left(\begin{array}{l}34,\\ old\delta s,\\ new\delta s,\\ 1,\\ 4,\\ old\gamma s,\\ new\beta s,\\ 1,\\ 36,\\ \Xi^{-1},\\ \Xi\end{array}\right) = B\delta pc\left(\begin{array}{l}33,\\ old\delta s,\\ \left(\begin{array}{l}\\ \\ B\delta arg\end{array}\left(\begin{array}{l}3,\\ old\delta s,\\ [],\\ 1,\\ 34,\\ old\gamma s,\\ new\beta s,\\ 1,\\ 36,\\ \Xi^{-1},\\ \Xi\end{array}\right),new\delta s\right),\\ 1,4,old\gamma s,new\beta s,\\ 1,36,\Xi^{-1},\Xi\end{array}\right),$$

where the call to $B\delta arg$ considers a module which is wired to two sources:

$$B\delta arg\left(\begin{array}{l}3,\\ old\delta s,\\ new\delta s,\\ 1,\\ 34,\\ old\gamma s,\\ new\beta s,\\ 1,\\ 36,\\ \Xi^{-1},\\ \Xi\end{array}\right) = B\delta arg\left(\begin{array}{l}2,\\ old\delta s,\\ \left(\begin{array}{l}B\delta\left(\begin{array}{l}1,\\ 3,\\ 34,\\ old\delta s,\\ old\gamma s,\\ new\beta s,\\ 1,\\ 36,\\ \Xi^{-1},\\ \Xi,\end{array}\right), new\delta s\end{array}\right),\\ 1,\\ 34,\\ old\gamma s,\\ new\beta s,\\ 1,\\ 36,\\ \Xi^{-1},\\ \Xi,\end{array}\right)$$

The 4^{th} argument to module 1 at program counter value 34 is not wired and therefore the call above would produce the delay function:

$$\delta_{1,3,34}(t,a,x) = t-1$$

The recursive call to $B\delta arg$ is:

$$B\delta arg\left(\begin{array}{l}2,\\ old\delta s,\\ new\delta s,\\ 1,\\ 34,\\ old\gamma s,\\ new\beta s,\\ 1,\\ 36,\\ \Xi^{-1},\\ \Xi\end{array}\right) = B\delta arg\left(\begin{array}{l}1,\\ old\delta s,\\ \left(\begin{array}{l}B\delta\left(\begin{array}{l}1,\\ 2,\\ 34,\\ old\delta s,\\ old\gamma s,\\ new\beta s,\\ 1,\\ 36,\\ \Xi^{-1},\\ \Xi,\end{array}\right), new\delta s\end{array}\right),\\ 1,\\ 34,\\ old\gamma s,\\ new\beta s,\\ 1,\\ 36,\\ \Xi^{-1},\\ \Xi,\end{array}\right)$$

It is known that $\beta_{34}(1,2)$ is defined, and that the input is wired to a source, since:

$$\Xi^{-1}(1,34) = (35,0)$$

and that from the source abstract dSCA:

$$\beta_0(35,2) = S$$

hence, the delay function will be the unit delay. It is a similar situation for the 1^{st} argument to this module at this program counter value, and of course the 0^{th} argument is wired to the program counter and so will, by definition, be the unit delay. The process generates the following list of delay functions for module 1 at program counter value 34:

$$\delta_{1,3,34}(t,a,x) = t-1,$$
$$\delta_{1,2,34}(t,a,x) = t-1,$$
$$\delta_{1,1,34}(t,a,x) = t-1,$$
$$\delta_{1,0,34}(t,a,x) = t-1$$

If the following call for program counter value 23, which occurs in the recursive path of $B\delta pc$, is considered then it can be seen that it makes some calls to other modules within the source abstract dSCA:

$$B\delta pc\begin{pmatrix} 23, \\ old\delta s, \\ new\delta s, \\ 1, \\ 4, \\ old\gamma s, \\ new\beta s, \\ 1, \\ 36, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} = B\delta pc\begin{pmatrix} 22, \\ old\delta s, \\ \begin{pmatrix} B\delta arg\begin{pmatrix} 3, \\ old\delta s, \\ [], \\ 1, \\ 23, \\ old\gamma s, \\ new\beta s, \\ 1, \\ 36, \\ \Xi^{-1}, \\ \Xi \end{pmatrix}, new\delta s \end{pmatrix}, \\ 1, 4, old\gamma s, new\beta s, \\ 1, 36, \Xi^{-1}, \Xi \end{pmatrix},$$

This produces a unit delay for the 4^{th} argument, but when considering the call for the 3^{rd} argument, the following is found:

$$B\delta arg\begin{pmatrix} 2, \\ old\delta s, \\ new\delta s, \\ 1, \\ 23, \\ old\gamma s, \\ new\beta s, \\ 1, \\ 36, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} = B\delta arg\begin{pmatrix} 1, \\ old\delta s, \\ \begin{pmatrix} B\delta\begin{pmatrix} 1, \\ 2, \\ 23, \\ old\delta s, \\ old\gamma s, \\ new\beta s, \\ 1, \\ 36, \\ \Xi^{-1}, \\ \Xi, \end{pmatrix}, new\delta s \end{pmatrix}, \\ 1, 23, old\gamma s, new\beta s, \\ 1, 36, \Xi^{-1}, \Xi \end{pmatrix},$$

The call to $B\delta$ now invokes the case where the wiring is to a module and thus a new delay needs to be determined:

$$B\delta \left(\begin{array}{l} 1, \\ 2, \\ 23, \\ old\delta s \\ old\gamma s, \\ old\beta s, \\ 1, \\ 36, \\ \Xi^{-1}, \\ \Xi \end{array} \right) = Build\delta \left(\begin{array}{l} \delta_{1,2,23}, \\ t - ((pc\_val - pc\_val_{tgt}^{res}) \bmod Max_N^{tgt}) \end{array} \right)$$

where:

$$\begin{aligned} pc\_val_{src} &= snd \left(RetTerm \left(GetEl \left(\begin{array}{l} \Xi^{-1}, \\ 1, \\ 23 \end{array} \right), 2 \right) \right) \\ &= snd(RetTerm(\Xi^{-1}(1, 23) = (24, 0), 2)) \\ &= snd((24, 0)) \\ &= 0 \end{aligned}$$

and:

$$\begin{aligned} mod\_val_{src} &= fst \left(RetTerm \left(GetEl \left(\begin{array}{l} \Xi^{-1}, \\ 1, \\ 23 \end{array} \right), 2 \right) \right) \\ &= fst(RetTerm(\Xi^{-1}(1, 23) = (24, 0), 2)) \\ &= fst((24, 0)) \\ &= 24 \end{aligned}$$

which means that:

$$\begin{aligned} mod\_val^{res}_{src} &= fst\left(RetTerm\left(GetEl\left(\begin{array}{l} old\beta s, \\ 0, \\ 24, \\ 2 \end{array}\right), 2\right)\right) \\ &= fst(RetTerm(\beta_0(24,2) = 28, 2)) \\ &= fst(28) \\ &= 28 \end{aligned}$$

and:

$$\begin{aligned} pc\_val^{res}_{src} &= pc\_val_{src} - \left(t - RetTerm\left(GetEl\left(\begin{array}{l} old\delta s, \\ 24, \\ 2, \\ 0 \end{array}\right), 2\right)\right) \bmod 1 \\ &= 0 - (t - RetTerm\left(\delta_{24,2,0}(t,a,x) = t-1, 2\right)) \bmod 1 \\ &= 0 - (t - t + 1) \bmod 1 \\ &= 0 - 1 \bmod 1 \\ &= 0 \end{aligned}$$

the value of $pc\_val^{res}_{tgt}$ can therefore be determined to be:

$$\begin{aligned} pc\_val^{res}_{tgt} &= snd\left(RetTerm\left(GetEl\left(\begin{array}{l} \Xi, \\ 28, \\ 0 \end{array}\right), 2\right)\right) \\ &= snd\left(RetTerm\left(\Xi(28,0) = (1,27), 2\right)\right) \\ &= snd\left((1,27)\right) \\ &= 27 \end{aligned}$$

finally, the instantiation of $B\delta$ can be completed as:

$$B\delta\begin{pmatrix}1,\\2,\\23,\\old\delta s\\old\gamma s,\\old\beta s,\\1,\\36,\\\Xi^{-1},\\\Xi\end{pmatrix} = Build\delta\begin{pmatrix}\delta_{1,2,23},\\t-((23-27) \bmod 36)\end{pmatrix}$$

$$= Build\delta\begin{pmatrix}\delta_{1,2,23},\\t-32)\end{pmatrix}$$

$$= (\delta_{1,2,23}(t,a,x) = t-32)$$

The $B\delta spc$ operation continues to recurse until it reaches the case where the program counter is 0 and then produces the delay functions for module 1 at program counter value 0. To complete the generation of delay functions, the second case of $B\delta s$ is instigated - where the module number is 0:

$$B\delta s\begin{pmatrix}0,\\old\delta s,\\new\delta s,\\4,\\old\gamma s,\\36,\\1,\Xi^{-1},\\\Xi,\end{pmatrix} = \left(Build\delta\begin{pmatrix}\delta_{0,0,0},\\t-1\end{pmatrix}, new\delta s\right)$$

$$= ((\delta_{pc,0,0}(t,a,x) = t-1), new\delta s)$$

The values of all the non-unit delays are given as:

$$\begin{array}{lll}
\delta_{1,1,0}(t,a,x) = t-35 & \delta_{1,1,6}(t,a,x) = t-29, & \delta_{1,1,22}(t,a,x) = t-34, \\
\delta_{1,2,0}(t,a,x) = t-34, & \delta_{1,2,6}(t,a,x) = t-28, & \delta_{1,2,22}(t,a,x) = t-33, \\
\delta_{1,3,0}(t,a,x) = t-33, & \delta_{1,1,9}(t,a,x) = t-24, & \delta_{1,1,23}(t,a,x) = t-33, \\
\delta_{1,1,1}(t,a,x) = t-33, & \delta_{1,2,9}(t,a,x) = t-30, & \delta_{1,2,23}(t,a,x) = t-32, \\
\delta_{1,2,1}(t,a,x) = t-32, & \delta_{1,2,10}(t,a,x) = t-30, & \delta_{1,1,24}(t,a,x) = t-32, \\
\delta_{1,1,3}(t,a,x) = t-33, & \delta_{1,1,11}(t,a,x) = t-26, & \delta_{1,2,24}(t,a,x) = t-31, \\
\delta_{1,2,3}(t,a,x) = t-32, & \delta_{1,2,11}(t,a,x) = t-30, & \delta_{1,1,25}(t,a,x) = t-31, \\
\delta_{1,3,3}(t,a,x) = t-31, & \delta_{1,2,12}(t,a,x) = t-30, & \delta_{1,2,25}(t,a,x) = t-30, \\
\delta_{1,1,4}(t,a,x) = t-31, & \delta_{1,1,13}(t,a,x) = t-28, & \delta_{1,1,26}(t,a,x) = t-30, \\
\delta_{1,2,4}(t,a,x) = t-30, & \delta_{1,2,13}(t,a,x) = t-30, & \delta_{1,2,26}(t,a,x) = t-29, \\
\delta_{1,1,5}(t,a,x) = t-30, & \delta_{1,1,21}(t,a,x) = t-35, & \delta_{1,1,27}(t,a,x) = t-29, \\
\delta_{1,2,5}(t,a,x) = t-29, & \delta_{1,2,21}(t,a,x) = t-34, & \delta_{1,2,27}(t,a,x) = t-28, \\
\delta_{1,2,14}(t,a,x) = t-30 &&
\end{array}$$

Initial State Equations

The generation of Initial State equations for the target abstract dSCA begins with a call to the $CreateIVs$ operation:

$$CreateIVs\left(\begin{array}{l} Source\_SCA, \\ 1, \\ 36, \\ \Xi^{-1} \end{array}\right) = BIVs\left(\begin{array}{l} 1, \\ 36, \\ GetEqIV(Source\_SCA), \\ [], \\ \Xi^{-1} \end{array}\right)$$

which expands to the recursive call:

$$BIVs\left(\begin{array}{l} 1, \\ 36, \\ oeqs, \\ neqs, \\ \Xi^{-1} \end{array}\right) = BIVs\left(\begin{array}{l} 0, \\ 36, \\ oeqs, \\ \left(BIVpc\left(\begin{array}{l} 35, \\ 1, \\ oeqs, \\ [] \end{array}\right), neqs\right), \\ \Xi^{-1} \end{array}\right)$$

We consider the call to $BIVpc$ first, this expands in all cases as (since the inverse mapping in our example is total and there will be no undefined operations required to maintain synchronicity):

$$BIVpc\left(\begin{array}{l}35,\\1,\\oeqs,\\[],\\\Xi^{-1}\end{array}\right) = BIVpc\left(\begin{array}{l}34,\\1,\\oeqs,\\\left(\ new\_val, []\ \right),\\\Xi^{-1}\end{array}\right)$$

where *new_val* is the following set of definitions:

$$\begin{aligned}
&= BuildIV\left(\begin{array}{l}1,\\35,\\RetTerm\left(GetEl\left(\begin{array}{l}oeqs,\\RetTerm\left(GetEl\left(\begin{array}{l}\Xi^{-1},\\1,\\35\end{array}\right),2\right)\end{array}\right),2\right)\end{array}\right)\\
&= BuildIV\left(\begin{array}{l}1,\\35,\\RetTerm\left(GetEl\left(\begin{array}{l}oeqs,\\RetTerm\left(\Xi^{-1}(1,35) = (36,0),2\right)\end{array}\right),2\right)\end{array}\right)\\
&= BuildIV\left(\begin{array}{l}1,\\35,\\RetTerm\left(GetEl\left(\begin{array}{l}oeqs,\\36,\\0\end{array}\right),2\right)\end{array}\right)\\
&= BuildIV\left(\begin{array}{l}1,\\35,\\RetTerm\left(V_{36}(0,a,x) = 0,2\right)\end{array}\right)\\
&= BuildIV\left(\begin{array}{l}1,\\35,\\0\end{array}\right)\\
&= (V_1(35,a,x) = 0)
\end{aligned}$$

The recursive call to $BIVpc$ above therefore becomes:

$$BIVpc\left(\begin{array}{l}35,\\1,\\oeqs,\\[],\\\Xi^{-1}\end{array}\right) = BIVpc\left(\begin{array}{l}34,\\1,\\oeqs,\\\left(\ V_1(35,a,x) = 0, []\ \right),\\\Xi^{-1}\end{array}\right)$$

This process continues for all values of the program counter for module 1, and then the recursive call to $BIVs$ where the module number is zero is made. In this situation, the following case of the $BIVs$ operation is invoked:

$$BIVs\begin{pmatrix} 0, \\ 35, \\ oeqs, \\ neqs, \\ \Xi^{-1} \end{pmatrix} = (BpcIVs(36, []), neqs)$$

where the call to $BpcIVs$ is expanded as:

$$\begin{aligned} BpcIVs\begin{pmatrix} 35, \\ [], \\ 36 \end{pmatrix} &= BpcIVs\left(\begin{array}{l} 34, \\ \left(BuildIV\begin{pmatrix} 0, \\ 35, \\ mod(35+1,36) \end{pmatrix}, []\right), \\ 36 \end{array}\right) \\ &= BpcIVs\begin{pmatrix} 34, \\ (V_0(35,a,x) = 0, []), \\ 36 \end{pmatrix} \end{aligned}$$

The recursion progresses, finally making a call to the base case where the program counter value is 0:

$$\begin{aligned} BpcIVs\begin{pmatrix} 0, \\ eqs, \\ 36 \end{pmatrix} &= \left(BuildIV\begin{pmatrix} 0, \\ 0, \\ 1 \end{pmatrix}, eqs\right) \\ &= (V_0(0,a,x) = 1, eqs) \end{aligned}$$

The result of applying the mapping to the Initial State equations is:

$$
\begin{array}{lll}
V_0(0,a,x) = 1, & V_0(1,a,x) = 2, & V_0(2,a,x) = 3, \\
V_0(3,a,x) = 4, & V_0(4,a,x) = 5, & V_0(5,a,x) = 6, \\
V_0(6,a,x) = 7, & V_0(7,a,x) = 8, & V_0(8,a,x) = 9, \\
V_0(9,a,x) = 10, & V_0(10,a,x) = 11, & V_0(11,a,x) = 12, \\
V_0(12,a,x) = 13, & V_0(13,a,x) = 14, & V_0(14,a,x) = 15, \\
V_0(15,a,x) = 16, & V_0(16,a,x) = 17, & V_0(17,a,x) = 18, \\
V_0(18,a,x) = 19, & V_0(19,a,x) = 20, & V_0(20,a,x) = 21, \\
V_0(21,a,x) = 22, & V_0(22,a,x) = 23, & V_0(23,a,x) = 24, \\
V_0(24,a,x) = 25, & V_0(25,a,x) = 26, & V_0(26,a,x) = 27, \\
V_0(27,a,x) = 28, & V_0(28,a,x) = 29, & V_0(29,a,x) = 30, \\
V_0(30,a,x) = 31, & V_0(31,a,x) = 32, & V_0(32,a,x) = 33, \\
V_0(33,a,x) = 34, & V_0(34,a,x) = 35, & V_0(35,a,x) = 0, \\
V_1(0,a,x) = stay, & V_1(1,a,x) = true, & V_1(2,a,x) = stay, \\
V_1(3,a,x) = up, & V_1(4,a,x) = true, & V_1(5,a,x) = false, \\
V_1(6,a,x) = false, & V_1(7,a,x) = down, & V_1(8,a,x) = up, \\
V_1(9,a,x) = true, & V_1(10,a,x) = true, & V_1(11,a,x) = false, \\
V_1(12,a,x) = false, & V_1(13,a,x) = false, & V_1(14,a,x) = true, \\
V_1(15,a,x) = false, & V_1(16,a,x) = 90, & V_1(17,a,x) = true, \\
V_1(18,a,x) = 0, & V_1(19,a,x) = true, & V_1(20,a,x) = 0, \\
V_1(21,a,x) = false, & V_1(22,a,x) = false, & V_1(23,a,x) = false, \\
V_1(24,a,x) = false, & V_1(25,a,x) = false, & V_1(26,a,x) = false, \\
V_1(27,a,x) = false, & V_1(28,a,x) = 0, & V_1(29,a,x) = 0, \\
V_1(30,a,x) = 0, & V_1(31,a,x) = 0, & V_1(32,a,x) = 0, \\
V_1(33,a,x) = 0, & V_1(34,a,x) = 0, & V_1(35,a,x) = 0
\end{array}
$$

State Transition Equations

To commence translating the State Transition equations, a call to the *CreateSTs* operation is made:

$$CreateSTs\begin{pmatrix} Source\_SCA, \\ 1, \\ 36, \\ \Xi^{-1} \end{pmatrix} = BSTs\begin{pmatrix} 1, \\ 36, \\ GetEqSTVF(Source\_SCA), \\ [], \\ \Xi^{-1}, \\ Create\beta s\begin{pmatrix} Source\_SCA, \\ k, \\ Max_N, \\ \Xi^{-1} \end{pmatrix}, \\ Create\delta s\begin{pmatrix} Source\_SCA, \\ k, \\ Max_N, \\ \Xi, \\ \Xi^{-1}, \end{pmatrix} \end{pmatrix}$$

which in turn makes a call to the $BSTs$ operation once it has extracted the source State Transition equations. This call is expanded as:

$$BSTs\begin{pmatrix} 1, \\ 36, \\ STVFs, \\ [], \\ \Xi^{-1}, \\ tgt\beta s, \\ tgt\delta s \end{pmatrix} = BSTs\begin{pmatrix} 0, \\ \begin{pmatrix} BST\begin{pmatrix} 36, \\ STVFs, \\ \Xi^{-1}, \\ tgt\beta s, \\ tgt\delta s, \\ 1 \end{pmatrix}, \\ [] \end{pmatrix}, \\ 36 \end{pmatrix}$$

It can be seen that this expansion makes a recursive call to itself, decrementing the value of the module number as it does so, and makes a call to the BST operation

which creates the Value function equation for the module under consideration. This call to BST is expanded as follows:

$$BST\begin{pmatrix}36,\\1,\\STVFs,\\\Xi^{-1},\\\beta s,\\\delta s,\end{pmatrix} = BuildST\left(\begin{pmatrix}1,\\NewST\begin{pmatrix}rewire\begin{pmatrix}new\_vfopdef,\\1,\\36,\\\beta eqs,\\\delta eqs\end{pmatrix},\\36,\\null\end{pmatrix}\end{pmatrix}\right)$$

where:

$$new\_vfopdef = NST\begin{pmatrix}36,\\1,\\[],\\STVFs,\\\Xi^{-1}\end{pmatrix}$$

The call to NST operation, expands as:

$$NST\begin{pmatrix}35,\\1,\\[],\\oeqs,\\\Xi^{-1}\end{pmatrix} = NST\left(\begin{pmatrix}34,\\1,\\\left([], Extract\begin{pmatrix}oeqs,\\mod\_val_{src},\\pc\_val_{src}\end{pmatrix}\right),\\oeqs,\\\Xi^{-1}\end{pmatrix}\right)$$

where:

$$\begin{aligned} mod\_val_{src} &= fst\left(RetTerm\left(GetEl\left(\begin{array}{l}\Xi^{-1},\\ 1,\\ 35\end{array}\right),2\right)\right)\\ &= fst\left(RetTerm\left((\Xi^{-1}(1,35)=(36,0)),2\right)\right)\\ &= fst(36,0)\\ &= 36 \end{aligned}$$

and:

$$\begin{aligned} pc\_val_{src} &= snd\left(RetTerm\left(GetEl\left(\begin{array}{l}\Xi^{-1},\\ 1,\\ 35\end{array}\right),2\right)\right)\\ &= snd\left(RetTerm\left((\Xi^{-1}(1,35)=(36,0)),2\right)\right)\\ &= snd(36,0)\\ &= 0 \end{aligned}$$

Thus the NST call becomes:

$$\begin{aligned} NST\left(\begin{array}{l}35,\\ 1,\\ [],\\ oeqs,\\ \Xi^{-1}\end{array}\right) &= NST\left(\begin{array}{l}34,\\ 1,\\ \left([],Extract\left(\begin{array}{l}oeqs,\\ 35,\\ 0\end{array}\right)\right),\\ oeqs,\\ \Xi^{-1}\end{array}\right)\\ &= NST\left(\begin{array}{l}34,\\ 1,\\ (0),\\ oeqs,\\ \Xi^{-1}\end{array}\right) \end{aligned}$$

The next instantiation of the recursive call to NST proceeds:

$$NST\begin{pmatrix} 34, \\ 1, \\ (0), \\ oeqs, \\ \Xi^{-1} \end{pmatrix} = NST\begin{pmatrix} 33, \\ 1, \\ \left([], Extract\begin{pmatrix} oeqs, \\ 34, \\ 0 \end{pmatrix}\right), \\ oeqs, \\ \Xi^{-1} \end{pmatrix}$$
$$= NST\begin{pmatrix} 33, \\ 1, \\ (0, sub(a_7(t), a_8(t))), \\ oeqs, \\ \Xi^{-1} \end{pmatrix}$$

NST finally completes it recursion for module 1 and produces the following list:

$$0, sub(a_7(t), a_8(t)), 0, sub(a_5(t), a_6(t)), 0, sub(a_3(t), a_4(t)),$$
$$0, sub(a_1(t), a_2(t)), gt(V_{35}(t,a,x), V_{36}(t,a,x)), gt(V_{33}(t,a,x), V_{34}(t,a,x)),$$
$$gt(V_{31}(t,a,x), V_{32}(t,a,x)), gt(V_{29}(t,a,x), V_{30}(t,a,x)), or(V_{27}(t,a,x), V_{28}(t,a,x)),$$
$$or(V_{25}(t,a,x), V_{26}(t,a,x)), or(V_{23}(t,a,x), V_{24}(t,a,x)), 0, true, 0, true,$$
$$90, false, gt(a_{10}(t), V_{21}(t,a,x)), eq(V_{22}(t,a,x), V_{20}(t,a,x)), eq(a_{10}(t), V_{19}(t,a,x)),$$
$$eq(V_{22}(t,a,x), V_{18}(t,a,x)), eq(a_{10}(t), V_{17}(t,a,x)), eq(V_{22}(t,a,x), V_{16}(t,a,x)),$$
$$up, down, and(V_{14}(t,a,x), V_{15}(t,a,x)), and(V_{12}(t,a,x), V_{13}(t,a,x)),$$
$$and(V_{10}(t,a,x), V_{11}(t,a,x)),$$
$$cond\begin{pmatrix} V_7(t,a,x), \\ V_8(t,a,x), \\ V_9(t,a,x) \end{pmatrix}, stay, or(V_5(t,a,x), V_6(t,a,x)), cond\begin{pmatrix} V_4(t,a,x), \\ V_3(t,a,x), \\ V_2(t,a,x) \end{pmatrix}$$

The next part of the process is to rewire the above list for the new network, which is done by the rewire operation:

$$rewire\begin{pmatrix} (e, es), \\ 1, \\ 35, \\ \beta eqs, \\ \delta eqs \end{pmatrix} = \begin{pmatrix} rw\begin{pmatrix} e, \\ mod\_val, \\ pc\_val, \\ \beta eqs, \\ \delta eqs \end{pmatrix}, rewire\begin{pmatrix} es, \\ 1, \\ 34, \\ \beta eqs, \\ \delta eqs \end{pmatrix} \end{pmatrix}$$

The *rewire* operation recurses through the list and produces a new list, in the same order but uses the *rw* operation to rewire each VFOpDef term encountered. For

the first term, 0, the first instance of rw is used:

$$rw\begin{pmatrix} 0, \\ \beta s, \\ \delta s, \\ 1, \\ 35 \end{pmatrix} = 0$$

The next term it encounters invokes the third form of rw:

$$rw\begin{pmatrix} sub(a_7(t), a_8(t)), \\ \beta s, \\ \delta s, \\ 1, \\ 34 \end{pmatrix} = sub\left(wire\begin{pmatrix} a_7(t), \\ \beta s, \\ \delta s, \\ 1, \\ 1, \\ 34 \end{pmatrix}, wire\begin{pmatrix} a_8(t), \\ \beta s, \\ \delta s, \\ 1, \\ 2, \\ 34 \end{pmatrix}\right)$$

this invokes the use of the "input" form of the $wire$ operation:

$$\begin{aligned} wire\begin{pmatrix} a_7(t), \\ \beta s, \delta s, 1, 1, 34 \end{pmatrix} &= a_{RetTerm(GetEl(\beta s,1,1,34),2)}(t) \\ &= a_{RetTerm((\beta s_{34}(1,1)=7),2)}(t) \\ &= a_7(t) \end{aligned}$$

The second argument is:

$$\begin{aligned} wire\begin{pmatrix} a_8(t), \\ \beta s, \delta s, 1, 2, 34 \end{pmatrix} &= a_{RetTerm(GetEl(\beta s,1,2,34),2)}(t) \\ &= a_{RetTerm((\beta s_{34}(1,2)=8),2)}(t) \\ &= a_8(t) \end{aligned}$$

therefore the rewired VFOpDef term will be:

$$sub(a_7(t), a_8(t))$$

The complete list of VFOpDef terms returned from the *rewire* operation is:

$$
\begin{array}{l}
0, \\
sub(a_7(t), a_8(t)), \\
0, \\
sub(a_5(t), a_6(t)), \\
0, \\
sub(a_3(t), a_4(t)), \\
0, \\
sub(a_1(t), a_2(t)), \\
gt(V_1(t-29,a,x), V_1(t-28,a,x)), \\
gt(V_1(t-30,a,x), V_1(t-29,a,x)), \\
gt(V_1(t-31,a,x), V_1(t-30,a,x)), \\
gt(V_1(t-33,a,x), V_1(t-31,a,x)), \\
or(V_1(t-33,a,x), V_1(t-32,a,x)), \\
or(V_1(t-34,a,x), V_1(t-33,a,x)), \\
or(V_1(t-35,a,x), V_1(t-34,a,x)), \\
0, \\
true, \\
0, \\
true, \\
90, \\
false, \\
gt(a_{10}(t), V_1(t-30,a,x)), \\
eq(V_1(t-28,a,x), V_1(t-30,a,x)), \\
eq(a_{10}(t), V_1(t-30,a,x)), \\
eq(V_1(t-26,a,x), V_1(t-30,a,x)), \\
eq(a_{10}(t), V_1(t-30,a,x)), \\
eq(V_1(t-24,a,x), V_1(t-30,a,x)), \\
up, \\
down, \\
and(V_1(t-29,a,x), V_1(t-28,a,x)), \\
and(V_1(t-30,a,x), V_1(t-29,a,x)), \\
and(V_1(t-31,a,x), V_1(t-30,a,x)), \\
cond \begin{pmatrix} V_1(t-33,a,x), \\ V_1(t-32,a,x), \\ V_1(t-31,a,x) \end{pmatrix}, \\
stay, \\
or(V_1(t-33,a,x), V_1(t-32,a,x)), \\
cond \begin{pmatrix} V_1(t-35,a,x), \\ V_1(t-34,a,x), \\ V_1(t-33,a,x) \end{pmatrix}
\end{array}
$$

The original list of terms is now fed into the $NewST$ operation, the purpose of which is to construct the conditional VFOpDef term for the module under consideration by recursing the list and creating the correct format. Finally, the new State

Transition equation is constructed using the *BuildST* operation, where results, for module 1 in the following (the more readable form for the conditional operation is used where appropriate):

$$V_1(t,a,x) = \begin{cases} cond\begin{pmatrix} V_1(t-35,a,x), \\ V_1(t-34,a,x), \\ V_1(t-33,a,x) \end{pmatrix} & \text{if } V_{pc}(t-1,a,x) = 0 \\ or(V_1(t-33,a,x), V_6(t-32,a,x)) & \text{if } V_{pc}(t-1,a,x) = 1 \\ start & \text{if } V_{pc}(t-1,a,x) = 2 \\ cond\begin{pmatrix} V_1(t-33,a,x), \\ V_1(t-32,a,x), \\ V_1(t-31,a,x) \end{pmatrix} & \text{if } V_{pc}(t-1,a,x) = 3 \\ and(V_1(t-31,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 4 \\ and(V_1(t-30,a,x), V_1(t-29,a,x)) & \text{if } V_{pc}(t-1,a,x) = 5 \\ and(V_1(t-29,a,x), V_1(t-28,a,x)) & \text{if } V_{pc}(t-1,a,x) = 6 \\ down & \text{if } V_{pc}(t-1,a,x) = 7 \\ up & \text{if } V_{pc}(t-1,a,x) = 8 \\ eq(V_1(t-24,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 9 \\ eq(a_9(t), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 10 \\ eq(V_1(t-26,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 11 \\ eq(a_9(t), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 12 \\ eq(V_1(t-28,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 13 \\ gt(a_9(t), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 14 \\ false & \text{if } V_{pc}(t-1,a,x) = 15 \\ 90 & \text{if } V_{pc}(t-1,a,x) = 16 \\ true & \text{if } V_{pc}(t-1,a,x) = 17 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 18 \\ true & \text{if } V_{pc}(t-1,a,x) = 19 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 20 \\ or(V_1(t-35,a,x), V_1(t-34,a,x)) & \text{if } V_{pc}(t-1,a,x) = 21 \\ or(V_1(t-34,a,x), V_1(t-33,a,x)) & \text{if } V_{pc}(t-1,a,x) = 22 \\ or(V_1(t-33,a,x), V_1(t-32,a,x)) & \text{if } V_{pc}(t-1,a,x) = 23 \\ gt(V_1(t-33,a,x), V_1(t-31,a,x)) & \text{if } V_{pc}(t-1,a,x) = 24 \\ gt(V_1(t-31,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 25 \\ gt(V_1(t-30,a,x), V_1(t-29,a,x)) & \text{if } V_{pc}(t-1,a,x) = 26 \\ gt(V_1(t-29,a,x), V_1(t-28,a,x)) & \text{if } V_{pc}(t-1,a,x) = 27 \\ sub(a_1(t), a_2(t)) & \text{if } V_{pc}(t-1,a,x) = 28 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 29 \\ sub(a_3(t), a_4(t)) & \text{if } V_{pc}(t-1,a,x) = 30 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 31 \\ sub(a_5(t), a_6(t)) & \text{if } V_{pc}(t-1,a,x) = 32 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 33 \\ sub(a_7(t), a_8(t)) & \text{if } V_{pc}(t-1,a,x) = 34 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 35 \end{cases}$$

The final step in the process is the recursive call to the BST operation with module number equal to 0, in this case the function above is returned with the following State Transition equation for the program counter appended on to it:

$$V_{pc}(t+1,a,x) = \begin{cases} mod(add\,(V_{pc}(t,a,x),1)\,,35) & \text{if } V_{pc}(t-1,a,x) = 0 \\ \vdots & \\ mod(add\,(V_{pc}(t,a,x),1)\,,35) & \text{if } V_{pc}(t-1,a,x) = 35 \end{cases}$$

The complete translated abstract dSCA can be seen described algebraically as shown in Appendix D.

11.4 Correctness

The generated target abstract dSCA created from transforming the abstract dSCA in the previous chapter can be seen to be the same as the Form 2 abstract dSCA given in Chapter 8.3.2 - the semantic proof of correctness given in that chapter shows that this abstract dSCA is a correct implementation of a solution to the GRCP.

Additionally, this and the previous abstract dSCA could exist in a hierarchy, and this will be shown by means of introducing mappings for:

- spaces;
- clocks;
- global states; and
- input streams.

Consider the Form 1 abstract dSCA given in Chapter 10 and the Form 2 abstract dSCA given above, we will now discuss the relationship between each of the models. To clearly distinguish between the two systems reference will be made to the Form 1 abstract dSCA as N_{dSCA1} and the Form 2 abstract dSCA as N_{dSCA2}. Components of

each network will be named as the sets I_{dSCA2} and I_{dSCA1} for the modules (or spaces), the sets In_{dSCA2} and In_{dSAC1} of input streams, and the sets Ch_{dSCA2} and Ch_{dSCA1} for the channels of the networks, where:

$$
\begin{aligned}
I_{dSCA1} &= \{0, 1, 2, 3, \ldots, k\} \\
I_{dSCA2} &= \{0, 1\} \\
In_{dSCA1} &= \{0, 1, \ldots, 10\} \\
In_{dSCA2} &= \{0, 1, \ldots, 10\} \\
Ch_{dSCA1} &= \{0, 1, \ldots, 36\} \\
Ch_{dSCA2} &= \{0, 1\} \\
Out_{dSCA1} &= \{1\} \\
Out_{dSCA2} &= \{1\}
\end{aligned}
$$

Component Abstraction To compare the behaviours of N_{dSCA2} and N_{dSCA1} the mappings between their components are first defined.

Spaces. Spaces between the two SCAs are clearly related by the provided mapping function, for it is this that has been used to create the Form 2 abstract dSCA. The mapping is more complicated than a simple module to module mapping since we need to take account of the value of the program counter to understand the source module:

The respacing $\pi : I_{tgt} \times N_{Max_N} \rightarrow I_{src}$ is clearly defined by:

$$
\begin{aligned}
\pi(i, pc) &= fst : \Xi^{-1}(i, pc) && \text{for i} = 1 \\
&= 0 && \text{for i} = 0
\end{aligned}
$$

Clocks. Each clock cycle in the source SCA is represented by 36 clock cycles in the abstract dSCA. Thus a retiming $\lambda : T_{dSCA2} \rightarrow T_{dSCA1}$ can be defined for all $t \in T_{dSCA2}$ as:

$$
\lambda(t) = \left\lfloor \frac{t}{\left(\frac{Max_{N_{dSCA2}}}{Max_{N_{dSCA1}}}\right)} \right\rfloor
$$

where $Max_{N_{dSCA2}} = 1$ and $Max_{N_{dSCA1}} = 36$ thus:

$$\lambda(t) = \left\lfloor \frac{t}{\left(\frac{36}{1}\right)} \right\rfloor$$

or further simplified to¿

$$\lambda(t) = \lfloor t/36 \rfloor$$

the corresponding immersion $\bar{\lambda} : T_{dSCA1} \rightarrow T_{dSCA2}$ is defined, for all $t \in T_{dSCA1}$ by $\bar{\lambda}(t) = 36t$, and $Start_\lambda$ is defined to therefore have the values $Start_\lambda = 0, 36, 72, 108, \ldots$.

States. There is no need to introduce a data abstraction map since it is defined that all SCAs under consideration will be based on the machine algebra M_A.

States are still a measurement of the channels in the relevant SCAs, however, it now makes sense to consider observable states rather than the whole state. The most appropriate observable states for this system is the output of m_1 in the target SCA and the output of module m_{36} in the source SCA - i.e. the modules in V_{out} for each network.

The state abstraction map $\phi : M_A^{Ch_{tgt}} \rightarrow M_A^{Ch_{src}}$ is introduced for observable states $s \in M_A^{Ch_1}$ as:

$$\phi(s, pc)(1) \quad = s(pc)$$

Input Streams. Input streams are on one hand relatively simple since the transformation neither adds or removes input streams from the network. However, there are timing issues. Take the gate sensor input, this is used by several modules in the Form 1 abstract dSCA and has a delay of unit length for all these modules to make it consistent with the SCA. In the Form 2 abstract dSCA, this value is required by module 1 but at several different times as the modules from the Form 1 abstract dSCA are now executing at different values of the program counter. The most appropriate and global solution to this problem is to require the input values to be available for a whole

cycle, in this case 36 lock cycles. This allows values to be available when required (this solution would need to be considered further for cycle inconsistent abstract dSCAs).

We define an input stream abstraction $\theta : [T_{dSCA2} \rightarrow M_A]^{Ch_{dSCA2}} \rightarrow [T_{dSCA1} \rightarrow M_A]^{Ch_{dSCA1}}$ to be the identity operation since input streams do not change between models, however we need to keep the result for 36 clock cycles (consistent with the retiming); thus we write:

$$\theta(a)(t) = a(\lambda(t))$$

Abstraction of global behaviour. The notion that the global behaviour of the Form 1 abstract dSCA abstracts that of the abstract form 2 dSCA is now formalised. Let V_1 and V_2 be the global state functions determined from the channel state functions of these 2 SCAs, then it is conjectured that the following diagram commutes:

Conjecture Given the above mappings, it is believed that the following diagram commutes:

$$\begin{array}{ccccccc}
T_{dSCA1} & \times [T_{dSCA1} \rightarrow M_{A_{dSCA1}}]^{In_{dSCA1}} \times & M_{A_{dSCA1}}^{Ch_{dSCA1}} & \xrightarrow{V_{dSCA1}} & M_{A_{dSCA1}}^{Ch_{dSCA1}} \\
\uparrow \lambda & & \uparrow \theta & & \uparrow \phi & & \uparrow \phi \\
Start_\lambda & \times [T_{dSCA2} \rightarrow M_{A_{dSCA2}}]^{In_{dSCA2}} \times & M_{A_{dSCA2}}^{Ch_{dSCA2}} & \xrightarrow{V_{dSCA2}} & M_{A_{dSCA2}}^{Ch_{dSCA2}}
\end{array}$$

11.5 Concluding Comments

This chapter has demonstrated the techniques required for mapping an abstract dSCA with one defining shape to an abstract dSCA with another defining shape. A demonstration has been given by taking the Form 1 abstract dSCA solution to the GRC Problem, which represents the computation, and mapping that to a Form 2 abstract dSCA that represents the computation device.

11.6 Sources

The definition of the mapping process is all my own work.

Chapter 12

Abstract dSCA to concrete dSCA

Purpose of Transformation
To transform an abstract dSCA with a defining shape of $\nabla = (n, m)$ *to a concrete dSCA with defining shape* $\nabla = (n, m)$

12.1 Process

This chapter highlights the key parts of the processes used for the transformation of an abstract dSCA with defining shape $\nabla = (n, m)$ to a concrete dSCA with a defining shape of $\nabla = (n, m)$. Appendix H contains the complete formal definition of this transformation.

The following equation lists, within a supplied abstract dSCA specification, are considered for transformation:

1. Wiring Functions;
2. Delay Functions;
3. Initial State Equations; and

4. State Transition Equations.

This chapter's first part considers the mechanisms for such a transformation and the second part demonstrates the application of the transformations to the Form 2 abstract dSCA produced in chapter 11. Recall that this abstract dSCA has a defining shape of $\nabla = (1, 36)$. The transformation will be to a cycle consistent concrete dSCA. It should be noted that if transformation to a cycle inconsistent concrete dSCA was required then alteration of the tuple lengths and the use of appropriate tuple mapping functions (examples of which are given in Chapter 7) would have to be used.

12.1.1 Prerequisites

The following prerequisites are required for the transformation:

- The source and object networks have $k > 1$ modules and $Max_N > 0$ component specifications in their modules definitions;
- The defining shape of the target network equals that of the source network; and
- Condition definitions of each adSCA module, except the programme counter, are of the format:

$$cond(pc = 0, a, cond(pc = 1, b, cond(pc = 2, c, cond(...))))$$

12.1.2 γ-Wiring Functions

The γ-wiring functions in the target concrete dSCA will not differ much from those in the source abstract dSCA since the "look and feel" of the SCA is not being altered. What is different is the introduction of a new input to argument 1 which will require arguments $1, \ldots, n(i)$ of the abstract dSCA becoming arguments $2, \ldots, n(i) + 1$ in the concrete dSCA. The new argument introduced in concrete dSCA is a wiring of the first argument to the output of the module itself.

Informally, to generate the target concrete dSCA γ-wiring functions from a source abstract dSCA the following process is followed:

- For each module m_i where $i \in \mathbb{N}_{k_2}$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$:
    * For each argument where $j \in \{2, \ldots, n(i) + 1\}$ create a new β-wiring function
    $$\gamma^2_{pc\_val}(i, j) = \gamma^1_{pc\_val}(i, j - 1)$$
    * For the o^{th} argument of each module create:
    $$\gamma'_{pc\_val}(i, 0) = M$$
    * For the 1^{st} argument of each module create:
    $$\gamma'_{pc\_val}(i, 1) = M$$
- For module 0 create Max_N β-wiring functions to wire m_0 back to itself.

12.1.3 β-Wiring Functions

In a similar way to how the target concrete dSCA γ-wiring functions were constructed from source abstract dSCA γ-wiring functions, so are the concrete dSCA β-wiring functions. The β-wiring functions in the target concrete dSCA again differ only in so much that the index of arguments $1, \ldots, n(i)$ shifts to $2, \ldots, n(i) + 1$.

Informally, to generate the target concrete dSCA β-wiring functions from a source abstract dSCA the following process is used:

- For each module m_i where $i \in \mathbb{N}_{k_2}$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$:

* For each argument where $j \in \{2, \ldots, n(i)+1\}$ create a new β-wiring function

$$\beta^2_{pc\_val}(i,j) = \beta^1_{pc\_val}(i,j-1)$$

* For the o^{th} argument of each module create:

$$\beta'_{pc\_val}(i,0) = M$$

* For the 1^{st} argument of each module create:

$$\beta'_{pc\_val}(i,1) = M$$

- For module 0 create Max_N β-wiring functions to wire m_0 back to itself.

12.1.4 Delay Functions

Delay functions for the concrete dSCA are all of unit delay, and there are a number equal to the wiring functions. Thus, a unit delay function will be created for every element in the newly generated γ-wiring equation list.

12.1.5 Initial State Equations

The Initial States for each module m_i, where $i \in \mathbb{N}_{k_2}$ are Max_N tuples of length Max_N (recall that the mapping is being defined for a cycle consistent abstract dSCA). We will make use of the fact that calculations will only care about the initial state given for $t = Max_N - 1$ and $t = 0$, by defining the tuple at time $t = 0$ and use that value for all other initial values until $t = Max - N - 1$ where the final initial state equation will be generated.

The usual recursive equations are given for walking the structure of the abstract source dSCA, rsulting in a call to the BIV operation:

$$BIV : N^3 \times dSCAISVEqList^2 \rightarrow dSCAISVEqList$$

which is defined recursively in two cases over the first argument. Firstly for when the first argument does not equal Max_N, then the operation is dealing with an initial state from a time prior to $t = Max_N - 1$, and as such an initial state will be created containing u elements in all positions, except for the first element (note that the positioning of the first element is dependant upon the tuple management schemes used, however for both schemes identified as of interest the first generated value is placed at position 0 in the tuple). BIV is defined as:

$$BIV\begin{pmatrix} pc\_val, \\ Max_N, \\ mod\_num, \\ oeqs, \\ neqs \end{pmatrix} = BIV\begin{pmatrix} pc\_val + 1, \\ Max_N, \\ mod\_num, \\ oeqs, \\ \begin{pmatrix} GenIVs\begin{pmatrix} mod\_num, \\ pc\_val, \\ Max_N, \\ oeqs \end{pmatrix}, neqs \end{pmatrix}, \end{pmatrix}$$

The $GenIVs$ operation used in BIV is given as:

$$GenIVs : N^3 \times dSCAISVEqList \rightarrow dSCAISVEquation$$

and is defined to create a Max_N length tuple with the first element being the initial value produced at time $t = 0$ in the source abstract dSCA initial values:

$$GenIVs\begin{pmatrix} mod\_num, \\ pc\_val, \\ Max_N, \\ oeqs \end{pmatrix} = BuildIV\begin{pmatrix} mod\_num, \\ pc\_val, \\ \begin{pmatrix} RetTerm(VF, 2), \\ u_0, \ldots, u_{Max_N - 2} \end{pmatrix} \end{pmatrix}$$

where:

$$VF = GetEl(oeqs, mod\_num, pc\_val)$$

The second case definition of BIV, where $t = Max_N - 1$, is given such that it constructs the complete initial state needed at time $Max_{N_2} - 1$ as:

$$BIV\begin{pmatrix} Max_N, \\ Max_N, \\ mod\_num, \\ oeqs, \\ neqs \end{pmatrix} = \begin{pmatrix} BuildIV\begin{pmatrix} mod\_num, \\ pc\_val, \\ InitState\begin{pmatrix} Max_N, \\ mod\_num, \\ oeqs, \\ [] \end{pmatrix} \end{pmatrix}, neqs \end{pmatrix}$$

The operation *InitState* is where the Initial State for module *mod_num* at time $t = Max_N - 1$ is created. Since we are using the array tuple management then the Initial State under these conditions will consist of a list of values with the first being the element calculated at $t = 0$ and the last being the one calculated at $t = Max_N$ in the source abstract dSCA. It is given as:

$$InitState : N^2 \times dSCAISVEqList \times TermList \rightarrow TermList$$

and is defined recursively, with the recursive case:

$$InitState\begin{pmatrix} pc\_val, \\ mod\_num, \\ oeqs, \\ nlist \end{pmatrix} = InitState\begin{pmatrix} pc\_val - 1, \\ mod\_num, \\ oeqs, \\ (RetTerm\,(VF, 2)\,, nlist) \end{pmatrix}$$

and the recursion being stopped by the 1^{st} argument reaching 0:

$$InitState\begin{pmatrix} 0, \\ mod\_num, \\ oeqs, \\ nlist, \\ \Xi^{-1} \end{pmatrix} = (RetTerm\,(VF, 2)\,, nlist)$$

where in both cases:

$$VF = GetEl(oeqs, mod\_num, pc\_val)$$

The base call to the recursive $BIVs$ operation is where the initial state for the program counter is given. It is defined as:

$$BIVs \left(\begin{array}{l} 0, \\ Max_N, \\ oeqs, \\ neqs \end{array} \right) = \left(BIVpc \left(\begin{array}{l} Max_N - 1, \\ [], \\ Max_N \end{array} \right), neqs \right)$$

where $BIVpc$ is given as:

$$BIVpc : N \times dSCAISVEqList \times N \rightarrow dSCAISVEqList$$

and is defined recursively over the values in Max_N, such that:

$$BIVpc \left(\begin{array}{l} pc\_val, \\ neqs, \\ Max_N \end{array} \right) = BIVpc \left(\begin{array}{l} pc\_val - 1, \\ \left(BuildIV \left(\begin{array}{l} 0, \\ Max_N, \\ pc\_val + 1 \bmod Max_N \end{array} \right), neqs \right), \\ Max_N \end{array} \right),$$

and:

$$BIVpc \left(\begin{array}{l} 0, \\ neqs, \\ Max_N \end{array} \right) = \left(BuildIV \left(\begin{array}{l} 0, \\ 0, \\ 1 \end{array} \right), neqs \right)$$

12.1.6 State Transition Equations

Consider the format of the State Transition equation in the source abstract dSCA, it will be similar to:

$$V_i(t,a,x) = \begin{cases} \vdots & \\ or(V_1(t-32,a,x), V_1(t-31,a,x)) & \text{if } V_{pc}(t-1,a,x) = 23 \\ gt(V_1(t-31,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 24 \\ \vdots & \end{cases}$$

the corresponding component specification in the concrete dSCA would be of the form:

$$V_i(t+1,a,x) = \begin{cases} \vdots & \\ \Upsilon \left(\begin{array}{l} V_{pc}(t,a,x), \\ V_1(t,a,x), \\ or \left(\begin{array}{l} \Pi^{tl}_{d_{1,1,23}}(V_1(t,a,x)), \\ \Pi^{tl}_{d_{1,2,23}}(V_1(t,a,x)) \end{array} \right) \end{array} \right) & \text{if } V_{pc}(t,a,x) = 23 \\ \Upsilon \left(\begin{array}{l} V_{pc}(t,a,x), \\ V_1(t,a,x), \\ gt \left(\begin{array}{l} \Pi^{tl}_{1,1,24}(V_1(t,a,x)), \\ \Pi^{tl}_{1,2,24}(V_1(t,a,x)) \end{array} \right) \end{array} \right) & \text{if } V_{pc}(t,a,x) = 24 \\ \vdots & \end{cases}$$

The differences are attributable to the introduction of the tuple management functions, Υ and Π (as well as the need to identify the value in the tuple that results are to be extracted from).

Informally, the process for creating the new State Transition equations is a two step process

- Generate the d functions - those that are used in the projection part of the tuple management functions

- Create the new State Transition equations.

We consider the two key steps of creating the d functions and the new State Transition equation in more detail next.

Generation of the d functions

For an indexed array tuple management approach the results are stored relative to the value of the program counter when that result was calculated. The values of the d functions for each argument, given a cycle consistent dSCA, can be determined by using the following formula:

$$d_{mod\_num,arg\_num,pc\_val} = (Max_N + pc\_val - \delta_{mod\_num,arg\_num,pc\_val})$$

As an example, if a module has a definition:

$$V_i(t,a,x) = \begin{cases} \vdots \\ cond \begin{pmatrix} V_1(t-34,a,x), \\ V_1(t-33,a,x), \\ V_1(t-32,a,x) \end{pmatrix} & \text{if } \mathrm{V_{pc}(t-1,a,x)} = 0 \\ \vdots \end{cases}$$

Then its arguments would be stored at positions 1,2 and 3 in the array. Assuming $Max_N = 36$, then if the first argument is considered, $d_{1,2,0}$ can be determined as:

$$d_{1,2,0} = (36 + 0 - \delta_{1,2,0}) - 1$$

From the definition of the value function it can be seen that $\delta_{1,2,0}(t,a,x) = t - 34$, therefore:

$$\begin{aligned} d_{1,2,0} &= (36 + 0 - 34) - 1 \\ &= (2) - 1 \\ &= 1 \end{aligned}$$

To generate the d functions the *Createds* operation is introduced that recurses over the structure of the concrete dSCA (since the source abstract dSCA and concrete dSCA are the same "shape" means there is no requirement to use the mapping function).

Having produced the d functions for the new network attention can be returned to the generation of the State Transition equations. Consider again the format of the State Transition equation in the source abstract dSCA, it will be similar to:

$$V_i(t,a,x) = \begin{cases} \vdots & \\ or(V_1(t-32,a,x), V_1(t-31,a,x)) & \text{if } cond1 \\ gt(V_1(t-31,a,x), V_1(t-30,a,x)) & \text{if } cond2 \\ \vdots & \end{cases}$$

and the corresponding component specification in the concrete dSCA would be of the form:

$$V_i(t+1,a,x) = \begin{cases} \vdots & \\ \Upsilon \left(\begin{array}{l} V_{pc}(\delta_{1,0,23}(t,a,x),a,x), \\ V_1(\delta_{1,1,23}(t,a,x),a,x), \\ or \left(\begin{array}{l} \Pi^{tl}_{d_{1,2,23}}(V_1(\delta_{1,2,23}(t,a,x),a,x)), \\ \Pi^{tl}_{d_{1,3,23}}(V_1(\delta_{1,3,23}(t,a,x),a,x)) \end{array} \right) \end{array} \right) & \text{if } cond1 \\ \Upsilon \left(\begin{array}{l} V_{pc}(\delta_{1,0,24}(t,a,x),a,x), \\ V_1(\delta_{1,1,24}(t,a,x),a,x), \\ gt \left(\begin{array}{l} \Pi^{tl}_{d_{1,2,24}}(V_1(\delta_{1,2,24}(t,a,x),a,x)), \\ \Pi^{tl}_{d_{1,3,24}}(V_1(\delta_{1,3,24}(t,a,x),a,x)) \end{array} \right) \end{array} \right) & \text{if } cond2 \\ \vdots & \end{cases}$$

The structure of the function does not change, except the introduction of the tuple management operations Υ and Π, so the operation can create the new State Transition equations by recursing over the list of source State Transition equations. This is done using the *CreateSTs* operation:

$$CreateSTs : adSCAAlgebra \times Function^2 \rightarrow dSCASTVEqList$$

which in turns finally calls a cs_rewire operation that is responsible for inserting the tuple management functions into the State Transition equation. The true path

component, i.e. the functionality that is used if the conditional component is true, needs to be manipulated to incorporate the tuple management functions, i.e given a component specification:

$$cond(a, b, c)$$

then b would be transformed into:

$$\Upsilon \left(\begin{array}{l} V_{pc}(t, a, x), \\ V_{mod\_num}(t, a, x), \\ rewire(b) \end{array} \right)$$

To achieve this the *cs_rewire* operation is introduced:

$$\begin{aligned} cs\_rewire: \; & Term \times N^3 \times ProjEqList \times \gamma dSCAEqList \times \\ & \beta dSCAEqList \times \delta dSCAEqList \times Function \rightarrow Term \end{aligned}$$

and it is defined as:

$$cs\_rewire \left(\begin{array}{l} trm, \\ mod\_num, \\ pc\_val, \\ Max_N, \\ ds, \\ \beta s, \\ \delta s, \\ \Upsilon, \\ \Pi \end{array} \right) = \Upsilon \left(\begin{array}{l} V_{pc}(t, a, x), \\ V_{mod\_num}(t, a, x), \\ rw \left(\begin{array}{l} trm, \\ mod\_num, \\ pc\_val, \\ Max_N, \\ ds, \\ \beta s, \\ \delta s, \\ \Pi \end{array} \right) \end{array} \right)$$

The rewire operation is responsible for rewiring the network using the new delay and wiring functions.

12.1.7 Transformation Process

Each of the operations above need to be coordinated together so that a new concrete dSCA can be created by transforming the source abstract dSCA. The *Create_cdSCA*

operation is provided to do this, it is given as:

$$Transform : adSCAAlgebra \times Function^2 \rightarrow cdSCAAlgebra$$

The operation takes the source concrete dSCA and is defined as:

$$Transform \begin{pmatrix} SCA_{src}, \\ \Upsilon, \\ \Pi \end{pmatrix} = CreatecdSCA \begin{pmatrix} Name, \\ M_{Atup} \ , \\ [], \\ [], \\ VFOp, \\ \gamma_0 : N^2 \rightarrow \{M, S, U\}, \\ \beta_0 : N^2 \rightarrow N, \\ \delta Op \ , \\ Create\gamma s(SCA_{src}) \ , \\ Create\beta s(SCA_{src}) \ , \\ Create\delta s(SCA_{src}) \ , \\ CreateIVs(SCA_{src}) \ , \\ CreateSTs \begin{pmatrix} SCA_{src}, \\ \Upsilon, \\ \Pi \end{pmatrix} \end{pmatrix}$$

where:

$$VFOp = \begin{pmatrix} V_0 : T \times M_{Atup}^n \times M_{Atup}^k \rightarrow M_{Atup}, \\ \vdots, \\ V_k : T \times M_{Atup}^n \times M_{Atup}^k \rightarrow M_{Atup} \end{pmatrix}$$

$$\delta Op = \begin{pmatrix} \delta_{0,0,0} : T \times M_{Atup}^n \times M_{Atup}^k \rightarrow T, \\ \vdots, \\ \delta_{i,j,0} : T \times M_{Atup}^n \times M_{Atup}^k \rightarrow T \end{pmatrix}$$

and:

$$k = num\_mod(SCA_{src})$$
$$j = Get\_MaxA(SCA_{src})$$
$$n = num\_inp(SCA_{src})$$

It is not intended to bring together all the operations defined in this chapter into a written down specification in this thesis for reasons of brevity. If this was to be performed, then it would appear similar to the specification provided for the SCA to abstract dSCA transformation in Appendix F.

12.2 Correctness of transformation

Theorem 12.2.1. *The transformation of a Form 2 abstract dSCA to a concrete dSCA preserves correctness.*

The Form 2 abstract dSCA and transformed result, the concrete dSCA, exist in a hierarchy and it is possible to show that the transformation is correct by considering Poole, Holden and Tucker's work on hierarchy of Spatially Expanded Systems.

Let N_{dSCA1} be a $\mathbb{N}_k^{dSCA1} > 1$ module source Form 2 abstract dSCA network with $n^{dSCA1} > 0$ sources processing data from a set M_A^{dSCA1} against a global clock T^{dSCA1}

Let N_{dSCA2} be a $\mathbb{N}_k^{dSCA2} > 1$ module concrete dSCA network with $n^{dSCA2} > 0$ sources processing data from a set M_A^{dSCA2} against a global clock T^{dSCA2} as generated from N_{dSCA1} using the abstract dSCA to concrete dSCA transformation.

Poole, Holden and Tucker claimed that if it was possible to generate appropriate mappings and show the following diagram commutes then the two spatially expanded systems under consideration were correct with respect to each other.

$$\begin{array}{llll} T_{dSCA1} & \times[T_{dSCA1} \rightarrow M_{A_{dSCA1}}]^{In_{dSCA1}} \times & M_{A_{dSCA1}}^{Ch_{dSCA1}} & \xrightarrow{V_{dSCA1}} M_{A_{dSCA1}}^{Ch_{dSCA1}} \\ \uparrow \lambda & \uparrow \theta & \uparrow \phi & \uparrow \phi \\ Start_{\lambda} & \times[T_{dSCA2} \rightarrow M_{A_{dSCA2}}]^{In_{dSCA2}} \times & M_{A_{dSCA2}}^{Ch_{dSCA2}} & \xrightarrow{V_{dSCA2}} M_{A_{dSCA2}}^{Ch_{dSCA2}} \end{array}$$

Mappings are needed for four areas:

- spaces;
- clocks;
- global states; and
- input streams.

The mappings are defined as follows:

Spaces. Spaces (modules) in the two networks are equivalent, for modules m_i where $i \in \mathbb{N}_k^{dSCA2}$ thus it is appropriate to define the respacing operation $\pi : I_{N_{dSCA2}} \rightarrow I_{N_{dSCA1}}$ as:

$$\pi(i) = i$$

Clocks. There is no timing abstraction between networks, thus the retiming $\lambda : T^{dSCA2} \rightarrow T^{dSCA1}$, where for $t \in T^{dSCA2}$, can be appropriately defined as:

$$\lambda(t) = t$$

Input Streams. There are no data or temporal abstractions required for inputs since these are not altered by the transformation. Thus is it appropriate to define the input stream abstraction $\theta : [T^{dSCA2} \rightarrow M_{A_{dSCA2}}]^{n^{SCA2}} \rightarrow [T^{dSCA1} \rightarrow M_{A_{dSCA1}}]^{n^{dSCA1}}$ as the operation:

$$\begin{aligned} \theta(a)(t) &= a(\lambda(t)) \\ &= a(t) \end{aligned}$$

(Note: recall that we are comparing inputs between the Form 2 abstract and concrete dSCA).

Global States. It is defined in the transformation that the carrier data set for source abstract dSCA and target concrete dSCA differ by the introduction of tuples. These means that the state abstraction map is related to the tuple management functions (namely the tuple insertion function)

We therefore consider the state abstraction map $\phi : M_{A_{tup}}^{Ch_{adSCA2}} \rightarrow M_{A_{dSCA1}}^{Ch_{adSCA1}}$ for all states $s \in M_{A_{adSCA2}}^{Ch_{adSCA2}}$ to be defined as follows, for $i \in \mathbb{N}_k^{dSCA2}$ (assuming the use of queue tuple management functions):

$$\phi(s)(i) = \Pi_0^{Max_N} s(i)$$

Conjecture Given this set of mappings it is believed that the diagram above commutes, and proof of such is done in a similar manner as for Theorem 10.2.1.

12.3 Generalised Railroad Crossing Problem as a single processor Concrete dSCA

Finally the transformation of the (source) abstract dSCA from the previous chapter, which has a defining shape of $\nabla = (1, k)$, to the (target) concrete dSCA with a defining shape of $\nabla = (1, k)$, using the array style tuple management functions is considered. The prerequisites are reviewed first:

- The source and object networks have $k > 1$ modules and $Max_n > 0$ component specifications in its modules definitions; and
- Condition definitions of each adSCA module, except the programme counter, are of the format:

$$cond(pc = 0, a, cond(pc = 1, b, cond(pc = 2, c, cond(...))))$$

γ-wiring Functions

The process of transformation first creates the concrete dSCA γ-wiring functions. To do so, the $Create\gamma s$ operation is called as:

$$\begin{aligned} Create\gamma s\left(\ Source\_SCA,\ \right) &= B\gamma s\left(\begin{array}{l} GetNumModules(Source\_SCA), \\ GetMaxN(Source\_SCA), \\ Get\gamma Eqs(Source\_SCA), \end{array}\right) \\ &= B\gamma s\left(\begin{array}{l} 1, \\ 36, \\ Get\gamma Eqs(Source\_SCA), \end{array}\right) \end{aligned}$$

The call to $B\gamma s$ expands as follows:

$$B\gamma s\left(\begin{array}{l} 1, \\ 36, \\ old\gamma s, \end{array}\right) = Rewire\gamma s\left(\begin{array}{l} 1, \\ 36, \\ Reindex\gamma s\,(old\gamma s) \end{array}\right)$$

This call to the $Reindex\gamma s$ operation expands as:

$$Reindex\gamma s(e, es) = (Reindex\gamma(e), Reindex\gamma s(es))$$

The list of abstract dSCA γ-wiring functions contains as the first 8 elements:

$$\begin{array}{l} \gamma_0(1,0) = M, \\ \gamma_0(1,1) = M, \\ \gamma_0(1,2) = M, \\ \gamma_0(1,3) = M, \\ \gamma_1(1,0) = M, \\ \gamma_1(1,1) = M, \\ \gamma_1(1,2) = M, \\ \gamma_0(0,0) = M \end{array}$$

So if $Reindex\gamma s$ is applied to this list, the results would be:

$$Reindex\gamma s \begin{pmatrix} \gamma_0(1,0) = M, \\ \gamma_0(1,1) = M, \\ \gamma_0(1,2) = M, \\ \gamma_0(1,3) = M, \\ \gamma_1(1,0) = M, \\ \gamma_1(1,1) = M, \\ \gamma_1(1,2) = M, \\ \gamma_0(0,0) = M \end{pmatrix} = \begin{pmatrix} Reindex\gamma(\gamma_0(1,0) = M), \\ Reindex\gamma s \begin{pmatrix} \gamma_0(1,1) = M, \\ \gamma_0(1,2) = M, \\ \gamma_0(1,3) = M, \\ \gamma_1(1,0) = M, \\ \gamma_1(1,1) = M, \\ \gamma_1(1,2) = M, \\ \gamma_0(0,0) = M \end{pmatrix} \end{pmatrix}$$

$$= \vdots$$

$$= \begin{pmatrix} Reindex\gamma(\gamma_0(1,0) = M), \\ Reindex\gamma(\gamma_0(1,1) = M), \\ Reindex\gamma(\gamma_0(1,2) = M), \\ Reindex\gamma(\gamma_0(1,3) = M), \\ Reindex\gamma(\gamma_1(1,0) = M), \\ Reindex\gamma(\gamma_1(1,1) = M), \\ Reindex\gamma(\gamma_1(1,2) = M), \\ Reindex\gamma(\gamma_0(0,0) = M) \end{pmatrix}$$

which, after applying the $Reindex\gamma$ operation would result in the following list of γ-wiring functions:

$$\begin{gathered} \gamma_0(1,0) = M, \\ \gamma_0(1,2) = M, \\ \gamma_0(1,3) = M, \\ \gamma_0(1,4) = M, \\ \gamma_1(1,0) = M, \\ \gamma_1(1,2) = M, \\ \gamma_1(1,3) = M, \\ \gamma_0(0,0) = M \end{gathered}$$

Application of the $\gamma-$ wiring transformation to the complete list of $\gamma-$wiring operations from the abstract dSCA produces the following list of γ-wiring function for

module 1 of the concrete dSCA:

$\gamma_0(1,0) = M,$	$\gamma_0(1,2) = M,$	$\gamma_0(1,3) = M,$	$\gamma_0(1,4) = M,$
$\gamma_1(1,0) = M,$	$\gamma_1(1,2) = M,$	$\gamma_1(1,3) = M,$	$\gamma_1(1,4) = U,$
$\gamma_2(1,0) = M,$	$\gamma_2(1,2) = U,$	$\gamma_2(1,3) = U,$	$\gamma_2(1,4) = U,$
$\gamma_3(1,0) = M,$	$\gamma_3(1,2) = M,$	$\gamma_3(1,3) = M,$	$\gamma_3(1,4) = M,$
$\gamma_4(1,0) = M,$	$\gamma_4(1,2) = M,$	$\gamma_4(1,3) = M,$	$\gamma_4(1,4) = U,$
$\gamma_5(1,0) = M,$	$\gamma_5(1,2) = M,$	$\gamma_5(1,3) = M,$	$\gamma_5(1,4) = U,$
$\gamma_6(1,0) = M,$	$\gamma_6(1,2) = M,$	$\gamma_6(1,3) = M,$	$\gamma_6(1,4) = U,$
$\gamma_7(1,0) = M,$	$\gamma_7(1,2) = U,$	$\gamma_7(1,3) = U,$	$\gamma_7(1,4) = U,$
$\gamma_8(1,0) = M,$	$\gamma_8(1,2) = U,$	$\gamma_8(1,3) = U,$	$\gamma_8(1,4) = U,$
$\gamma_9(1,0) = M,$	$\gamma_9(1,2) = M,$	$\gamma_9(10,3) = M,$	$\gamma_9(1,4) = U,$
$\gamma_{10}(1,0) = M,$	$\gamma_{10}(1,2) = S,$	$\gamma_{10}(1,3) = M,$	$\gamma_{10}(1,4) = U,$
$\gamma_{11}(1,0) = M,$	$\gamma_{11}(1,2) = M$	$\gamma_{11}(1,3) = M,$	$\gamma_{11}(1,4) = U,$
$\gamma_{12}(1,0) = M,$	$\gamma_{12}(1,2) = S,$	$\gamma_{12}(1,3) = M,$	$\gamma_{12}(1,4) = U,$
$\gamma_{13}(1,0) = M,$	$\gamma_{13}(1,2) = M,$	$\gamma_{13}(1,3) = M,$	$\gamma_{13}(1,4) = U,$
$\gamma_{14}(1,0) = M,$	$\gamma_{14}(1,2) = S,$	$\gamma_{14}(1,3) = M,$	$\gamma_{14}(1,4) = U,$
$\gamma_{15}(1,0) = M,$	$\gamma_{15}(1,2) = U,$	$\gamma_{15}(1,3) = U,$	$\gamma_{15}(1,4) = U,$
$\gamma_{16}(1,0) = M,$	$\gamma_{16}(1,2) = U,$	$\gamma_{16}(1,3) = U,$	$\gamma_{16}(1,4) = U,$
$\gamma_{17}(1,0) = M,$	$\gamma_{17}(1,2) = U,$	$\gamma_{17}(1,3) = U,$	$\gamma_{17}(1,4) = U,$
$\gamma_{18}(1,0) = M,$	$\gamma_{18}(1,2) = U,$	$\gamma_{18}(1,3) = U,$	$\gamma_{18}(1,4) = U,$
$\gamma_{19}(1,0) = M,$	$\gamma_{19}(1,2) = U,$	$\gamma_{19}(1,3) = U,$	$\gamma_{19}(1,4) = U,$
$\gamma_{20}(1,0) = M,$	$\gamma_{20}(1,2) = U,$	$\gamma_{20}(1,3) = U,$	$\gamma_{20}(1,4) = U,$
$\gamma_{21}(1,0) = M,$	$\gamma_{21}(1,2) = M,$	$\gamma_{21}(1,3) = M,$	$\gamma_{21}(1,4) = U,$
$\gamma_{22}(1,0) = M,$	$\gamma_{22}(1,2) = M,$	$\gamma_{22}(1,3) = M,$	$\gamma_{22}(1,4) = U,$
$\gamma_{23}(1,0) = M,$	$\gamma_{23}(1,2) = M,$	$\gamma_{23}(1,3) = M,$	$\gamma_{23}(1,4) = U,$
$\gamma_{24}(1,0) = M,$	$\gamma_{24}(1,2) = M,$	$\gamma_{24}(1,3) = M,$	$\gamma_{24}(1,4) = U,$
$\gamma_{25}(1,0) = M,$	$\gamma_{25}(1,2) = M,$	$\gamma_{25}(1,3) = M,$	$\gamma_{25}(1,4) = U,$
$\gamma_{26}(1,0) = M,$	$\gamma_{26}(1,2) = M,$	$\gamma_{26}(1,3) = M,$	$\gamma_{26}(1,4) = U,$
$\gamma_{27}(1,0) = M,$	$\gamma_{27}(1,2) = M,$	$\gamma_{27}(1,3) = M,$	$\gamma_{27}(1,4) = U,$
$\gamma_{28}(1,0) = M,$	$\gamma_{28}(1,2) = S,$	$\gamma_{28}(1,3) = S,$	$\gamma_{28}(1,4) = U,$
$\gamma_{29}(1,0) = M,$	$\gamma_{29}(1,2) = U,$	$\gamma_{29}(1,3) = U,$	$\gamma_{29}(1,4) = U,$
$\gamma_{30}(1,0) = M,$	$\gamma_{30}(1,2) = S,$	$\gamma_{30}(1,3) = S,$	$\gamma_{30}(1,4) = U,$
$\gamma_{31}(1,0) = M,$	$\gamma_{31}(1,2) = U,$	$\gamma_{31}(1,3) = U,$	$\gamma_{31}(1,4) = U,$
$\gamma_{32}(1,0) = M,$	$\gamma_{32}(1,2) = S,$	$\gamma_{32}(1,3) = S,$	$\gamma_{32}(1,4) = U,$
$\gamma_{33}(1,0) = M,$	$\gamma_{33}(1,2) = U,$	$\gamma_{33}(1,3) = U,$	$\gamma_{33}(1,4) = U,$
$\gamma_{34}(1,0) = M,$	$\gamma_{34}(1,2) = S,$	$\gamma_{34}(1,3) = S,$	$\gamma_{34}(1,4) = U,$
$\gamma_{35}(1,0) = M,$	$\gamma_{35}(1,2) = U,$	$\gamma_{35}(1,3) = U,$	$\gamma_{35}(1,4) = U$

and for the program counter, where $0 \leq pc\_val \leq Max_N - 1$ the following γ-wiring functions are produced:

$$\gamma_{pc\_val}(pc, 0) = M$$

After the $Reindex\gamma s$ operation completes, the result is used as an input to the call to the *Rewire* operation:

$$Rewire\gamma s \begin{pmatrix} 1, \\ 36, \\ Reindex\gamma s\,(old\gamma s) \end{pmatrix}$$

which expands as:

$$Rewire\gamma s \begin{pmatrix} 1, \\ 36, \\ new\gamma s \end{pmatrix} = Rewire\gamma s \begin{pmatrix} 0, \\ 36, \\ \begin{pmatrix} ReWire\gamma pc \begin{pmatrix} 35, \\ 1, \\ [] \end{pmatrix}, new\gamma s \end{pmatrix} \end{pmatrix}$$

where the call to $Rewire\gamma pc$ results in the expansion of the first case (where $pc\_val > 0$):

$$\begin{aligned} Rewire\gamma pc \begin{pmatrix} 35, \\ 1, \\ new\gamma s \end{pmatrix} &= Rewire\gamma pc \begin{pmatrix} 34, \\ 1, \\ \begin{pmatrix} Build\gamma \begin{pmatrix} 1, \\ 1, \\ 35, \\ M, \end{pmatrix}, new\gamma s \end{pmatrix} \end{pmatrix} \\ &= Rewire\gamma pc \begin{pmatrix} 34, \\ 1, \\ ((\gamma_{35}(1,1) = M), new\gamma s) \end{pmatrix} \end{aligned}$$

The recursion in $Rewire\gamma pc$ will continue until the base case is reached, where $pc\_val = 0$, in which case the following definition is invoked:

$$Rewire\gamma pc \begin{pmatrix} 0, \\ 1, \\ old\gamma s, \\ new\gamma s \end{pmatrix} = \begin{pmatrix} Build\gamma \begin{pmatrix} 1, \\ 1, \\ 0, \\ M, \end{pmatrix}, new\gamma s \end{pmatrix}$$

Finally, the recursive call to *Rewire*γs where the module number is 0 is reached and in such a case the following definition is used:

$$Rewire\gamma s \begin{pmatrix} 0, \\ 36, \\ new\gamma s \end{pmatrix} = new\gamma s$$

Completing the *Rewire*γs operation completes the generation of γ-wiring functions for the concrete dSCA. The list below shows the γ-wiring functions produced for the GRCP solution:

$\gamma_0(1,0) = M,$	$\gamma_0(1,2) = M,$	$\gamma_0(1,3) = M,$	$\gamma_0(1,4) = M,$	$\gamma_0(1,1) = M,$
$\gamma_1(1,0) = M,$	$\gamma_1(1,2) = M,$	$\gamma_1(1,3) = M,$	$\gamma_1(1,4) = U,$	$\gamma_1(1,1) = M,$
$\gamma_2(1,0) = M,$	$\gamma_2(1,2) = U,$	$\gamma_2(1,3) = U,$	$\gamma_2(1,4) = U,$	$\gamma_2(1,1) = M,$
$\gamma_3(1,0) = M,$	$\gamma_3(1,2) = M,$	$\gamma_3(1,3) = M,$	$\gamma_3(1,4) = M,$	$\gamma_3(1,1) = M,$
$\gamma_4(1,0) = M,$	$\gamma_4(1,2) = M,$	$\gamma_4(1,3) = M,$	$\gamma_4(1,4) = U,$	$\gamma_4(1,1) = M,$
$\gamma_5(1,0) = M,$	$\gamma_5(1,2) = M,$	$\gamma_5(1,3) = M,$	$\gamma_5(1,4) = U,$	$\gamma_5(1,1) = M,$
$\gamma_6(1,0) = M,$	$\gamma_6(1,2) = M,$	$\gamma_6(1,3) = M,$	$\gamma_6(1,4) = U,$	$\gamma_6(1,1) = M,$
$\gamma_7(1,0) = M,$	$\gamma_7(1,2) = U,$	$\gamma_7(1,3) = U,$	$\gamma_7(1,4) = U,$	$\gamma_7(1,1) = M,$
$\gamma_8(1,0) = M,$	$\gamma_8(1,2) = U,$	$\gamma_8(1,3) = U,$	$\gamma_8(1,4) = U,$	$\gamma_8(1,1) = M,$
$\gamma_9(1,0) = M,$	$\gamma_9(1,2) = M,$	$\gamma_9(10,3) = M,$	$\gamma_9(1,4) = U,$	$\gamma_9(1,1) = M,$
$\gamma_{10}(1,0) = M,$	$\gamma_{10}(1,2) = S,$	$\gamma_{10}(1,3) = M,$	$\gamma_{10}(1,4) = U,$	$\gamma_{10}(1,1) = M,$
$\gamma_{11}(1,0) = M,$	$\gamma_{11}(1,2) = M$	$\gamma_{11}(1,3) = M,$	$\gamma_{11}(1,4) = U,$	$\gamma_{11}(1,1) = M,$
$\gamma_{12}(1,0) = M,$	$\gamma_{12}(1,2) = S,$	$\gamma_{12}(1,3) = M,$	$\gamma_{12}(1,4) = U,$	$\gamma_{12}(1,1) = M,$
$\gamma_{13}(1,0) = M,$	$\gamma_{13}(1,2) = M,$	$\gamma_{13}(1,3) = M,$	$\gamma_{13}(1,4) = U,$	$\gamma_{13}(1,1) = M,$
$\gamma_{14}(1,0) = M,$	$\gamma_{14}(1,2) = S,$	$\gamma_{14}(1,3) = M,$	$\gamma_{14}(1,4) = U,$	$\gamma_{14}(1,1) = M,$
$\gamma_{15}(1,0) = M,$	$\gamma_{15}(1,2) = U,$	$\gamma_{15}(1,3) = U,$	$\gamma_{15}(1,4) = U,$	$\gamma_{15}(1,1) = M,$
$\gamma_{16}(1,0) = M,$	$\gamma_{16}(1,2) = U,$	$\gamma_{16}(1,3) = U,$	$\gamma_{16}(1,4) = U,$	$\gamma_{16}(1,1) = M,$
$\gamma_{17}(1,0) = M,$	$\gamma_{17}(1,2) = U,$	$\gamma_{17}(1,3) = U,$	$\gamma_{17}(1,4) = U,$	$\gamma_{17}(1,1) = M,$
$\gamma_{18}(1,0) = M,$	$\gamma_{18}(1,2) = U,$	$\gamma_{18}(1,3) = U,$	$\gamma_{18}(1,4) = U,$	$\gamma_{18}(1,1) = M,$
$\gamma_{19}(1,0) = M,$	$\gamma_{19}(1,2) = U,$	$\gamma_{19}(1,3) = U,$	$\gamma_{19}(1,4) = U,$	$\gamma_{19}(1,1) = M,$
$\gamma_{20}(1,0) = M,$	$\gamma_{20}(1,2) = U,$	$\gamma_{20}(1,3) = U,$	$\gamma_{20}(1,4) = U,$	$\gamma_{20}(1,1) = M,$
$\gamma_{21}(1,0) = M,$	$\gamma_{21}(1,2) = M,$	$\gamma_{21}(1,3) = M,$	$\gamma_{21}(1,4) = U,$	$\gamma_{21}(1,1) = M,$
$\gamma_{22}(1,0) = M,$	$\gamma_{22}(1,2) = M,$	$\gamma_{22}(1,3) = M,$	$\gamma_{22}(1,4) = U,$	$\gamma_{22}(1,1) = M,$
$\gamma_{23}(1,0) = M,$	$\gamma_{23}(1,2) = M,$	$\gamma_{23}(1,3) = M,$	$\gamma_{23}(1,4) = U,$	$\gamma_{23}(1,1) = M,$
$\gamma_{24}(1,0) = M,$	$\gamma_{24}(1,2) = M,$	$\gamma_{24}(1,3) = M,$	$\gamma_{24}(1,4) = U,$	$\gamma_{24}(1,1) = M,$
$\gamma_{25}(1,0) = M,$	$\gamma_{25}(1,2) = M,$	$\gamma_{25}(1,3) = M,$	$\gamma_{25}(1,4) = U,$	$\gamma_{25}(1,1) = M,$
$\gamma_{26}(1,0) = M,$	$\gamma_{26}(1,2) = M,$	$\gamma_{26}(1,3) = M,$	$\gamma_{26}(1,4) = U,$	$\gamma_{26}(1,1) = M,$
$\gamma_{27}(1,0) = M,$	$\gamma_{27}(1,2) = M,$	$\gamma_{27}(1,3) = M,$	$\gamma_{27}(1,4) = U,$	$\gamma_{27}(1,1) = M,$

$\gamma_{28}(1,0) = M,$	$\gamma_{28}(1,2) = S,$	$\gamma_{28}(1,3) = S,$	$\gamma_{28}(1,4) = U,$	$\gamma_{28}(1,1) = M,$
$\gamma_{29}(1,0) = M,$	$\gamma_{29}(1,2) = U,$	$\gamma_{29}(1,3) = U,$	$\gamma_{29}(1,4) = U,$	$\gamma_{29}(1,1) = M,$
$\gamma_{30}(1,0) = M,$	$\gamma_{30}(1,2) = S,$	$\gamma_{30}(1,3) = S,$	$\gamma_{30}(1,4) = U,$	$\gamma_{30}(1,1) = M,$
$\gamma_{31}(1,0) = M,$	$\gamma_{31}(1,2) = U,$	$\gamma_{31}(1,3) = U,$	$\gamma_{31}(1,4) = U,$	$\gamma_{31}(1,1) = M,$
$\gamma_{32}(1,0) = M,$	$\gamma_{32}(1,2) = S,$	$\gamma_{32}(1,3) = S,$	$\gamma_{32}(1,4) = U,$	$\gamma_{32}(1,1) = M,$
$\gamma_{33}(1,0) = M,$	$\gamma_{33}(1,2) = U,$	$\gamma_{33}(1,3) = U,$	$\gamma_{33}(1,4) = U,$	$\gamma_{33}(1,1) = M,$
$\gamma_{34}(1,0) = M,$	$\gamma_{34}(1,2) = S,$	$\gamma_{34}(1,3) = S,$	$\gamma_{34}(1,4) = U,$	$\gamma_{34}(1,1) = M,$
$\gamma_{35}(1,0) = M,$	$\gamma_{35}(1,2) = U,$	$\gamma_{35}(1,3) = U,$	$\gamma_{35}(1,4) = U,$	$\gamma_{35}(1,1) = M$

where, for the program counter, where $0 \leq pc\_val \leq Max_N - 1$ the following γ-wiring functions are defined:

$$\gamma_{pc\_val}(pc, 0) = M$$

β-wiring Functions

The new β-wiring function for module 0 of the concrete dSCA are defined, for $0 \leq pc\_val \leq 35$, as:

$$\beta_{pc\_val}(pc, 0) = pc$$

and for module 1:

$\beta_0(1,0) = pc,$	$\beta_0(1,2) = 1,$	$\beta_0(1,3) = 1,$	$\beta_0(1,4) = 1,$	$\beta_0(1,1) = 1,$
$\beta_1(1,0) = pc,$	$\beta_1(1,2) = 1,$	$\beta_1(1,3) = 1,$	$\beta_1(1,4) = \omega,$	$\beta_1(1,1) = 1,$
$\beta_2(1,0) = pc,$	$\beta_2(1,2) = \omega,$	$\beta_2(1,3) = \omega,$	$\beta_2(1,4) = \omega,$	$\beta_2(1,1) = 1,$
$\beta_3(1,0) = pc,$	$\beta_3(1,2) = 1,$	$\beta_3(1,3) = 1,$	$\beta_3(1,4) = 1,$	$\beta_3(1,1) = 1,$
$\beta_4(1,0) = pc,$	$\beta_4(1,2) = 1,$	$\beta_4(1,3) = 1,$	$\beta_4(1,4) = \omega,$	$\beta_4(1,1) = 1,$
$\beta_5(1,0) = pc,$	$\beta_5(1,2) = 1,$	$\beta_5(1,3) = 1,$	$\beta_5(1,4) = \omega,$	$\beta_5(1,1) = 1,$
$\beta_6(1,0) = pc,$	$\beta_6(1,2) = 1,$	$\beta_6(1,3) = 1,$	$\beta_6(1,4) = \omega,$	$\beta_6(1,1) = 1,$
$\beta_7(1,0) = pc,$	$\beta_7(1,2) = \omega,$	$\beta_7(1,3) = \omega,$	$\beta_7(1,4) = \omega,$	$\beta_7(1,1) = 1,$
$\beta_8(1,0) = pc,$	$\beta_8(1,2) = \omega,$	$\beta_8(1,3) = \omega,$	$\beta_8(1,4) = \omega,$	$\beta_8(1,1) = 1,$
$\beta_9(1,0) = pc,$	$\beta_9(1,2) = 1,$	$\beta_9(1,3) = 1,$	$\beta_9(1,4) = \omega,$	$\beta_9(1,1) = 1,$
$\beta_{10}(1,0) = pc,$	$\beta_{10}(1,2) = 9,$	$\beta_{10}(1,3) = 1,$	$\beta_{10}(1,4) = \omega,$	$\beta_{10}(1,1) = 1,$
$\beta_{11}(1,0) = pc,$	$\beta_{11}(1,2) = 1,$	$\beta_{11}(1,3) = 1,$	$\beta_{11}(1,4) = \omega,$	$\beta_{11}(1,1) = 1,$
$\beta_{12}(1,0) = pc,$	$\beta_{12}(1,2) = 9,$	$\beta_{12}(1,3) = 1,$	$\beta_{12}(1,4) = \omega,$	$\beta_{12}(1,1) = 1,$
$\beta_{13}(1,0) = pc,$	$\beta_{13}(1,2) = 1,$	$\beta_{13}(1,3) = 1,$	$\beta_{13}(1,4) = \omega,$	$\beta_{13}(1,1) = 1,$
$\beta_{14}(1,0) = pc,$	$\beta_{14}(1,2) = 9,$	$\beta_{14}(1,3) = 1,$	$\beta_{14}(1,4) = \omega,$	$\beta_{14}(1,1) = 1,$
$\beta_{15}(1,0) = pc,$	$\beta_{15}(1,2) = \omega,$	$\beta_{15}(1,3) = \omega,$	$\beta_{15}(1,4) = \omega,$	$\beta_{15}(1,1) = 1,$

$\beta_{16}(1,0) = pc,$	$\beta_{16}(1,2) = \omega,$	$\beta_{16}(1,3) = \omega,$	$\beta_{16}(1,4) = \omega,$	$\beta_{16}(1,1) = 1,$
$\beta_{17}(1,0) = pc,$	$\beta_{17}(1,2) = \omega,$	$\beta_{17}(1,3) = \omega,$	$\beta_{17}(1,4) = \omega,$	$\beta_{17}(1,1) = 1,$
$\beta_{18}(1,0) = pc,$	$\beta_{18}(1,2) = \omega,$	$\beta_{18}(1,3) = \omega,$	$\beta_{18}(1,4) = \omega,$	$\beta_{18}(1,1) = 1,$
$\beta_{19}(1,0) = pc,$	$\beta_{19}(1,2) = \omega,$	$\beta_{19}(1,3) = \omega,$	$\beta_{19}(1,4) = \omega,$	$\beta_{19}(1,1) = 1,$
$\beta_{20}(1,0) = pc,$	$\beta_{20}(1,2) = \omega,$	$\beta_{20}(1,3) = \omega,$	$\beta_{20}(1,4) = \omega,$	$\beta_{20}(1,1) = 1,$
$\beta_{21}(1,0) = pc,$	$\beta_{21}(1,2) = 1,$	$\beta_{21}(1,3) = 1,$	$\beta_{21}(1,4) = \omega,$	$\beta_{21}(1,1) = 1,$
$\beta_{22}(1,0) = pc,$	$\beta_{22}(1,2) = 1,$	$\beta_{22}(1,3) = 1,$	$\beta_{22}(1,4) = \omega,$	$\beta_{22}(1,1) = 1,$
$\beta_{23}(1,0) = pc,$	$\beta_{23}(1,2) = 1,$	$\beta_{23}(1,3) = 1,$	$\beta_{23}(1,4) = \omega,$	$\beta_{23}(1,1) = 1,$
$\beta_{24}(1,0) = pc,$	$\beta_{24}(1,2) = 1,$	$\beta_{24}(1,3) = 1,$	$\beta_{24}(1,4) = \omega,$	$\beta_{24}(1,1) = 1,$
$\beta_{25}(1,0) = pc,$	$\beta_{25}(1,2) = 1,$	$\beta_{25}(1,3) = 1,$	$\beta_{25}(1,4) = \omega,$	$\beta_{25}(1,1) = 1,$
$\beta_{26}(1,0) = pc,$	$\beta_{26}(1,2) = 1,$	$\beta_{26}(1,3) = 1,$	$\beta_{26}(1,4) = \omega,$	$\beta_{26}(1,1) = 1,$
$\beta_{27}(1,0) = pc,$	$\beta_{27}(1,2) = 1,$	$\beta_{27}(1,3) = 1,$	$\beta_{27}(1,4) = \omega,$	$\beta_{27}(1,1) = 1,$
$\beta_{28}(1,0) = pc,$	$\beta_{28}(1,2) = 1,$	$\beta_{28}(1,3) = 2,$	$\beta_{28}(1,4) = \omega.$	$\beta_{28}(1,1) = 1,$
$\beta_{29}(1,0) = pc,$	$\beta_{29}(1,2) = \omega,$	$\beta_{29}(1,3) = \omega,$	$\beta_{29}(1,4) = \omega,$	$\beta_{29}(1,1) = 1,$
$\beta_{30}(1,0) = pc,$	$\beta_{30}(1,2) = 3,$	$\beta_{30}(1,3) = 4,$	$\beta_{30}(1,4) = \omega,$	$\beta_{30}(1,1) = 1,$
$\beta_{31}(1,0) = pc,$	$\beta_{31}(1,2) = \omega,$	$\beta_{31}(1,3) = \omega,$	$\beta_{31}(1,4) = \omega,$	$\beta_{31}(1,1) = 1,$
$\beta_{32}(1,0) = pc,$	$\beta_{32}(1,2) = 5,$	$\beta_{32}(1,3) = 6,$	$\beta_{32}(1,4) = \omega,$	$\beta_{32}(1,1) = 1,$
$\beta_{33}(1,0) = pc,$	$\beta_{33}(1,2) = \omega,$	$\beta_{33}(1,3) = \omega,$	$\beta_{33}(1,4) = \omega,$	$\beta_{33}(1,1) = 1,$
$\beta_{34}(1,0) = pc,$	$\beta_{34}(1,2) = 7,$	$\beta_{34}(1,3) = 8,$	$\beta_{34}(1,4) = \omega,$	$\beta_{34}(1,1) = 1,$
$\beta_{35}(1,0) = pc,$	$\beta_{35}(1,2) = \omega,$	$\beta_{35}(1,3) = \omega,$	$\beta_{35}(1,4) = \omega,$	$\beta_{35}(1,1) = 1$

Delay Functions

Creating the delay functions for the concrete dSCA is performed by the *Createδs* operation:

$$Create\delta s\left(\ Source\_SCA\ \right) = B\delta s\left(\begin{array}{l} Create\gamma s(Source\_SCA), \\ [] \end{array}\right)$$

which calls the $B\delta s$ operation with the first argument being the list returned from the generation of the concrete dSCAs γ-wiring functions (as shown above) and an empty list for the new delay functions. The call expands as follows since there is more than one element in the list of wiring functions:

$$B\delta s\left(\begin{array}{l} (e, es), \\ [], \end{array}\right) = B\delta s\left(\begin{array}{l} es, \\ \left(\begin{array}{l} Build\delta\left(\begin{array}{l} GetArg(RetTerm(e,1),1), \\ GetArg(RetTerm(e,1),2), \\ GetIndex(RetTerm(e,1),1), \\ t-1 \end{array}\right), [] \end{array}\right) \end{array}\right)$$

The first element in the γ-wiring function list, produced above, is:

$$\gamma_0(1,0) = M$$

The call to $B\delta s$ will progress in the following manner:

$$\begin{aligned}
B\delta s\left(\begin{array}{l}(e,es),\\ [],\end{array}\right) &= B\delta s\left(\begin{array}{l} es,\\ \left(\begin{array}{l} Build\delta\left(\begin{array}{l} GetArg(\gamma_0(1,0),1),\\ GetArg(\gamma_0(1,0),2),\\ GetIndex(\gamma_0(1,0),1),\\ t-1\end{array}\right),[]\end{array}\right)\end{array}\right)\\
&= B\delta s\left(\begin{array}{l} es,\\ \left(\begin{array}{l} Build\delta\left(\begin{array}{l} 1,\\ 0,\\ 0,\\ t-1\end{array}\right),[]\end{array}\right)\end{array}\right)\\
&= B\delta s\left(\begin{array}{l} es,\\ (\delta_{1,0,0}(t,a,x) = t-1),[]\end{array}\right)
\end{aligned}$$

The recursion will complete when there is only one element left in the list of γ-wiring functions, in which case the following call to $B\delta s$ is used:

$$B\delta s\left(\begin{array}{l} e,\\ neqs,\end{array}\right) = Build\delta\left(\begin{array}{l} GetArg(RetTerm(e,1),1),\\ GetArg(RetTerm(e,1),2),\\ GetIndex(RetTerm(e,1),1),\\ t-1\end{array}\right)$$

For this example, it can be seen from the γ-wiring function transformation above that the last element in the list is:

$$\gamma_{35}(pc,0) = M$$

which under the defined transformation will produce the delay function:

$$\delta_{pc,0,35}(t,a,x) = t-1$$

The complete list of delay functions can be seen in Appendix E.

Initial State Equations

Construction of the new Initial State equations commences with a call to the $CreateIVs$ operation:

$$CreateIVs\left(\; Source\_SCA \; \right) = BIVs \left(\begin{array}{l} num\_modules(Source\_SCA), \\ GetMaxN(Source\_SCA), \\ GetIV(Source\_SCA), \\ [] \end{array} \right)$$

where the call to $BIVs$ is expanded as the recursive call:

$$BIVs \left(\begin{array}{l} 1, \\ 36, \\ oeqs, \\ neqs \end{array} \right) = BIVs \left(\begin{array}{l} 0, \\ 36, \\ oeqs, \\ \left(BIV \left(\begin{array}{l} 0, \\ 35, \\ 1, \\ oeqs, \\ [] \end{array} \right), neqs \right) \end{array} \right)$$

Observe that the call to BIV is in this case a recursive call to itself, with the value of the first argument incremented by one:

$$BIV \left(\begin{array}{l} 0, \\ 35, \\ 1, \\ oeqs, \\ neqs \end{array} \right) = BIV \left(\begin{array}{l} 1, \\ 35, \\ 1, \\ oeqs, \\ \left(GenIVs \left(\begin{array}{l} 1, \\ 0, \\ 35, \\ oeqs \end{array} \right), neqs \right) \end{array} \right)$$

The result of the $GenIV$ operation will be the following Initial State equation:

$$V_1(0,a,x) = \begin{pmatrix} stay,u,u,u,u,u,u,u,u, \\ u,u,u,u,u,u,u,u,u, \\ u,u,u,u,u,u,u,u,u, \\ u,u,u,u,u,u,u,u,u \end{pmatrix}$$

The recursive nature of $BIVs$ is such that the result above is repeated until we produce the Initial State equation for module 1 at time $t = 34$:

$$V_1(34,a,x) = \begin{pmatrix} stay,u,u,u,u,u,u,u,u, \\ u,u,u,u,u,u,u,u,u, \\ u,u,u,u,u,u,u,u,u, \\ u,u,u,u,u,u,u,u,u \end{pmatrix}$$

The next call to $BIVs$ is to the non-recursive version:

$$BIV\begin{pmatrix} 35, \\ 35, \\ 1, \\ oeqs, \\ neqs, \\ \Xi^{-1} \end{pmatrix} = \begin{pmatrix} BuildIV\begin{pmatrix} 1, \\ 35, \\ InitState\begin{pmatrix} 35, \\ 1, \\ oeqs, \\ [], \\ \Xi^{-1} \end{pmatrix} \end{pmatrix}, neqs \end{pmatrix}$$

which results in a call to the *InitState* operation, which is itself defined recursively, and in this case results in:

$$
\begin{aligned}
InitState\begin{pmatrix} 35, \\ 1, \\ oeqs, \\ [], \\ \Xi^{-1} \end{pmatrix} &= InitState\begin{pmatrix} 34, \\ 1, \\ oeqs, \\ \left(RetTerm\left(GetEl\begin{pmatrix} oeqs, \\ 1, \\ 35 \end{pmatrix}, 2\right), []\right), \\ \Xi^{-1} \end{pmatrix} \\
&= InitState\begin{pmatrix} 34, \\ 1, \\ oeqs, \\ (RetTerm\left(V_1(35, a, x) = 0, 2\right), []), \\ \Xi^{-1} \end{pmatrix} \\
&= InitState\begin{pmatrix} 34, \\ 1, \\ oeqs, \\ (0, []), \\ \Xi^{-1} \end{pmatrix}
\end{aligned}
$$

the next recursive call will look like:

$$
InitState\begin{pmatrix} 35, \\ 1, \\ oeqs, \\ [], \end{pmatrix} = InitState\begin{pmatrix} 33, \\ 1, \\ oeqs, \\ (0, 0, []), \end{pmatrix}
$$

This recursion continues until the value for the first argument reaches 0, in which case the following result is produced:

$$InitState\begin{pmatrix} 0, \\ 1, \\ oeqs, \\ nlist, \end{pmatrix} = \begin{pmatrix} stay, & true, & stay, & up, & true, & false, & false, \\ down, & up, & true, & true, & false, & false, & false, \\ true, & false, & 90, & true, & 0, & true, & 0, \\ false, & false, & false, & false, & false, & false, & false, \\ 0, & 0, & 0, & 0, & 0, & 0, & 0, \\ 0 & & & & & & \end{pmatrix}$$

This result from $InitState$ is subsequently used to construction the Initial State equation at time $t = Max_N - 1 = 35$:

$$V_1(35, a, x) = \begin{pmatrix} stay, & true, & stay, & up, & true, & false, & false, \\ down, & up, & true, & true, & false, & false, & false, \\ true, & false, & 90, & true, & 0, & true, & 0, \\ false, & false, & false, & false, & false, & false, & false, \\ 0, & 0, & 0, & 0, & 0, & 0, & 0, \\ 0 & & & & & & \end{pmatrix}$$

Next the case where $BIVs$ is called with the module number equal to 0. In this case the base case of $BIVs$ is invoked as:

$$BIVs\begin{pmatrix} 0, \\ 36, \\ oeqs, \\ neqs, \end{pmatrix} = \begin{pmatrix} BIVpc\begin{pmatrix} 35, \\ [], \\ 35, \end{pmatrix}, neqs \end{pmatrix}$$

where $BIVpc$ builds the following list of Initial State equations:

$$\begin{aligned} &V_0(35, a, x) = 0, \\ &V_0(34, a, x) = 35, \\ &\vdots \\ &V_0(1, a, x) = 2, \\ &V_0(0, a, x) = 1, \end{aligned}$$

The complete list of Initial State equations can be seen in the algebraic specification of the concrete dSCA at Appendix E.

State Transition Equations

To generate the State Transition equations the *CreateST* operation is called with the source SCA as an argument:

$$CreateSTs\left(\begin{array}{l} Source\_SCA, \\ \Upsilon, \\ \Pi \end{array}\right) = BSTs\left(\begin{array}{l} GetEqSTVF(Source\_SCA), \\ [], \\ Createds(Source\_SCA), \\ Create\beta s(Source\_SCA), \\ Create\delta s(Source\_SCA), \\ GetMaxN(Source\_SCA), \\ \Upsilon, \\ \Pi \end{array}\right)$$

All of the arguments are either extracted from the source specification, e.g. extracting the previous delay functions, or are created using elements of this transformation, e.g. the creation of new wiring functions. Arguments that have to be created have already been shown in this chapter, with the exception of *Createds*, which is now shown.

Generation of the d function for the concrete dSCA network commences with a call to the *Createds* operation:

$$createds\left(\ Source\_SCA\ \right) = Bds\left(\begin{array}{l} GetNumModules(Source\_SCA), \\ GetMaxN(Source\_SCA), \\ GetMaxA(Source\_SCA) + 1, \\ Get\delta Eqs(Source_SCA), \\ [] \end{array}\right)$$

which for the source abstract dSCA under consideration, with a defining shape of $(1, 35)$ and maximum number of arguments of 4, can be written:

$$createds\left(\; Source\_SCA \; \right) = Bds \left(\begin{array}{l} GetNumModules(Source\_SCA), \\ GetMaxN(Source\_SCA), \\ GetMaxA(Source\_SCA) + 1, \\ Get\delta Eqs(Source_SCA), \\ [] \end{array} \right)$$

$$= Bds \left(\begin{array}{l} 1, \\ 36, \\ 5, \\ Get\delta Eqs(Source_SCA), \\ [] \end{array} \right)$$

The call to Bds expands into the recursive call:

$$Bds \left(\begin{array}{l} 1, \\ 36, \\ 5, \\ oldeqs, \\ [] \end{array} \right) = Bds \left(\begin{array}{l} 0, \\ 36, \\ 5, \\ oldeqs, \\ \left(Bdspc \left(\begin{array}{l} 35, \\ 1, \\ 5, \\ 36, \\ oldeqs, \\ [] \end{array} \right), [] \right) \end{array} \right)$$

Expanding the call to *Bdspc* results in the recursive call:

$$Bdspc\begin{pmatrix} 35, \\ 1, \\ 5, \\ 36, \\ oldeqs, \\ [] \end{pmatrix} = Bdspc\begin{pmatrix} 34, \\ 1, \\ 5, \\ 36, \\ oldeqs, \\ \begin{pmatrix} Bdsarg\begin{pmatrix} 4, \\ 35, \\ 1, \\ 35, \\ oldeqs, \\ [] \end{pmatrix}, [] \end{pmatrix} \end{pmatrix}$$

Now, the call to the *Bdsarg* operation expands to the recursive call of:

$$Bdsarg\begin{pmatrix} 4, \\ 1, \\ 35, \\ 36, \\ oldeqs, \\ [] \end{pmatrix} = Bdsarg\begin{pmatrix} 3, \\ 1, \\ 35, \\ 36, \\ oldeqs, \\ \begin{pmatrix} Buildd\begin{pmatrix} 1, \\ 4, \\ 35, \\ 35, \\ d\_val, \end{pmatrix}, [] \end{pmatrix} \end{pmatrix}$$

with:

$$
\begin{aligned}
d\_val &= (Max_N + pc\_val) - \left(t - RetTerm \left(GetEl \left(\begin{array}{l} oldeqs, \\ 1, \\ 3, \\ 35, \end{array} \right), 2 \right) \right) \\
&= (36 + 35) - (t - RetTerm\,(\delta_{1,3,35} = t - 1)\,, 2) \\
&= (71) - (t - (t - 1)) \\
&= 71 - (t - t + 1) \\
&= 71 - 1 \\
&= 70
\end{aligned}
$$

thus:

$$
\begin{aligned}
Bdsarg \left(\begin{array}{l} 4, \\ 1, \\ 35, \\ 36, \\ oldeqs, \\ [] \end{array} \right) &= Bdsarg \left(\begin{array}{l} 3, \\ 1, \\ 35, \\ 36, \\ oldeqs, \\ \left(Buildd \left(\begin{array}{l} 1, \\ 4, \\ 35, \\ 35, \\ 70, \end{array} \right), [] \right) \end{array} \right) \\
&= Bdsarg \left(\begin{array}{l} 3, \\ 1, \\ 35, \\ 36, \\ oldeqs, \\ \left(d^{35}_{1,4,35} = 70, [] \right) \end{array} \right)
\end{aligned}
$$

Values are of limited interest until we arrive at program counter values of 27 since all values of the delay function up to this point are the unit delay. For program

counter 27 the 5th argument is unwired, thus of unit delay, however the recursive call to $Bdsarg$ for the 4^{th} argument, is as follows:

$$Bdsarg\begin{pmatrix}3,\\1,\\27,\\36,\\oldeqs,\\neqs\end{pmatrix} = Bdsarg\begin{pmatrix}2,\\1,\\27,\\36,\\oldeqs,\\\begin{pmatrix}Buildd\begin{pmatrix}1,\\3,\\27,\\35,\\d\_val,\end{pmatrix}, d^{35}_{1,4,27} = 62, neqs\end{pmatrix}\end{pmatrix}$$

and d_val is calculated thus:

$$\begin{aligned} d\_val &= (Max_N + pc\_val) - \left(t - RetTerm\left(GetEl\begin{pmatrix}oldeqs,\\1,\\2,\\27,\end{pmatrix}, 2\right)\right)\\ &= (36 + 27) - (t - RetTerm\left(\delta_{1,2,27} = t - 28\right), 2)\\ &= (63) - (t - (t - 28))\\ &= 63 - (t - t + 28)\\ &= 63 - 28\\ &= 35 \end{aligned}$$

and therefore, *Bdsarg* can be traced as follows:

$$Bdsarg\begin{pmatrix}3,\\1,\\27,\\36,\\oldeqs,\\neqs\end{pmatrix} = Bdsarg\begin{pmatrix}2,\\1,\\27,\\36,\\oldeqs,\\\begin{pmatrix}Buildd\begin{pmatrix}1,\\3,\\27,\\35,\\35,\end{pmatrix}, d^{35}_{1,4,27}=62, neqs\end{pmatrix}\end{pmatrix}$$

$$= Bdsarg\begin{pmatrix}2,\\1,\\27,\\36,\\oldeqs,\\\left(d^{35}_{1,3,27}=35, d^{35}_{1,4,27}=62, neqs\right)\end{pmatrix}$$

Similarly the recursive call with the argument value of 2 will result in the recursive call:

$$Bdsarg\begin{pmatrix}1,\\1,\\27,\\36,\\oldeqs,\\\begin{pmatrix}d^{35}_{1,2,27}=34,\\d^{35}_{1,3,27}=35,\\d^{35}_{1,4,27}=62,\\neqs\end{pmatrix}\end{pmatrix}$$

At this point, the recursive call to *Bdsarg* is made, which simply returns the list of new d equations:

$$Bdsarg\left(\begin{array}{l} 1, \\ 1, \\ 27, \\ 36, \\ oldeqs, \\ \left(\begin{array}{l} d^{35}_{1,2,27} = 34, \\ d^{35}_{1,3,27} = 35, \\ d^{35}_{1,4,27} = 62, \\ neqs \end{array}\right) \end{array}\right) = \left(\begin{array}{l} d^{35}_{1,2,27} = 34, \\ d^{35}_{1,3,27} = 35, \\ d^{35}_{1,4,27} = 62, \\ neqs \end{array}\right)$$

The next step will be to recurse on the next value of the program counter, which will continue until the following base case is invoked:

$$Bdspc\left(\begin{array}{l} 0, \\ 1, \\ 5, \\ 36, \\ oldeqs, \\ neqs \end{array}\right) = \left(Bdsarg\left(\begin{array}{l} 5, \\ 0, \\ 1, \\ 36, \\ oldeqs, \\ [] \end{array}\right), neqs\right)$$

which would result in:

$$\left(d^{35}_{1,2,0} = 1, d^{35}_{1,3,0} = 2, d^{35}_{1,4,0} = 3, neqs \right)$$

at this point the generation of d-values is now finished by recursively calling the *Bds* operation, using its base case:

$$Bds\left(\begin{array}{l} 0, \\ 36, \\ 5, \\ oldeqs, \\ neqs \end{array}\right) = neqs$$

The values of d which are interesting, i.e. are not of unit delay, are:

$d^{35}_{1,2,0} = 1,$	$d^{35}_{1,3,3} = 7,$	$d^{35}_{1,2,6} = 13,$	$d^{35}_{1,3,11} = 17,$	$d^{35}_{1,3,21} = 23,$	$d^{35}_{1,3,25} = 31$
$d^{35}_{1,3,0} = 2,$	$d^{35}_{1,4,3} = 8,$	$d^{35}_{1,3,6} = 14,$	$d^{35}_{1,3,12} = 18,$	$d^{35}_{1,2,22} = 24,$	$d^{35}_{1,2,26} = 32$
$d^{35}_{1,4,0} = 3,$	$d^{35}_{1,2,4} = 9,$	$d^{35}_{1,3,9} = 15,$	$d^{35}_{1,2,13} = 21,$	$d^{35}_{1,3,22} = 25,$	$d^{35}_{1,3,26} = 33$
$d^{35}_{1,2,1} = 4,$	$d^{35}_{1,3,4} = 10,$	$d^{35}_{1,2,9} = 21,$	$d^{35}_{1,3,13} = 19,$	$d^{35}_{1,2,23} = 26,$	$d^{35}_{1,2,27} = 34$
$d^{35}_{1,3,1} = 5,$	$d^{35}_{1,2,5} = 11,$	$d^{35}_{1,4,10} = 16,$	$d^{35}_{1,3,14} = 20,$	$d^{35}_{1,3,23} = 27,$	$d^{35}_{1,3,27} = 35$
$d^{35}_{1,2,3} = 6,$	$d^{35}_{1,3,5} = 12,$	$d^{35}_{1,2,11} = 21,$	$d^{35}_{1,2,21} = 22,$	$d^{35}_{1,2,24} = 28,$	$d^{35}_{1,3,24} = 29$
					$d^{35}_{1,2,25} = 30$

Other calculated values, which will not be used in the eventual concrete dSCA due to them being wired to inputs or the special ω module are:

$d^{35}_{1,4,1} = 36$	$d^{35}_{1,2,10} = 45$	$d^{35}_{1,2,17} = 52$	$d^{35}_{1,4,23} = 58$	$d^{35}_{1,2,31} = 66$
$d^{35}_{1,2,2} = 37$	$d^{35}_{1,4,10} = 45$	$d^{35}_{1,3,17} = 52$	$d^{35}_{1,4,24} = 59$	$d^{35}_{1,3,31} = 66$
$d^{35}_{1,3,2} = 37$	$d^{35}_{1,4,11} = 46$	$d^{35}_{1,4,17} = 52$	$d^{35}_{1,4,25} = 60$	$d^{35}_{1,4,31} = 66$
$d^{35}_{1,4,2} = 37$	$d^{35}_{1,2,12} = 47$	$d^{35}_{1,2,18} = 53$	$d^{35}_{1,4,26} = 61$	$d^{35}_{1,2,32} = 67$
$d^{35}_{1,4,4} = 39$	$d^{35}_{1,4,12} = 47$	$d^{35}_{1,3,18} = 53$	$d^{35}_{1,4,27} = 62$	$d^{35}_{1,3,32} = 67$
$d^{35}_{1,4,5} = 40$	$d^{35}_{1,4,13} = 48$	$d^{35}_{1,4,18} = 53$	$d^{35}_{1,2,28} = 63$	$d^{35}_{1,4,32} = 67$
$d^{35}_{1,4,6} = 41$	$d^{35}_{1,2,14} = 49$	$d^{35}_{1,2,19} = 54$	$d^{35}_{1,3,28} = 63$	$d^{35}_{1,2,33} = 68$
$d^{35}_{1,2,7} = 42$	$d^{35}_{1,4,14} = 49$	$d^{35}_{1,3,19} = 54$	$d^{35}_{1,4,28} = 63$	$d^{35}_{1,3,33} = 68$
$d^{35}_{1,3,7} = 42$	$d^{35}_{1,2,15} = 50$	$d^{35}_{1,4,19} = 54$	$d^{35}_{1,2,29} = 64$	$d^{35}_{1,4,33} = 68$
$d^{35}_{1,4,7} = 42$	$d^{35}_{1,3,15} = 50$	$d^{35}_{1,2,20} = 55$	$d^{35}_{1,3,29} = 64$	$d^{35}_{1,2,34} = 69$
$d^{35}_{1,2,8} = 43$	$d^{35}_{1,4,15} = 50$	$d^{35}_{1,3,20} = 55$	$d^{35}_{1,4,29} = 64$	$d^{35}_{1,3,34} = 69$
$d^{35}_{1,3,8} = 43$	$d^{35}_{1,3,16} = 51$	$d^{35}_{1,4,20} = 55$	$d^{35}_{1,2,30} = 65$	$d^{35}_{1,4,34} = 69$
$d^{35}_{1,4,8} = 43$	$d^{35}_{1,2,16} = 51$	$d^{35}_{1,4,21} = 56$	$d^{35}_{1,3,30} = 65$	$d^{35}_{1,2,35} = 70$
$d^{35}_{1,4,9} = 44$	$d^{35}_{1,4,16} = 51$	$d^{35}_{1,4,22} = 57$	$d^{35}_{1,4,30} = 65$	$d^{35}_{1,3,35} = 70$
				$d^{35}_{1,4,35} = 70$

The process of generating the State Transition equations can now continue. Recall that the initial call would be:

$$CreateSTs\begin{pmatrix} Source\_SCA, \\ \Upsilon, \\ \Pi \end{pmatrix} = BSTs\begin{pmatrix} GetEqSTVF(Source\_SCA), \\ [], \\ Createds(Source\_SCA), \\ Create\beta s(Source\_SCA), \\ Create\delta s(Source\_SCA), \\ GetMaxN(Source\_SCA), \\ \Upsilon, \\ \Pi \end{pmatrix}$$

In the GRCP example there are two modules, thus there are two equations in the list of State Transition equations from the source abstract dSCA. The expansion of the call to $BSTs$ is given as:

$$BSTs\left(\begin{array}{l}(e, eqs),\\ neqs,\\ newds,\\ new\beta s,\\ new\delta s,\\ 36,\\ \Upsilon,\\ \Pi\end{array}\right) = BSTs\left(\begin{array}{l}eqs,\\ \left(\begin{array}{ll}BSTck & \left(\begin{array}{l}e,\\ GetIndex(RetTerm(e,1)),\\ 0,\\ Max_N,\\ newds,\\ new\beta s,\\ new\delta s,\\ \Upsilon,\\ \Pi\end{array}\right)\end{array}, neqs\right),\\ newds,\\ new\beta s,\\ new\delta s,\\ 36,\\ \Upsilon,\\ \Pi\end{array}\right)$$

Considering module 1, then the call to $BSTck$ will result in the following call to the BST operation, since the module is not m_0:

$$BST\left(\begin{array}{l} cond\left(\begin{array}{l} V_{pc}(t,a,x)=0,\\ b,\\ c\end{array}\right),\\ 1,\\ 0,\\ 36,\\ newds,\\ new\beta s,\\ new\delta s,\\ \Upsilon,\\ \Pi\end{array}\right) = cond\left(\begin{array}{l} pc\_rewire\left(V_{pc}(t,a,x)=0\right),\\ cs\_rewire\left(\begin{array}{l} b,\\ 1,0,36,\\ newds,\beta s,\delta s,\Upsilon,\Pi\end{array}\right),\\ BST\left(\begin{array}{l} c,\\ 1,1,36,\\ newds,new\beta s,new\delta s,\Upsilon,\Pi\end{array}\right)\end{array}\right)$$

Each of the parts on the right hand side will be expanded as:

$$\begin{aligned} pc\_rewire\left(V_{pc}(t,a,x)=0\right) &= (V_{pc}(t,a,x) = RetTerm(V_{pc}(t,a,x)=0,2))\\ &= (V_{pc}(t,a,x)=0)\end{aligned}$$

$$cs\_rewire\left(\begin{array}{l} cond\left(\begin{array}{l} V_1(t-35,a,x),\\ V_1(t-34,a,x),\\ V_1(t-33,a,x)\end{array}\right),\\ 1,0,36,\\ newds,\beta s,\delta s,\Upsilon,\Pi\end{array}\right) = \Upsilon\left(\begin{array}{l} V_{pc}(t,a,x),\\ V_1(t,a,x),\\ rw\left(\begin{array}{l} cond\left(\begin{array}{l} V_1(t-35,a,x),\\ V_1(t-34,a,x),\\ V_1(t-33,a,x)\end{array}\right),\\ 1,\\ 0,\\ 36,\\ ds,\\ \beta s,\\ \delta s,\\ \Pi\end{array}\right)\end{array}\right)$$

with the rw operation expanded as:

$$rw\left(\begin{array}{l} cond\left(\begin{array}{l} V_1(t-35,a,x), \\ V_1(t-34,a,x), \\ V_1(t-33,a,x) \end{array}\right), \\ 1, \\ 0, \\ 36, \\ ds, \\ \beta s, \\ \delta s, \\ \Pi \end{array}\right) = cond\left(\begin{array}{l} wire\left(\begin{array}{l} V_1(t-35,a,x), \\ \beta s, \delta s, ds, \\ 1,2,0, \\ 36,\Pi \end{array}\right), \\ wire\left(\begin{array}{l} V_1(t-34,a,x), \\ \beta s, \delta s, ds, \\ 1,3,0, \\ 36,\Pi \end{array}\right), \\ wire\left(\begin{array}{l} V_1(t-33,a,x), \\ \beta s, \delta s, ds, \\ 1,4,0, \\ 36,\Pi \end{array}\right), \end{array}\right)$$

As an example, the first $wire$ call expands as:

$$wire\left(\begin{array}{l} V_1(t-35,a,x), \\ \beta s, \delta s, ds, \\ 1,2,0, \\ 36,\Pi \end{array}\right) = \Pi_{prj\_val}^{36-1}(V_{new\_index}(new\_time,a,x))$$

with:

$$\begin{aligned} prj\_val &= RetTerm\left(GetEl\left(\begin{array}{l} ds, \\ 1, \\ 2, \\ 0 \end{array}\right), 2\right) \\ &= RetTerm\left(ds_{1,2,0} = 1, 2\right) \\ &= 1 \end{aligned}$$

$$
\begin{aligned}
new\_index &= RetTerm\left(GetEl\left(\begin{array}{l}\beta s,\\ 1,\\ 2,\\ 0\end{array}\right),2\right)\\
&= RetTerm\left(\beta_0(1,2)=1,2\right)\\
&= 1
\end{aligned}
$$

$$
\begin{aligned}
new\_time &= RetTerm\left(GetEl\left(\begin{array}{l}\delta s,\\ 1,\\ 2,\\ 0\end{array}\right),2\right)+1\\
&= RetTerm\left(\delta_{1,2,0}(t,a,x)=t-1,2\right)+1\\
&= t-1+1\\
&= t
\end{aligned}
$$

therefore the first call to the *wire* operation is:

$$
wire\left(\begin{array}{l}V_1(t-35,a,x),\\ \beta s,\delta s,ds,\\ 1,2,0,\\ 36,\Pi\end{array}\right)=\Pi_1^{35}(V_1(t,a,x))
$$

and rw therefore becomes:

$$
rw\left(\begin{array}{l}cond\left(\begin{array}{l}V_1(t-35,a,x),\\ V_1(t-34,a,x),\\ V_1(t-33,a,x)\end{array}\right),\\ 1,\\ 0,\\ 36,\\ ds,\\ \beta s,\\ \delta s,\\ \Pi\end{array}\right)=cond\left(\begin{array}{l}\Pi_1^{35}(V_1(t,a,x)),\\ \Pi_2^{35}(V_1(t,a,x)),\\ \Pi_3^{35}(V_1(t,a,x))\end{array}\right)
$$

$$BST\left(\begin{array}{l} cond\left(\begin{array}{l} V_{pc}(t,a,x)=0,\\ b,\\ c \end{array}\right),\\ 1,\\ 0,\\ 36,\\ newds,\\ new\beta s,\\ new\delta s,\\ \Upsilon,\\ \Pi \end{array}\right) = cond\left(\begin{array}{l} V_{pc}(t,a,x)=0,\\ \Upsilon\left(\begin{array}{l} V_{pc}(t,a,x),\\ V_1(t,a,x),\\ cond\left(\begin{array}{l} \Pi_1^{35}(V_1(t,a,x)),\\ \Pi_2^{35}(V_1(t,a,x)),\\ \Pi_3^{35}(V_1(t,a,x)) \end{array}\right) \end{array}\right),\\ BST\left(\begin{array}{l} c,\\ 1,1,36,\\ newds, new\beta s, new\delta s, \end{array}\right) \end{array}\right)$$

The expansion of BST continues recursively until the complete function for module 1 is produced (this can be seen in Appendix E).

If the call to $BSTck$ is considering module 0, then $BSTck$ returns the original definition as the program counter definition does not change. In the GCRP example, the result of $BSTck$ for module 0 will be:

$$V_{pc}(t+1,a,x) = \begin{cases} mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 0 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 1 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 2 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 3 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 4 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 5 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 6 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 7 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 8 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 9 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 10 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 11 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 12 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 13 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 14 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 15 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 16 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 17 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 18 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 19 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 20 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 21 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 22 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 23 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 24 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 25 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 26 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 27 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 28 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 29 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 30 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 31 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 32 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 33 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 34 \\ mod\,(add\,(V_{pc}(t,a,x),1),36) & \text{if } V_{pc}(t-1,a,x) = 35 \end{cases}$$

Recursive calls to $BSTs$ continue until the base recursive call to $BSTs$ is made, which results in the base case call to just $BSTck$ with the appropriate functionality from above selected depending upon the module number under consideration.

For brevity we do not reproduce the transformed State Transition equations for other modules here, instead, they are defined in the concrete dSCA specification given in Appendix E.

12.4 Correctness of concrete dSCA Example

The generated target concrete dSCA created from transforming the abstract dSCA in chapter 11 can be seen to be the same as the concrete dSCA given in Chapter 8.4 - the discussion of correctness given in that chapter is therefore still valid.

Additionally, this concrete dSCA and the previous abstract dSCA could exist in a hierarchy, and this will be demonstrated by introducing mappings for:

- spaces;
- clocks;
- global states; and
- input streams.

Spaces

Spaces (modules) in the two networks are equivalent, for modules m_i where $i \in \mathbb{N}_k^{dSCA2}$ thus it is appropriate to define the respacing operation $\pi : I_{N_{dSCA2}} \rightarrow I_{N_{dSCA1}}$ as:

$$\pi(i) = i$$

Clocks

There is no timing abstraction between networks, thus the retiming $\lambda : T^{dSCA2} \rightarrow T^{dSCA1}$, where for $t \in T^{dSCA2}$, can be appropriately defined as:

$$\lambda(t) = t$$

Global States

The set of sensible obvservable states for this SCA is the output of module 1 at regular intervals, given by the retiming. Additionally, the appropriate value from the tuple must be projected, and in this case it will be element at the value of the program counter. The global state of an SCA at time $t \in T$ is defined as the set of values held by all the observed channels at time $t \in T$, We introduce a global state abstraction mapping, ϕ of the form:

$$\phi_{obs} : A_{dSCA2}^{Ch_{dSCA2}} \rightarrow A_{dSCA1}^{Ch_{dSCA1}}$$

which is defined to project out the relevant element of module 1.

Input Streams

Input streams for the source and target dSCAs also exhibit a one-to-one mapping, and we therefore construct the stream abstraction function:

$$\theta : [T_{dSCA2} \rightarrow M_{A_{dSCA2}}]^{In_{dSCA2}} \rightarrow [T_{dSCA1} \rightarrow M_{A_{dSCA1}}]^{In_{dSCA1}}$$

as:

$$\theta(a_i(t)) = a_i(\lambda(t))$$

where $\lambda(t) = t$, thus

$$\theta(a_i(t)) = a_i(t)$$

Conjecture It is believed that the following diagram commutes:

$$\begin{array}{ccccccc}
T_{dSCA1} & \times [T_{dSCA1} \rightarrow M_{A_{dSCA1}}]^{In_{dSCA1}} \times & M_{A_{dSCA1}}^{Ch_{dSCA1}} & \xrightarrow{V_{dSCA1}} & M_{A_{dSCA1}}^{Ch_{dSCA1}} \\
\uparrow \lambda & & \uparrow \theta & & \uparrow \phi & & \uparrow \phi \\
Start_{\lambda} & \times [T_{dSCA2} \rightarrow M_{A_{dSCA2}}]^{In_{dSCA2}} \times & M_{A_{tup_{dSCA2}}}^{Ch_{dSCA2}} & \xrightarrow{V_{dSCA2}} & M_{A_{tup_{dSCA2}}}^{Ch_{dSCA2}}
\end{array}$$

12.5 Concluding Comments

This chapter has demonstrated the techniques required for mapping an abstract dSCA to a concrete dSCA with the same defining shape (and same type of cycle consistency). The techniques have been demonstrated by taking the Form 2 abstract dSCA solution to the GRC Problem and generating the appropriate concrete dSCA solution.

12.6 Sources

This transformation is all my own work.

Chapter 13

Summary and Future Work

This thesis set out to investigate whether a method could be developed to support the transformation of an algorithm described as a Synchronous Concurrent Algorithm to its implementation on a piece of hardware, also described as a SCA. Through the investigations it has been determined that this is the case, and that if a small number of syntactic extensions are made to the standard model of SCAs then a concise set of models can be produced that ease the understanding of such transformations. The benefits of restricting the extensions to syntactic ones are that the dicsussion never moves away from the well-founded notion of SCAs, and hence the work done on such algorithms is still valid in the new models.

To summarise, this work has introduced abstract and concrete dynamic SCAs and has demonstrated that there exist algebraic transformations that allow an algorithm described as an SCA to be transformed into an an implementation on hardware that is described as an SCA.

It has also been identified that the simplest and most compact transformations will take place with algorithms that can be described as cycle consistent abstract dSCAs.

The implementation of the models and techniques has been demonstrated by applying them to the Generalised Railroad Crossing Problem. Subsequently it has

been shown that all models of the solution exist within a hierarchy, thus conjectures on the proof of correctness can be made. Sensibly, the first piece of future work should be to demonstrate that these conjectures are true.

Proposed further work can be divided into X sections:

- Work on documented techniques;
- Extensions to techniques;
- Extending Boundaries; and
- Related Further Work.

Work on Techniques Documented.

Perhaps the one weakness of the work has been the manual nature of the transformations performed, since the author had very limited access to algebraic specification tools. However, the algebraic style and nature taken give confidence that implementation in actual tools will be straightforward, with only small adjustments needed to the descriptions given to take account of any notation required be the chosen tool. Hence the second proposed piece of work is the automation of the transformations.

Areas that should be explored within the transformations themselves are those concerned with the options we have not considered, for example gaining a further understanding of the changes to the transformations required if cycle inconsistent SCAs are considered; or the impact of other mapping functions or tuple management systems.

Extensions to techniques.

In developing our transformations, the work has been conscious of issues that will begin to tax the minds of safety engineers in the future. Take a system that is today implemented in a particular way, that has functionality that may be required in the future, but the hardware it is implemented on may not be available (or there

may be another reason for changing the hardware implementation). If this is the case, then extending the set of transformations to include a transformation from concrete to abstract dSCA would aid the understanding of the new system. The old implementation could be turned into an abstract dSCA and then manipulated as required before being turned back into a concrete dSCA representing the new hardware configuration.

One other piece of future work would be the investigation of allowing the machine algebra, M_A to alter across models. The current work requires M_A to be consistent across all models, which immediately precludes the use of higher level data objects. Allowing M_A to alter introduces the benefits of higher level programming concepts to be used, such as enumerations, but adds levels of complexity to transformations. Some care can be used in determining what can be allowed, enumerations for example would be relatively easy to implement as in one abstract model they can be enumerations and at a lower level of abstraction could be implemented as integers. Potential complexities arise where the abstract data types require the more concrete dSCAs to implement multiple modules per high level concept. An example of this would be an abstract dSCA that uses integers, and a concrete dSCA that only operated on bits; if the abstract dSCA used 8-bit words, then the concrete dSCA would need eight modules per abstract module to manage 32-bit integer operations. Note that SCAs, and more importantly the hierarchy of SCAs, can manage this as Poole Tucker and Holden show in their paper. Another interesting element to look at for future work is that of increasing the spatial efficiency for cycle inconsistent dSCAs that are not totally cycle inconsistent.

Extending Boundaries.

At the boundaries of this work there is ample opportunity for future work. It was noted in the introduction of this thesis that work has been done on directly producing functional language programmes from formal specification languages, and

we have indicated how our work was initially inspired by the dataflow approach to implementation of functional languages. Fruitful results maybe gained by bridging the gap between the work on generation of functional language programs from formal methods, implementation of those programs as dataflow graphs, and finally implementation of those dataflow graphs as SCAs (where this work can then complete the path to actual implementation). At the other boundary, this work has targeted a machine with a shared-memory like implementation, other models of computing should be considered.

Related Further Work.

Finally, the author feels that the field of Petri-nets may provide some benefits when looking at analysis of the SCAs used in out transformations. Heiner and Heisel discuss the modelling of safety-critical systems with Z and Petri nets (see [HH99]), and it would appear, at a trivial level, there is a link between SCAs and Petri Nets - in that SCAs can be converted to Petri Nets. A classical Petri net is a directed graph which consists of nodes and arcs (see Peterson [Pet81]), an SCA consists of nodes and channels - however, it may be more appropriate to consider the nodes of an SCA graph as arcs in a petri net and the channels as nodes, and introduce a new petri net node for the clock. Considering SCAs as petri nets may open up the work already done on safety analysis using petri nets (e.g. Leverson and Stolzy's work [LS87]).

This thesis has achieved the aim it set out to study.

Bibliography

[ABR99] E. Astesiano, M. Broy, and G. Reggio. *Algebraic Foundations of Systems Sepcifications*, chapter 13, pages 467–520. Springer, 1999.

[ACJ+96] M. A. Ardis, J. A. Chaves, L. J. Jagadeesan, P. Mataga, C. Puchol, M. G. Staskauskas, and J. von Olnhausen. A framework for evaluating specification methods for reactive systems experience report. *IEEE Transactions on Software Engineering*, 22(6):378–389, June 1996.

[AH28] W. Ackermann and D. Hilbert. *Gundzuge der Theoretischen Logik.* Springer, Berlin, 1928.

[AP80] V. Kathail Arvind and K. K. Pingali. A dataflow architecture with tagged tokens. Technical Report LCS Memo TM-174, MIT, 1980.

[AWM+77] S. Amoroso, P. Wegner, D. Morris, D. White, W. Loper, W. Cambell, and C. Showaltzer. Language evaluation coordinating committee report to the Higher Order Language Working Group (HOLWG). Technical Report AD-A037 634, US DoD, January 1977.

[B$\ddot{5}$4] C. Böhm. Calculatrices Digitales: Du déchiffrage de formules logico-mathématiques par la machine mêmem dans la conception du programme [Digital Computers: On the deciphering of logical-mathematical formulae by the machine itself during the conception of the program]. *Annali di Mathematica Pura ed Applicata*, 37(4):175–217, 1954.

[Bac54] J. Backus. The IBM 701 speedcoding system. *Journal of the Association for Computing Machinery*, 1:4–6, 1954.

[Bac59] J. Backus. The syntax and semantics of the proposed algebraic langauge og the zurich acm-gamm conference. In *Proceedings of an International Conference on Information Processing*, pages 125–132, UNESCO, Paris, 1959. Butterworth, London.

[Bac78] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Comm. ACM*, 21(8):613–641, August 1978.

[Bar97] J. Barnes. *High Integrity Ada - the SPARK approach.* Addison-Wesley, 1997.

[BBB+57] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. *Proceedings of the Western Joint Computer Conference, Los Angeles*, 1957.

[BBG+63] J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, A.J. Perlis, H. Rutishauser, K. Samelson, and B. Vanquois. Revised report on the algorithmic language ALGOL 68. *Communications of the ACM*, 6:1–23, 1963.

[BF93] R. W. Butler and G. B. Finelli. The infeasability of quantifying the reliability of life-critical real-time software. In *IEEE Transactions on Software Engineering*, volume 19, pages 3–12. January 1993.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification*, chapter The Algebraic Specification Formalism ASF. Addison-Wesley, 1989.

[BHL+96] J. P. Bowen, C. A. R. Hoare, H. Langmaack, E.-R. Olderog, and A. P. Ravn. A ProCoS II project final report: ESPRIT Basic Research project 7071. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 59:76–99, June 1996.

[Bin97] D. Binkley. C++ in safety critical systems. In R. Hamlet, editor, *Annals of Software Engineering*, volume 4, pages 223–234. Kluwer Academic Publishers, 1997.

[BMSU98] N. Bjorner, Z. Manna, H. Sipma, and T. Uribe. Deductive verification of real-time systems using STeP. Technical Report STAN-CS-TR-98-1616, Computer Science Department, Stanford University, December 1998.

[Boe81] B. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[Bow96] J. P. Bowen. Hardware compilation of the ProCoS gas burner case study using logic programming. In G. Brown, editor, *Proc. ProCoS-US Hardware Synthesis and Verification Workshop*, Cornell University, Ithaca, New York, USA, 14–16 August 1996.

[Bur96] R Burnett. Anticipate and prevent - managing the legal risks in safety critical software. In F Redmill and T Anderson, editors, *Safety Critical Systems: The Convergence of High Tech and Human Factors. Prooceedings of the Fourth Safety-critical Systems Symposium*, pages 139–152, London, 1996. Springer-Verlag.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[CA78] L. Chen and A. Avizienis. N-version programming: A fault-toleance approach to reliability of software operation. In *Digest of papers FTCS-8: Eight Annual International Conference on Fault Tolerant Computing*, pages 3–9, June 1978.

[CBB+96] M. Chudleigh, C. Berridge, B. Butler, R. May, and I. Poole. SADLI : Functional programming in a safety critical application. In F. Redmill and T. Anderson, editors, *Safety-critical Systems in The convergence of High Tech and Human Factors - Proc. of the 4th Safety-critical Systems Symposium*. Springer-Verlag, 1996.

[CDE+99] M. Clavel, F. Dur'an, S. Eker, P. Lincoln, N. Mart'i-Oliet, J. Meseguer, and J. Quesada. The Maude system. In *Procs. of RTA '99*, number 1631 in LNCS, pages 240–243. Springer, 1999.

[CEN97] CENELEC. Railway applications: Software for railway control and protection systems. Technical Report EN-50128, CENELEC, 1997.

[CG90] B. Carre and J. Garnsworthy. Spark - an annotated ada subset for safety-critical programming. In *Tri-Ada*, 1990.

[CGM92] B. Carre, J. Garnsworthy, and W. Marsh. Spark : A safety-related ada subset. In *Ada UK Conference*, October 1992.

[CGR93] D. Craigen, S. Gerhart, and T.J. Ralston. An international survey of industrial applications of formal methods (volume 1: Purpose, approach, analysis and conclusions, volume 2: Case studies. Technical Report NIST GCR 93/626-V1 & NIST GCR 93-626-V2, U.S. Department of Commerce, Technology Administration, National Istitute of Standards and Technology, Computer Systems Laboratory., National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA, 1993.

[Cho56] N. Chomsky. Three models for the description of a language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[Cho59] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[Chu36a] A. Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1:40–41, 1936.

[Chu36b] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[Coh88] A. Cohn. A proof correctness of the viper microprocessor: The first level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71. Kluwer, 1988.

[Coh89] A. Cohn. Correctness properties of the viper block model: The second level. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91. Springer-Verlag, 1989.

[Com94] Main Commission. Report on the accident to airbus a320-211 aircraft in Warsaw on 14 september 1993. Technical report, Aircraft Accident Investigation Warsaw, Warsaw, 1994.

[Cur75] M. R. Currie. Memorandum from director defense research and engineering. Internal documents to the US Government, January 1975.

[Cur76] M. R. Currie. Memorandum from director defense research and engineering. Internal documents to the US Government, May 1976.

[Cut89] N. J. Cutland. *Computability: An introduction to recursive function theory.* Cambridge University Press, Cambridge, 1989.

[Dav93] D. Davies. Legal liability. In Bennet P, editor, *Safety Aspects of Computer Control.* Butterworth-Heinemann Ltd, 1993.

[DI90] R. Di Giovanni and P. L. Iachini. HOOD and Z for the development of complex systems. In Dines Bjørner, C. A. R. Hoare, and Hans Langmaack, editors, *VDM '90, VDM and Z - Formal Methods in Software Development, Third International Symposium of VDM Europe, Kiel, FRG, April*, volume 428 of *Lecture Notes in Computer Science*, pages 262–289. VDM-Europe, Springer-Verlag, 1990.

[Dij68] E. Dijkstra. Goto considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[Dij75] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dij76] E. Dijkstra. *A discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Dij82] E. Dijkstra. *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag New York, 1982.

[DKLM98] M. Dunstan, T. Kelsey, S. Linton, and U. Martin. Lightweight formal methods for computer algebra systems. In *International Symposium on Symbolic and Algebraic Computation*, pages 80–87, 1998.

[DoD84] DoD. Military standard: System safety program requirements. Standard MIL-STD-882B, Department of Defense, Washington DC 20301, USA, 30 March 1984.

[Dra92] C. Draper. Practical experiences of Z and SSADM. In Jonathan P. Bowen and J. E. Nicholls, editors, *Z User Workshop*, Workshops in Computing, pages 240–251. Springer, 1992.

[Dun79] W. Duncan. Letter from the secretary of defense. Internal documents to the US Government, May 1979.

[ELC+98] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *Software Engineering*, 24(1):4–14, 1998.

[EM92] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1992.

[ESA91] ESA. ESA Software Engineering Standards. Technical Report PSS-05-0 Issue 2, European Space Agency, 8–10 rue Mario-Nikis, 75738 Paris Cedex, France, February 1991.

[ESP94] ESPRIT. *Predictably Dependable Computing Systems Second Year Report*, 1994.

[EST91] S.M Eker, V. Stavridou, and J.V. Tucker. Verification of synchronous concurrent algorithms using OBJ3. a case study of the pixel planes architecture. In G Jones and M Sheeran, editors, *Designing Correct Circuits*, pages 231–252. Springer, 1991.

[ET88] S.M. Eker and J.V. Tucker. Specification, derivation and verification of concurrent line drawing algorithms and architectures. In R. A. Earnshaw, editor, *Theoretical foundations of computer graphics and CAD*, pages 449–516, Heidelberg, 1988. Springer-Verlah.

[ET89a] S.M. Eker and J.V. Tucker. Specification and verification of synchronous concurrent algorithms: a case study of the pixel planes architecture. In R A Earnshaw P M Drew and T R Heywood, editors, *Parallel processing for computer vision and display*, pages 16–49. Addison-Wesley, 1989.

[ET89b] S.M. Eker and J.V. Tucker. Tools for the development of rasterisation algorithms. In R A Earnshaw and B Wyvill, editors, *New Advances in Computer Graphics, Proceedings of Computer Graphics Internationsal '89*, pages 53–89, Tokyo, 1989. Springer.

[FBGL94] J. Fitzgerald, T.M. Brookes, M.A. Green, and P.G. Larsen. Formal and informal specifications of a secure system component: first results in a comparative study. In Naftali Denvir and Bertran, editors, *Formal Methods Europe '94 - Industrial Benefit of Formal Methods*, pages 35–44. Springer-Verlag, October 1994.

[FH98] A.C.J. Fox and N. A. Harman. Algebraic models of superscalar microprocessor implementations: A case study. Technical Report CSR20-98, University of Wales, Swansea, 1998.

[FKPW96] M. Flick, K. Kemp, E. Pofahl, and K. Wolf. Rules for programming in C. http://tuvasi.com/c-rules.htm, 1996.

[Flo62] R. W. Floyd. On the nonexistence of a phrase structure grammar for ALGOL 60. *Communications of the ACM*, 5(10):483–484, 1962.

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwatz, editor, *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[GAM] Supplier guide for validation of automated systems in pharmaceutical manufacture. Technical report, ISPE, 3816 W. Linebaugh Avenue, Suite 412, Tampa, Florida 33624, USA.

[GKW85] J. R. Gurd, C. C Kirkham, and I. Watson. The manchester prototype dataflow computer. *Comm. ACM*, 28(1):34–52, January 1985.

[Gov84] UK Governement. *Health and Safety at Work etc. Act 1974 (1984, c.37).* Her Majesty's Stationary Office, 1984.

[Hat95] L. Hatten. Safer C : Developing software for high-integrity and safety-critical systems. In *McGraw-Hill International Series in Software Engineering.* McGraw-Hill, 1995.

[Hei95] M. Heisel. Six steps towards provably safe software. In Gerhard Rabe, editor, *SAFECOMP'95: 14th International Conference on Computer Safety, Reliability and Security*, pages 191–205, Belgirate, Italy, 1995. Springer-Verlag.

[Het84] W. Hetzel. *The complete guide to software testing.* Granada, 1984.

[HH99] M. Heiner and M. Heisel. Modeling safety-critical systems with Z and petri nets. *Lecture Notes in Computer Science: Computer Safety, Reliability and Security*, 1698:361–374, 1999.

[HJL93] C. Heitmeyer, R. Jeffords, and B Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proceedings of the 10th International Workshop on Real-Time Operating Systems and Software*, May 1993.

[HL94a] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verificaton of real-time system. Technical Report

MIT/LCS/TM-511, Labratory for Computer Science, Massachusetts Institute of Technology, November 1994.

[HL94b] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verificaton of real-time system. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 120–131. IEEE Computer Society Press, 1994.

[HLOR93] C. A. R. Hoare, Hans Langmaack, Ernst-Rdiger Olderog, and Anders P. Ravn. Overview of ESPRIT ProCoS II project. Technical Report [CO-ORD CARH 1/1], Oxford University Computing Laboratory, September 1993.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hoa96] C. A. R. Hoare. How did software get so reliable without proof. In M-C. Gaudel and J. Woodcock, editors, *Formal Methods Europe: Industrial Benefit and Advances in Formal Methods.* Springer, 1996.

[Hob90] K. Hobley. *Formal Specification and Verification of Synchronous Concurrent Algorithms.* PhD thesis, University of Swansea, 1990.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, NATO ASI Series, pages 477–498. Springer, 1985.

[HPT96] A V. Holden, M.J. Poole, and J.V. Tucker. An algorithmic model of the mammalian heart: propogation, vulnerability, re-entry and fibrillation. *International Journal of Bifurcation and Chaos*, 6:1623–1635, 1996.

[HT88] N.A. Harman and J.V. Tucker. Formal specifications and the design of verifiable computers. In *Proceddings of 1988 UK IT Conference*, pages 500–503. Institute of Electrical Engineers (IEE), 1988.

[HT89] N.A. Harman and J.V. Tucker. Clocks, retimings, and the formal specification of a UART. In G Milne, editor, *The fusion of hardware design and verification*, pages 375–396. North-Holland, 1989.

[HT90] N.A. Harman and J.V. Tucker. The formal specification of a digital correlator 1: User specification process. In K McEvoy and J V Tucker, editors, *Theoretical foundations of LSI design*, pages 161–262. Cambridge University Press, 1990.

[HT91] N.A. Harman and J.V. Tucker. Consistent refinements of specifications for digital systems. In P Prinetto and P Camurati, editors, *Proceedings of ESPRIT BRA CHARME Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 281–304, Amsterdam, 1991. Elsevier.

[HT93] N.A. Harman and J.V. Tucker. Algebraic models of computers and the correctness of micro processors. In G J Milne and L Pierre, editors, *Correct Hardware design and verification methods*, pages 92–108, Berlin, 1993. Springer Lecture Notes in Computer Science 683.

[HT94] K. Hobley and J.V. Tucker. Clocks, retimings and the transformation of synchronous concurrent algorithms. In G Megson, editor, *Transformational approaches to systolic design*, pages 99–132. Chapman Hall, 1994.

[HT96] N.A. Harman and J.V. Tucker. Algebraic models of microprocessors: architecture and organisation. *Acta Informatica*, 33:421–456, 1996.

[HT97] N.A. Harman and J.V. Tucker. Algebraic models of microprocessors: the verification of a simple computer. In V Stavridou, editor, *Mathematics for dependable systems II, Proceedings of the Second IMA Conference*, pages 135–169. Oxford University Press, 1997.

[HTT89] K.M. Hobley, B.C. Thompson, and J.V. Tucker. Specification and verification of synchronous concurrent algorithms: a case study of a convolution algorithm. In G Milne, editor, *The fusion of hardware design and verification*, pages 347–374. North-Holland, 1989.

[HTT90] A.V. Holden, B.C. Thompson, and J.V. Tucker. The computational structure of neural systems. In A V Holden and V I Kryukov, editors, *Neurocomputers and attention. I:Neurobiology, synchronisation and chaos*, pages 223–240. Manchester University Press, 1990.

[Hun94] W.A. Hunt. *FM8501 : A Verified Microprocessor*, volume 795 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, Berlin, 1994.

[IBN96] Ulla Isaksen, Jonathan P. Bowen, and Nimal Nissanke. System and software safety in critical systems, 1996.

[IEC86] IEC. *IEC 880 : Software for computers in the safety systems of nuclear power stations.* 1986.

[IEC96] IEC. Medical electrical equipment - part 1: General requirements for safety 4: Collateral standard: Programmable electrical medical systems. Technical Report IEC 601-1-4, IEC, 1996.

[IEC99] IEC. *IEC 61508 : Functional Safety : safety related systems Parts 1-7.* 1999.

[Int94] Intel. Statistical analysis of floating point flaw. Technical report, Intel, November 1994.

[ISO87] ISO. *ISO 8652:1987 - Programming Language - Ada.* 1987.

[ISO89] ISO. ISO 8807. information processing systems - open systems interconnection - lotos - a formal description technique based on temporal ordering of observational behaviour. Technical report, ISO, 1989.

[ISO90] ISO. *ISO/IEC 9899:1990 - Programming Language - C.* 1990.

[ISO95] ISO. *Information Technology - Programming Languages - Ada. Ada Reference Manual - ISO/IEC 8652:1995(E).* 1995.

[ISO00] ISO/IEC. ISO/IEC TR 15942:2000 information technology - programming languages - guide for the use of the ada programming language in high integrity systems. Technical report, 2000.

[JM92] C. B. Jones and A. M. Mccauley. Formal methods - selected historical references. Technical Report UMCS-92-12-2, Manchester University, Department of Computer Science, 1992.

[Jon96] C. B. Jones. Formal methods light: A rigorous approach to formal methods. *Computer*, 29(4):20–21, 1996.

[JW96] D. Jackson and J. Wing. Formal methods light: Lightweight formal methods. *Computer*, 29(4):21–22, 1996.

[KL86] J. Knight and N. Leveson. an experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[KL90] J. Knight and N. Leveson. A reply to the criticisms of the knight and leveson experiment. *ACM Software Engineering Notes*, January 1990.

[Kle52] S. C. Kleene. *Introduction to Metamathematics.* North-Holland, Amsterdam, 1952.

[Kle67] S. C. Kleene. *Mathematical Logic.* Wiley, New York, 1967.

[KMP98] Y. Kesten, Z. Manna, and A. Pnueli. Verification of clocked and hybrid systems. In G. Rozenberg and F. W. Vaandrager, editors, *Lectures in Embedded Systems*, number LNCS 1494, pages 4–73. Springer-Verlag, 1998.

[KR78] B. Kernighan and D. Ritchie. *The C programming Language.* Prentice Hall, 1978.

[Lam91] L. Lamport. The temporal actions of logic. Technical Report Research Report 79, Digital Equipment Corporation, Systems Research Center, December 1991.

[Lam92] L. Lamport. Hybrid systems in TLA +. In *Hybrid Systems*, pages 77–102, 1992.

[Lev86] Nancy G. Leveson. Software safety: Why, what, and how. *ACM Computing Surveys*, 18(2):125–163, 1986.

[Lio96] J. Lions. Ariane 5 - flight 501 failure. Technical report, European Space Agenct, 1996.

[LS87] N. G. Leveson and J. L. Stolzy. Safety analysis using petri nets. In *IEEE Transactions on Software Engineering*, 1987.

[LS91] C E Leiserson and J B Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[LS97] K. Lano and A. Sanchez. Design of real-time control systems for event driven operations. In *Formal Methods Europe*, volume 1313 of *LNCS*. Springer-Verlag, 1997.

[LT93] N. Leverson and C. S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[Lut93] R. R. Lutz. Analyzing software requirements errors in safety critical, embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, 1993. IEEE Computer Society Press.

[LvKBO87] D.C. Luckham, F.W. von Henke, Krieg-Bruckner, and O. Owe. *ANNA - A language for annotating Ada Programs*, volume 26 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[McC63] John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963. corrected version of 1961 paper given at Western Joint Computer Conference, May 1961.

[Mil80] R. Milner. Calculus of communicating systems. In *Lecture Notes in Computer Science*, volume 92. Springer, 1980.

[MISRA94] MIRA Motor Industry Software Reliability Association. *Development Guidelines For Vehicle Based Software*. November 1994.

[MO95] M Müller-Olm. Compiling the gas burner case study. Technical Report [MMO 16/1], ProCoS Technical Report, August 1995.

[MOD89] MOD. *Interim Defence Standard 00-55 - The Procurement of Safety Critical Software in Defence Equipment.* Crown, 1989.

[MOD91] MOD. *Interim Defence Standard 00-56 : Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment.* Crown, 1991.

[MOD96] MOD. *Defence Standard 00-56 : Safety Management Requirements for Defence Systems.* Crown, 1996.

[MOD97] MOD. *Defence Standard 00-55 : The Procurement of Safety Related Software in Defence Equipment.* Crown, 1997.

[MP93] Z. Manna and A. Pnueli. Models for reactivity. *ACTA Information*, 30:609–678, 1993.

[MT87] A. R. Martin and J. V. Tucker. Concurrent assignment representation of synchronous systems. In A. J. Nijmand J. W. de Bakker and P.C. Treleaven, editors, *PARLE: Parallel architectures and languages Europe. Vol II: Parallel Languages*, pages 369–386. Springer-Verlag, 1987.

[MT88] K. Meinke and J.V. Tucker. Scope and limits of synchronous concurrent computation. In F H Vogt, editor, *Concurreny '88*, Springer Lecture Notes in Computer Science, pages 163–180. Springer-Verlag, 1988.

[MT93] B. McConnell and J.V. Tucker. Infinite synchronous concurrent algorithms: the specification and verification of a hardware stack. In W Brauer F L Bauer and H Schwichtenberg, editors, *Proceedings of*

NATO Summer School 1991 at Marktoberdorf, in Logic and algebra of specification, pages 321–375. Springer, 1993.

[MT89] A.R. Martin and J.V. Tucker. Concurrent assignment representation of synchronous systems. *Parallel Computing*, 9:227–256, 1988-89.

[MV89] S. Mauw and G.J. Veltink. An introduction to psf. In J. Diaz and F. Orejas, editors, *Proc TAPSOFT'89, vol 2*, volume 352 of *Lecture Notes in Computer Science*, pages 375–389. Springer, 1989.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII, 1990.

[MZL02] K. Lundqvist M. Zimmerman and N. Leveson. Investigating the readability of state-based formal requirements specification languages. In *Interneational Conference on Software Engineering*, May 2002.

[NK91] T. Nakajo and H. Kume. A case history analysis of software error cause-effect relationships. *IEEE Transactions on Software Engineering*, 8(17):830–838, August 1991.

[NR69] P. Naur and B. Randell, editors. *Software Engineering: Report on a Conference sponsored by the NATO Science Committee.* Scientific Affairs Division, NATO, Garmisch, Germany, January 1969.

[Pet81] J. L. Peterson. *Petri net theory and modelling of systems.* Prentice Hall, 1981.

[PHT98] M. J. Poole, A. V. Holden, and J. V. Tucker. Hierarchies of spatially extended systems and synchronous concurrent algorithms. In *Prospects for Hardware Foundations*, pages 184–235, 1998.

[Puc95] C. Puchol. A solution to the generalized railroad crossing problem in ESTEREL. Technical Report CS-TR-95-05, University of Austin at Texas, 1, 1995.

[PW99] M. Piveropoulos and A. Welings. Requirements engineering for hard real-time systems: the σ notation and a case study, 1999.

[Rei91] W. Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80(1):1–34, 1991.

[Rei98] W. Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets.* Springer, 1998.

[RIA91] RIA. Safety related software for railway signalling. BRB/LU Ltd/RIA technical specification no. 23, Railway Industry Association, 6 Buckingham Gate, London SW1E 6JP, UK, 1991. Consultative Document.

[RLKL95] B. Randell, J-C. Laprie, H. Kopetz, and B. Littlewood, editors. *Predictably Dependable Computer Systems.* Springer-Verlag, Berlin, 1995.

[RRH93] A.P. Ravn, H. Rischel, and K.M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Transactions in Software Engineering*, 19(1):41–55, January 1993.

[RTC92] RTCA/EUROCAE. *DO-178B/ED-12B : Software Considerations in Airborne Systems and Equipment Certification.* RTCA/EUROCAE, RTCA, Inc. 1140 Connecticut Avenue, N.W., Suite 1020, WASHINGTON DC 20036, USA, 1992.

[Rum77] J. Rumbaugh. A dataflow multiprocessor. *IEEE Trans. on Comp.*, 2:140–146, 1977.

[Rus94] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[SA90] L. Semmens and P. Allen. Using Yourdon and Z: An approach to formal specification. In J. E. Nicholls, editor, *Proceedings of the Fifth Annual Z User Meeting on Z User Workshop*, Workshops in Computing, pages 228–253. Springer-Verlag, 1990.

[Sha85] J.A. Sharp. *Data Flow Computing.* Wiley, 1985.

[Som95] I. Sommerville. *Software Engineering, 5th Edition.* Addison Wesley, 5th edition, 1995.

[SPM] SPMN. Spmn software development bulletin.

[Ste95] K. Stephens. *An Algebraic Approach to Syntax, Semantics, and Compilation.* PhD thesis, University of Wales, Swansea, 1995.

[Ste97] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[Ste00a] L. J. Steggles. Specifying and verifying real-time systems using second-order algebraic methods: A case study of the railroad crossing controller. Technical Report CS-TR: 697, Department of Computing Science, University of Newcastle, 2000.

[Ste00b] L. J. Steggles. Specifying and verifying real-time systems using second-order algebraic methods: A case study of the railroad crossing controller. In *Journal of Universal Computer Science*, volume 6, pages 460–473. Springer, 2000.

[Tac04] A. Tacy. Safety engineering summary. Private Correspondance to British Computer Society, later edited in BCS publications, 2004.

[TBL96] J. Fitzgerald T. Brookes and P. Larsen. Formal and informal specifications of a secure system component: Final results in a cmparative study. In J.C.P Woodcock M-C. Gauel, editor, *Formal Methods Europe '96: Industrial Benefit and Advances in Formal Methods*, number 1051, pages 214–227. Springer-Verlag, 1996.

[Tho87] B.C. Thompson. *A mathematical Theory of Synchronous Concurrent Algorithms.* PhD thesis, School of COmputer Studies, University of Leeds, 1987.

[TT85] B. C. Thompson and J. V. Tucker. Theoretical considerations in algortihm design. In R. A. Earnshaw, editor, *Fundamental Algorithms for Computer Graphics*, pages 855–878, Heidelberg, 1985. Springer.

[TT88] B.C. Thompson and V Tucker, J. A parallel deterministic language and its application to synchronous concurrent algorithms,. In *Proceedings of 1988 UJ IT Conference*, pages 228,231. Institute of Electrical Engineers (IEE), 1988.

[TT91] J.V. Tucker and B.C. Thompson. Algebraic specification of synchronous concurrent algorithms and architectures. Technical Report CSR-9-91, Universtiy of Wales, Swansea, 1991.

[TT94] J.V. Tucker and B.C Thompson. Equational specifications of synchronous concurrent algorithms and architectures. Technical Report CSR-15-94, Universtiy of Wales, Swansea, 1994.

[Tur36] A. M. Turing. On computable numbers, with an application of the Entscheidungsproblem. *Proceedings London Mathematical Society*, 42,43:230–265, 544–546, 1936. Reprinted in Dav65.

[Wea99] B. Whichman and et al. Guidance on the use of the ada programming language in high integrity systems. In *Ada Letters*, 1999.

[Whi93] W. Whitaker. Ada - the project, the dod high order language working group. *ACM SIGPLAN Notices*, 28(3), March 1993.

[Wil49] M. V. Wilkes. Program design for a high-speed automatic calculating machine. *Journal of Scientific Instruments*, 26:217–220, 1949.

[Wil52] M. V. Wilkes. Pure and applied programming. *Proceedings of the ACM National Conference*, pages 121–124, 1952.

[Wil53] M. V. Wilkes. The use of a 'floating-address' system for orders in an automatic digital computer. *Proceedings of Cambridge Philosophical Society*, 49:84–89, 1953.

[Wir90] M. Wirsing. Algebraic specification. In J van Leewen, editor, *The Handbook of theoretical computer science*, pages 675–788. Elsevier, 1990.

[WWG96] M A Watson-Walker and R J Gray. The institution of railway signalling engineers licensing scheme - promoting competence in the workplace. In F Redmill and T Anderson, editors, *Safety Critical Symposium: The Convergence of High Tech and Human Factors. Proceedings of the Fourth Safety Critical Systems Symposium*, pages 124–138, London, 1996. Springer-Verlag.

Appendix A

Fundamental Algebraic Specifications

A.1 Synchronous Concurrent Algorithm Specification (SCAAlgebra)

This defines the specification that defines a standard Synchronous Concurrent Algorithm.

Begin

Specification *SCAAlgebra*

Import

Sorts

Constant Symbols

Function Names

$$\begin{aligned}
CreateSCA: \; & Name \times ImpList \times SortList \times \\
& ConsList \times VFOpList \times \gamma OpList \times \\
& \beta OpList \times \delta OpList \times IVEqList \times \\
& STEqList \times \gamma EqList \times \beta EqList \times \\
& \delta EqList \rightarrow SCAAlgebra \\
GetImport: \; & SCAAlgebra \rightarrow ImpList \\
GetSorts: \; & SCAAlgebra \rightarrow SortList \\
GetConsts: \; & SCAAlgebra \rightarrow ConsList \\
GetVFOps: \; & SCAAlgebra \rightarrow VFOpList \\
Get\gamma Ops: \; & SCAAlgebra \rightarrow \gamma OpList \\
Get\beta Ops: \; & SCAAlgebra \rightarrow \beta OpList \\
Get\delta Ops: \; & SCAAlgebra \rightarrow \delta OpList \\
GetIVEqs: \; & SCAAlgebra \rightarrow IVEqList \\
GetSTEqs: \; & SCAAlgebra \rightarrow STEqList
\end{aligned}$$

$Get\gamma Eqs : SCAAlgebra \rightarrow \gamma EqList$
$Get\beta Eqs : SCAAlgebra \rightarrow \beta EqList$
$Get\delta Eqs : SCAAlgebra \rightarrow \delta EqList$

Equations

$$CreateSCA \begin{pmatrix} name, \\ import, \\ sorts, \\ constants, \\ opsVF, \\ ops\gamma, \\ ops\beta, \\ ops\delta, \\ eqsVFIV, \\ eqsVFST, \\ eqs\gamma, \\ eqs\beta, \\ eqs\delta, \end{pmatrix} = \begin{pmatrix} \textbf{Begin} \\ \quad \textbf{Specification } name \\ \quad \textbf{Import } import \\ \quad \textbf{Sorts } sorts \\ \quad \textbf{Constant Symbols } constants \\ \quad \textbf{VF Function Names } opsVF \\ \quad \gamma \textbf{ Function Names } ops\gamma \\ \quad \beta \textbf{ Function Names } ops\beta \\ \quad \delta \textbf{ Function Names } ops \\ \quad \textbf{IV Equations } eqsVFIV \\ \quad \textbf{ST Equations } eqsVFST \\ \quad \gamma \textbf{ Equations } eqs\gamma \\ \quad \beta \textbf{ Equations } eqs\beta \\ \quad \delta \textbf{ Equations } eqs\delta \\ \textbf{End} \end{pmatrix}$$

$$GetImport \begin{pmatrix} \textbf{Begin} \\ \quad \textbf{Specification } name \\ \quad \textbf{Import } import \\ \quad \textbf{Sorts } sorts \\ \quad \textbf{Constant Symbols } constants \\ \quad \textbf{VF Function Names } opsVF \\ \quad \gamma \textbf{ Function Names } ops\gamma \\ \quad \beta \textbf{ Function Names } ops\beta \\ \quad \delta \textbf{ Function Names } ops \\ \quad \textbf{IV Equations } eqsVFIV \\ \quad \textbf{ST Equations } eqsVFST \\ \quad \gamma \textbf{ Equations } eqs\gamma \\ \quad \beta \textbf{ Equations } eqs\beta \\ \quad \delta \textbf{ Equations } eqs\delta \\ \textbf{End} \end{pmatrix} = import$$

$$GetSorts \begin{pmatrix} \textbf{Begin} \\ \quad \textbf{Specification } name \\ \quad \textbf{Import } import \\ \quad \textbf{Sorts } sorts \\ \quad \textbf{Constant Symbols } constants \\ \quad \textbf{VF Function Names } opsVF \\ \quad \gamma \textbf{ Function Names } ops\gamma \\ \quad \beta \textbf{ Function Names } ops\beta \\ \quad \delta \textbf{ Function Names } ops \\ \quad \textbf{IV Equations } eqsVFIV \\ \quad \textbf{ST Equations } eqsVFST \\ \quad \gamma \textbf{ Equations } eqs\gamma \\ \quad \beta \textbf{ Equations } eqs\beta \\ \quad \delta \textbf{ Equations } eqs\delta \\ \textbf{End} \end{pmatrix} = sorts$$

GetConsts = *constants*

Begin
Specification *name*
Import *import*
Sorts *sorts*
Constant Symbols *constants*
VF Function Names *opsVF*
γ **Function Names** *ops*γ
β **Function Names** *ops*β
δ **Function Names** *ops*
IV Equations *eqsVFIV*
ST Equations *eqsVFST*
γ **Equations** *eqs*γ
β **Equations** *eqs*β
δ **Equations** *eqs*δ
End

GetVFOps = *opsVF*

Begin
Specification *name*
Import *import*
Sorts *sorts*
Constant Symbols *constants*
VF Function Names *opsVF*
γ **Function Names** *ops*γ
β **Function Names** *ops*β
δ **Function Names** *ops*
IV Equations *eqsVFIV*
ST Equations *eqsVFST*
γ **Equations** *eqs*γ
β **Equations** *eqs*β
δ **Equations** *eqs*δ
End

Getγ*Ops* = *ops*γ

Begin
Specification *name*
Import *import*
Sorts *sorts*
Constant Symbols *constants*
VF Function Names *opsVF*
γ **Function Names** *ops*γ
β **Function Names** *ops*β
δ **Function Names** *ops*
IV Equations *eqsVFIV*
ST Equations *eqsVFST*
γ **Equations** *eqs*γ
β **Equations** *eqs*β
δ **Equations** *eqs*δ
End

$$
\begin{array}{ll}
\left.\begin{array}{l}
\textbf{Begin} \\
\quad \textbf{Specification } name \\
\quad \textbf{Import } import \\
\quad \textbf{Sorts } sorts \\
\quad \textbf{Constant Symbols } constants \\
\quad \textbf{VF Function Names } opsVF \\
\quad \gamma \textbf{ Function Names } ops\gamma \\
\quad \beta \textbf{ Function Names } ops\beta \\
\quad \delta \textbf{ Function Names } ops \\
\quad \textbf{IV Equations } eqsVFIV \\
\quad \textbf{ST Equations } eqsVFST \\
\quad \gamma \textbf{ Equations } eqs\gamma \\
\quad \beta \textbf{ Equations } eqs\beta \\
\quad \delta \textbf{ Equations } eqs\delta \\
\textbf{End}
\end{array}\right\} & Get\beta Ops = ops\beta \\
\left.\begin{array}{l}
\textbf{Begin} \\
\quad \textbf{Specification } name \\
\quad \textbf{Import } import \\
\quad \textbf{Sorts } sorts \\
\quad \textbf{Constant Symbols } constants \\
\quad \textbf{VF Function Names } opsVF \\
\quad \gamma \textbf{ Function Names } ops\gamma \\
\quad \beta \textbf{ Function Names } ops\beta \\
\quad \delta \textbf{ Function Names } ops \\
\quad \textbf{IV Equations } eqsVFIV \\
\quad \textbf{ST Equations } eqsVFST \\
\quad \gamma \textbf{ Equations } eqs\gamma \\
\quad \beta \textbf{ Equations } eqs\beta \\
\quad \delta \textbf{ Equations } eqs\delta \\
\textbf{End}
\end{array}\right\} & Get\delta Ops = ops\delta \\
\left.\begin{array}{l}
\textbf{Begin} \\
\quad \textbf{Specification } name \\
\quad \textbf{Import } import \\
\quad \textbf{Sorts } sorts \\
\quad \textbf{Constant Symbols } constants \\
\quad \textbf{VF Function Names } opsVF \\
\quad \gamma \textbf{ Function Names } ops\gamma \\
\quad \beta \textbf{ Function Names } ops\beta \\
\quad \delta \textbf{ Function Names } ops \\
\quad \textbf{IV Equations } eqsVFIV \\
\quad \textbf{ST Equations } eqsVFST \\
\quad \gamma \textbf{ Equations } eqs\gamma \\
\quad \beta \textbf{ Equations } eqs\beta \\
\quad \delta \textbf{ Equations } eqs\delta \\
\textbf{End}
\end{array}\right\} & GetIVEqs = eqsVFIF
\end{array}
$$

$$
\begin{array}{lll}
 & \left\{\begin{array}{l}
\textbf{Begin} \\
\quad \textbf{Specification } name \\
\quad \textbf{Import } import \\
\quad \textbf{Sorts } sorts \\
\quad \textbf{Constant Symbols } constants \\
\quad \textbf{VF Function Names } opsVF \\
\quad \gamma \textbf{ Function Names } ops\gamma \\
\quad \beta \textbf{ Function Names } ops\beta \\
\quad \delta \textbf{ Function Names } ops \\
\quad \textbf{IV Equations } eqsVFIV \\
\quad \textbf{ST Equations } eqsVFST \\
\quad \gamma \textbf{ Equations } eqs\gamma \\
\quad \beta \textbf{ Equations } eqs\beta \\
\quad \delta \textbf{ Equations } eqs\delta \\
\textbf{End}
\end{array}\right. & \\
GetSTEqs & & = eqsVFST \\
 & \left\{\begin{array}{l}
\textbf{Begin} \\
\quad \textbf{Specification } name \\
\quad \textbf{Import } import \\
\quad \textbf{Sorts } sorts \\
\quad \textbf{Constant Symbols } constants \\
\quad \textbf{VF Function Names } opsVF \\
\quad \gamma \textbf{ Function Names } ops\gamma \\
\quad \beta \textbf{ Function Names } ops\beta \\
\quad \delta \textbf{ Function Names } ops \\
\quad \textbf{IV Equations } eqsVFIV \\
\quad \textbf{ST Equations } eqsVFST \\
\quad \gamma \textbf{ Equations } eqs\gamma \\
\quad \beta \textbf{ Equations } eqs\beta \\
\quad \delta \textbf{ Equations } eqs\delta \\
\textbf{End}
\end{array}\right. & \\
Get\gamma Eqs & & = eqs\gamma \\
 & \left\{\begin{array}{l}
\textbf{Begin} \\
\quad \textbf{Specification } name \\
\quad \textbf{Import } import \\
\quad \textbf{Sorts } sorts \\
\quad \textbf{Constant Symbols } constants \\
\quad \textbf{VF Function Names } opsVF \\
\quad \gamma \textbf{ Function Names } ops\gamma \\
\quad \beta \textbf{ Function Names } ops\beta \\
\quad \delta \textbf{ Function Names } ops \\
\quad \textbf{IV Equations } eqsVFIV \\
\quad \textbf{ST Equations } eqsVFST \\
\quad \gamma \textbf{ Equations } eqs\gamma \\
\quad \beta \textbf{ Equations } eqs\beta \\
\quad \delta \textbf{ Equations } eqs\delta \\
\textbf{End}
\end{array}\right. & \\
Get\beta Eqs & & = eqs\beta
\end{array}
$$

$$
Get\delta Eqs \left(\begin{array}{l}
\textbf{Begin} \\
\quad \textbf{Specification } name \\
\quad \textbf{Import } import \\
\quad \textbf{Sorts } sorts \\
\quad \textbf{Constant Symbols } constants \\
\quad \textbf{VF Function Names } opsVF \\
\quad \gamma \textbf{ Function Names } ops\gamma \\
\quad \beta \textbf{ Function Names } ops\beta \\
\quad \delta \textbf{ Function Names } ops \\
\quad \textbf{IV Equations } eqsVFIV \\
\quad \textbf{ST Equations } eqsVFST \\
\quad \gamma \textbf{ Equations } eqs\gamma \\
\quad \beta \textbf{ Equations } eqs\beta \\
\quad \delta \textbf{ Equations } eqs\delta \\
\textbf{End}
\end{array} \right) = eqs\delta
$$

End

A.2 Machine Algebra (M_A) Specification

This describes the machine algebra specification used throughout the example in this thesis.

Begin

Specification M_A

Import N, B

Sorts

Constant Symbols $0, true, false, u$

Function Names

$$add : N \cup \{u\} \times N \cup \{u\} \rightarrow N \cup \{u\}$$
$$sub : N \cup \{u\} \times N \cup \{u\} \rightarrow N \cup \{u\}$$
$$and : B \cup \{u\} \times B \cup \{u\} \rightarrow B \cup \{u\}$$
$$or : B \cup \{u\} \times B \cup \{u\} \rightarrow B \cup \{u\}$$
$$not : B \cup \{u\} \rightarrow B$$
$$eq : N \cup \{u\} \times N \cup \{u\} \rightarrow B \cup \{u\}$$
$$lt : N \cup \{u\} \times N \cup \{u\} \rightarrow B \cup \{u\}$$
$$gt : N \cup \{u\} \times N \cup \{u\} \rightarrow B \cup \{u\}$$
$$cond : B \cup \{u\} \times N \cup \{u\} \times N \cup \{u\} \rightarrow N \cup \{u\}$$

Equations

$$add(a, 0) = a$$
$$add(a, succ(b)) = add(succ(a), b)$$
$$add(u, b) = u$$
$$add(a, u) = u$$
$$sub(a, 0) = a$$
$$sub(succ(a), succ(b)) = sub(a, b)$$
$$sub(u, b) = u$$
$$sub(a, u) = u$$
$$and(true, true) = true$$
$$and(true, false) = false$$
$$and(false, true) = false$$
$$and(false, false) = false$$
$$and(u, b) = u$$
$$and(a, u) = u$$
$$or(true, true) = true$$
$$or(true, false) = true$$
$$or(false, true) = true$$
$$or(false, false) = false$$
$$or(u, b) = u$$
$$or(a, u) = u$$
$$not(true) = false$$
$$not(false) = true$$
$$not(u) = u$$
$$eq(0, 0) = true$$
$$eq(succ(a), 0) = false$$
$$eq(0, succ(b)) = false$$
$$eq(succ(a), succ(b)) = eq(a, b)$$
$$eq(u, b) = u$$
$$eq(a, u) = u$$

$$
\begin{aligned}
lt(0,0) &= false \\
lt(succ(a),0) &= false \\
lt(0,succ(b)) &= true \\
lt(succ(a),succ(b)) &= lt(a,b) \\
lt(u,b) &= u \\
lt(a,u) &= u \\
gt(0,0) &= false \\
gt(succ(a),0) &= true \\
gt(0,suc(b)) &= false \\
gt(succ(a),succ(b)) &= gt(a,b) \\
gt(u,b) &= u \\
gt(a,u) &= u \\
cond(false,b,c) &= c \\
cond(true,b,c) &= b \\
cond(u,b,c) &= u
\end{aligned}
$$

End

A.3 Important List Specifications

A.3.1 γ Function Equation List

Begin

Specification $\gamma SCAEqList$
Import $\gamma SCAEquation$
Constant Symbols $[]$
Function Names

$$\_,\_ : \gamma SCAEquation \times \gamma SCAEqList \rightarrow \gamma SCAEqList$$
$$hd : \gamma SCAEqList \rightarrow \gamma SCAEquation$$
$$tl : \gamma SCAEqList \rightarrow \gamma SCAEqList$$
$$GetEl : \gamma SCAEqList \times N^2 \rightarrow \gamma SCAEqList$$

Equations

$$a, b = a, b$$
$$hd\,([]) = []$$
$$hd\,(a, as) = a$$
$$tl\,([]) = []$$
$$tl\,(a, as) = as$$
$$GetEl([], i) = null$$
$$GetEl\,((\gamma(i, j) = x, eqs), i, j)) = \gamma(i, j) = x$$
$$GetEl\,((\gamma(y, z) = x, eqs), i, j)) = GetEl(eqs, i, j)$$

End

A.3.2 dSCA γ Function Operation List

Begin

Specification $\gamma dSCAEqList$
Import $\gamma dSCAEquation$
Constant Symbols $[]$
Function Names

$$\_,\_ : \gamma dSCAEquation \times \gamma dSCAEqList \rightarrow \gamma dSCAEqList$$
$$hd : \gamma dSCAEqList \rightarrow \gamma dSCAEquation$$
$$tl : \gamma dSCAEqList \rightarrow \gamma dSCAEqList$$
$$GetEl : \gamma dSCAEqList \times N^3 \rightarrow \gamma dSCAEqList$$

Equations

$$a, b = a, b$$
$$hd\,([]) = []$$
$$hd\,(a, as) = a$$
$$tl\,([]) = []$$
$$tl\,(a, as) = as$$
$$GetEl([], i) = null$$
$$GetEl\,((\gamma_k(i, j) = z, eqs), i, j, k) = \gamma_k(i, j) = z$$
$$GetEl\,((\gamma_a(b, c) = z, eqs), i, j, k) = GetEl(eqs, i, j, k)$$

End

A.3.3 β Function Operation List

Begin

Specification $\beta SCAEqList$
Import $\beta SCAEquation$
Constant Symbols []
Function Names

$$\_,\_ : \beta SCAEquation \times \beta SCAEqList \rightarrow \beta SCAEqList$$
$$hd : \beta SCAEqList \rightarrow \beta SCAEquation$$
$$tl : \beta SCAEqList \rightarrow \beta SCAEqList$$
$$GetEl : \beta SCAEqList \times N^2 \rightarrow \beta SCAEqList$$

Equations

$$a, b = a, b$$
$$hd([]) = []$$
$$hd(a, as) = a$$
$$tl([]) = []$$
$$tl(a, as) = as$$
$$GetEl([], i) = null$$
$$GetEl((\beta(i, j) = x, eqs), i, j) = \beta(i, j) = x$$
$$GetEl((\beta(y, z) = x, eqs), i, j) = GetEl(eqs, i, j)$$

End

A.3.4 dSCA β Function Operation List

Begin

Specification $\beta dSCAEqList$
Import $\beta dSCAEquation$
Constant Symbols []
Function Names

$$\_,\_ : \beta dSCAEquation \times \beta dSCAEqList \rightarrow \beta dSCAEqList$$
$$hd : \beta dSCAEqList \rightarrow \beta dSCAEquation$$
$$tl : \beta dSCAEqList \rightarrow \beta dSCAEqList$$
$$GetEl : \beta dSCAEqList \times N^3 \rightarrow \beta dSCAEqList$$

Equations

$$a, b = a, b$$
$$hd([]) = []$$
$$hd(a, as) = a$$
$$tl([]) = []$$
$$tl(a, as) = as$$
$$GetEl([], i) = null$$
$$GetEl((\beta_k(i, j) = x, eqs), i, j, k) = \beta_k(i, j) = x$$
$$GetEl((\beta_a(b, c) = x, eqs), i, j, k) = GetEl(eqs, i, j, k)$$

End

A.3.5 δ Function Operation List

Begin

Specification $\delta SCAEqList$
Import $\delta SCAEquation$
Constant Symbols []
Function Names

$$\_,\_ : \delta SCAEquation \times \delta SCAEqList \rightarrow \delta SCAEqList$$
$$hd : \delta SCAEqList \rightarrow \delta SCAEquation$$
$$tl : \delta SCAEqList \rightarrow \delta SCAEqList$$
$$GetEl : \delta SCAEqList \times N^2 \rightarrow \delta SCAEqList$$

Equations

$$a, b = a, b$$
$$hd([]) = []$$
$$hd(a, as) = a$$
$$tl([]) = []$$
$$tl(a, as) = as$$
$$GetEl([], i) = null$$
$$GetEl((\delta_{i,j}(t, a, x) = t', eqs), i, j) = \delta_{i,j}(t, a, x) = t'$$
$$GetEl((\delta_{b,c}(t, a, x) = t', eqs), i, j) = GetEl(eqs, i, j)$$

End

A.3.6 dSCA δ Function Operation List

Begin

Specification $\delta dSCAEqList$
Import $\delta dSCAEquation$
Constant Symbols []
Function Names

$$\_,\_ : \delta dSCAEquation \times \delta dSCAEqList \rightarrow \delta dSCAEqList$$
$$hd : \delta dSCAEqList \rightarrow \delta dSCAEquation$$
$$tl : \delta dSCAEqList \rightarrow \delta dSCAEqList$$
$$GetEl : \delta dSCAEqList \times N^3 \rightarrow \delta dSCAEqList$$

Equations

$$a, b = a, b$$
$$hd([]) = []$$
$$hd(a, as) = a$$
$$tl([]) = []$$
$$tl(a, as) = as$$
$$GetEl([], i) = null$$
$$GetEl((\delta_{i,j,k}(t, a, x) = t', eqs), i, j, k) = \delta_{i,j,k}(t, a, x) = t'$$
$$GetEl((\delta_{m,p,q}(t, a, x) = t', eqs), i, j, k) = GetEl(eqs, i)$$

End

A.3.7 Project Function Equation List

Begin

Specification $ProjEqList$
Import $ProjEquation$
Constant Symbols $[]$
Function Names

$$\_,\_ : ProjEquation \times ProjEqList \rightarrow ProjEqList$$
$$hd : ProjEqList \rightarrow ProjEquation$$
$$tl : ProjEqList \rightarrow ProjEqList$$
$$GetEl : ProjEqList \times N^3 \rightarrow ProjEqList$$

Equations

$$a, b = a, b$$
$$hd\,([]) = []$$
$$hd\,(a, as) = a$$
$$tl\,([]) = []$$
$$tl\,(a, as) = as$$
$$GetEl([], i) = null$$
$$GetEl\,((d(i,j,k) = t, eqs), i, j, k) = d(i,j,k) = t$$
$$GetEl\,((d(m,p,q) = t, eqs), i, j, k) = GetEl(eqs, i)$$

End

A.3.8 Map Function Equation List

Begin

Specification $MapEqList$
Import $MapEquation$
Constant Symbols $[]$
Function Names

$$\_,\_ : MapEquation \times MapEqList \rightarrow MapEqList$$
$$hd : MapEqList \rightarrow MapEquation$$
$$tl : MapEqList \rightarrow MapEqList$$
$$GetEl : MapEqList \times N^2 \rightarrow MapEqList$$

Equations

$$a, b = a, b$$
$$hd\,([]) = []$$
$$hd\,(a, as) = a$$
$$tl\,([]) = []$$
$$tl\,(a, as) = as$$
$$GetEl([], i) = null$$
$$GetEl\,((\Xi(i,j) = t, eqs), i, j) = \Xi(i,j) = t$$
$$GetEl\,((\Xi(m,n) = t, eqs), i, j) = GetEl(eqs, i)$$

End

A.3.9 SCA Initial State Equation List

Begin

Specification	$ISVEqList$
Import	$ISVEquation$
Constant Symbols	$[]$
Function Names	$\_,\_ : ISVEquation \times ISVEqList \rightarrow ISVEqList$
	$hd : ISVEqList \rightarrow ISVEquation$
	$tl : ISVEqList \rightarrow ISVEqList$
	$GetEl : ISVEqList \times N \rightarrow ISVEquation$
Equations	$a, b = a, b$
	$hd([]) = []$
	$hd(a, as) = a$
	$tl([]) = []$
	$tl(a, as) = as$
	$GetEl \quad ([], i) = null$
	$GetEl \quad ((V_i(0, a, x) = z, eqs), i)$
	$\quad = V_i(0, a, x) = z$
	$GetEl \quad ((V_a(0, a, x) = z, eqs), i)$
	$\quad = GetEl(eqs, i)$

End

A.3.10 dSCA Initial State Equation List

Begin

Specification	$dSCAISVEqList$
Import	$dSCAISVEquation$
Constant Symbols	$[]$
Function Names	
	$\_,\_ : dSCAISVEquation \times dSCAISVEqList \rightarrow dSCAISVEqList$
	$hd : dSCAISVEqList \rightarrow dSCAISVEquation$
	$tl : dSCAISVEqList \rightarrow dSCAISVEqList$
	$GetEl : dSCAISVEqList \times N \rightarrow dSCAISVEquation$
Equations	
	$a, b = a, b$
	$hd([]) = []$
	$hd(a, as) = a$
	$tl([]) = []$
	$tl(a, as) = as$
	$GetEl([], i) = null$
	$GetEl((V_i(0, a, x) = z, eqs), i) = V_i(0, a, x) = z$
	$GetEl((V_a(0, a, x) = z, eqs), i) = GetEl(eqs, i)$

End

A.3.11 SCA State Transition Equation List

Begin

Specification $STVEqList$
Import $STVEquation$
Constant Symbols $[]$
Function Names

$$\_,\_ : STVEquation \times STVEqList \rightarrow STVEqList$$
$$hd : STVEqList \rightarrow STVEquation$$
$$tl : STVEqList \rightarrow STVEqList$$
$$GetEl : STVEqList \times N \rightarrow STVEquation$$

Equations

$$a, b = a, b$$
$$hd([]) = []$$
$$hd(a, as) = a$$
$$tl([]) = []$$
$$tl(a, as) = as$$
$$GetEl([], i) = null$$
$$GetEl((V_i(t, a, x) = z, eqs), i) = V_i(t, a, x) = z$$
$$GetEl((V_a(t, a, x) = z, eqs), i) = GetEl(eqs, i)$$

End

A.3.12 dSCA State Transition Equation List

Begin

Specification $dSCASTVEqList$
Import $dSCAISVEquation$
Sorts
Constant Symbols $[]$
Function Names

$$\_,\_ : dSCAISVEquation \times STVFEqList \rightarrow dSCASTVEqList$$
$$hd : dSCASTVEqList \rightarrow dSCAISVEquation$$
$$tl : dSCASTVEqList \rightarrow dSCASTVEqList$$
$$GetEl : dSCASTVEqList \times N \rightarrow dSCAISVEquation$$

Equations

$$a, b = a, b$$
$$hd([]) = []$$
$$hd(a, as) = a$$
$$tl([]) = []$$
$$tl(a, as) = as$$
$$GetEl([], i) = null$$
$$GetEl((V_i(t, a, x) = z, eqs), i) = V_i(t, a, x) = z$$
$$GetEl((V_a(t, a, x) = z, eqs), i) = GetEl(eqs, i)$$

End

A.4 Equation Specifications

A.4.1 SCA State Transition Equation

Begin

Specification *STVEquation*

Import

Sorts

Constant Symbols

Function Names

$CreateVF : VFCallTerm \times VFOpTerm \rightarrow STVEquation$
$RetTerm : STVEquation \times N \rightarrow Term$
$RetIndex : STVEquation \times N \rightarrow N$

Equations

$CreateVF(t_1, t_2) = (t_1 = t_2)$
$RetTerm(V_n(t, a, x) = f_n, 1) = V_n(t, a, x)$
$RetTerm(V_n(t, a, x) = f_n, 2) = f_n$
$RetIndex(V_n(t, a, x) = z) = n$

End

A.4.2 dSCA State Transition Equation

Begin

Specification *dSCASTVEquation*

Import

Sorts

Constant Symbols

Function Names

$CreateVF : VFCallTerm \times VFOpTerm \rightarrow dSCASTVEquation$
$RetTerm : dSCASTVEquation \times N \rightarrow Term$
$RetIndex : dSCASTVEquation \times N \rightarrow N$

Equations

$CreateVF(t_1, t_2) = (t_1 = t_2)$
$RetTerm(V_n(t, a, x) = f_n, 1) = V_n(t, a, x)$
$RetTerm(V_n(t, a, x) = f_n, 2) = f_n$
$RetIndex(V_n(t, a, x) = z) = n$

End

A.4.3 SCA Initial State Equation

Begin

Specification $ISVEquation$

Import

Sorts

Constant Symbols

Function Names

$CreateVF : VFCallTerm \times VFOpTerm \rightarrow ISVEquation$
$RetTerm : ISVEquation \times N \rightarrow Term$
$RetIndex : ISVEquation \times N \rightarrow N$

Equations

$CreateVF(t_1, t_2) = (t_1 = t_2)$
$RetTerm(V_n(t, a, x) = f_n, 1) = V_n(t, a, x)$
$RetTerm(V_n(t, a, x) = f_n, 2) = f_n$
$RetIndex(V_n(t, a, x) = z) = n$

End

A.4.4 dSCA Initial State Equation

Begin

Specification $dSCAISVEquation$

Import

Sorts

Constant Symbols

Function Names

$CreateVF : VFCallTerm \times VFOpTerm \rightarrow dSCAISVEquation$
$RetTerm : dSCAISVEquation \times N \rightarrow Term$
$RetIndex : dSCAISVEquation \times N \rightarrow N$

Equations

$CreateVF(t_1, t_2) = (t_1 = t_2)$
$RetTerm(V_n(t, a, x) = f_n, 1) = V_n(t, a, x)$
$RetTerm(V_n(t, a, x) = f_n, 2) = f_n$
$RetIndex(V_n(t, a, x) = z) = n$

End

Appendix B

SCA Definition of GRCP

Begin

Specification	SCA
Import	M_A, T
Sorts	SCA_Algebra
Constant Symbols	[]
VF Function Names	$V_i : T \times M_A^n \times M_A^k \rightarrow M_A$
β Function Names	$\beta : N \times N \rightarrow N$
γ Function Names	$\gamma : N \times N \rightarrow \{S, M\}$
δ Function Names	$\delta_{i,j} : T \times M_A^n \times M_A^k \rightarrow T$

γ Equations

$$
\begin{array}{llll}
\gamma(1,1) = M, & \gamma(7,1) = M, & \gamma(15,1) = S, & \gamma(27,1) = M, \\
\gamma(1,2) = M, & \gamma(7,2) = M, & \gamma(15,2) = M, & \gamma(27,2) = M, \\
\gamma(1,3) = M, & \gamma(10,1) = M, & \gamma(22,1) = M, & \gamma(28,1) = M, \\
\gamma(2,1) = M, & \gamma(10,2) = M, & \gamma(22,2) = M, & \gamma(28,2) = M, \\
\gamma(2,2) = M, & \gamma(11,1) = S, & \gamma(23,1) = M, & \gamma(29,1) = S, \\
\gamma(4,1) = M, & \gamma(11,2) = M, & \gamma(23,2) = M, & \gamma(29,2) = S, \\
\gamma(4,2) = M, & \gamma(12,1) = M, & \gamma(24,1) = M, & \gamma(31,1) = S, \\
\gamma(4,3) = M, & \gamma(12,2) = M, & \gamma(24,2) = M, & \gamma(31,2) = S,
\end{array}
$$

β Equations

$$
\begin{array}{llll}
\gamma(5,1) = M, & \gamma(13,1) = S, & \gamma(25,1) = M, & \gamma(33,1) = S, \\
\gamma(5,2) = M, & \gamma(13,2) = M, & \gamma(25,2) = M, & \gamma(33,2) = S, \\
\gamma(6,1) = M, & \gamma(14,1) = M, & \gamma(26,1) = M, & \gamma(35,1) = S, \\
\gamma(6,2) = M, & \gamma(14,2) = M, & \gamma(26,2) = M, & \gamma(35,2) = S, \\
\beta(1,1) = 2, & \beta(7,1) = 14, & \beta(15,1) = 9, & \beta(27,1) = 33, \\
\beta(1,2) = 3, & \beta(7,2) = 15, & \beta(15,2) = 21, & \beta(27,2) = 34, \\
\beta(1,3) = 4, & \beta(10,1) = 22, & \beta(22,1) = 23, & \beta(28,1) = 35, \\
\beta(2,1) = 5, & \beta(10,2) = 16, & \beta(22,2) = 24, & \beta(28,2) = 36, \\
\beta(2,2) = 6, & \beta(11,1) = 9, & \beta(23,1) = 25, & \beta(29,1) = 1, \\
\beta(4,1) = 7, & \beta(11,2) = 17, & \beta(23,2) = 26, & \beta(29,2) = 2, \\
\beta(4,2) = 8, & \beta(12,1) = 22, & \beta(24,1) = 27, & \beta(31,1) = 3, \\
\beta(4,3) = 9, & \beta(12,2) = 18, & \beta(24,2) = 28, & \beta(31,2) = 4, \\
\beta(5,1) = 10, & \beta(13,1) = 9, & \beta(25,1) = 29, & \beta(33,1) = 5, \\
\beta(5,2) = 11, & \beta(13,2) = 19, & \beta(25,2) = 30, & \beta(33,2) = 6, \\
\beta(6,1) = 12, & \beta(14,1) = 22, & \beta(26,1) = 31, & \beta(35,1) = 7, \\
\beta(6,2) = 13, & \beta(14,2) = 20, & \beta(26,2) = 32, & \beta(35,2) = 8,
\end{array}
$$

δ Equations

$$
\begin{array}{lll}
\delta_{1,1}(t,a,x) = t-1, & \delta_{11,1}(t,a,x) = t-1, & \delta_{25,1}(t,a,x) = t-1, \\
\delta_{1,2}(t,a,x) = t-1, & \delta_{11,2}(t,a,x) = t-1, & \delta_{25,2}(t,a,x) = t-1, \\
\delta_{1,3}(t,a,x) = t-1, & \delta_{12,1}(t,a,x) = t-1, & \delta_{26,1}(t,a,x) = t-1, \\
\delta_{2,1}(t,a,x) = t-1, & \delta_{12,2}(t,a,x) = t-1, & \delta_{26,2}(t,a,x) = t-1, \\
\delta_{2,2}(t,a,x) = t-1, & \delta_{13,1}(t,a,x) = t-1, & \delta_{27,1}(t,a,x) = t-1, \\
\delta_{4,1}(t,a,x) = t-1, & \delta_{13,2}(t,a,x) = t-1, & \delta_{27,2}(t,a,x) = t-1, \\
\delta_{4,2}(t,a,x) = t-1, & \delta_{14,1}(t,a,x) = t-1, & \delta_{28,1}(t,a,x) = t-1, \\
\delta_{4,3}(t,a,x) = t-1, & \delta_{14,2}(t,a,x) = t-1, & \delta_{28,2}(t,a,x) = t-1, \\
\delta_{5,1}(t,a,x) = t-1, & \delta_{15,1}(t,a,x) = t-1, & \delta_{29,1}(t,a,x) = t-1, \\
\delta_{5,2}(t,a,x) = t-1, & \delta_{15,2}(t,a,x) = t-1, & \delta_{29,2}(t,a,x) = t-1, \\
\delta_{6,1}(t,a,x) = t-1, & \delta_{22,1}(t,a,x) = t-1, & \delta_{31,1}(t,a,x) = t-1, \\
\delta_{6,2}(t,a,x) = t-1, & \delta_{22,2}(t,a,x) = t-1, & \delta_{31,2}(t,a,x) = t-1, \\
\delta_{7,1}(t,a,x) = t-1, & \delta_{23,1}(t,a,x) = t-1, & \delta_{33,1}(t,a,x) = t-1, \\
\delta_{7,2}(t,a,x) = t-1, & \delta_{23,2}(t,a,x) = t-1, & \delta_{33,2}(t,a,x) = t-1, \\
\delta_{10,1}(t,a,x) = t-1, & \delta_{24,1}(t,a,x) = t-1, & \delta_{35,1}(t,a,x) = t-1, \\
\delta_{10,2}(t,a,x) = t-1, & \delta_{24,2}(t,a,x) = t-1, & \delta_{35,2}(t,a,x) = t-1
\end{array}
$$

IV Equations

$$
\begin{array}{lll}
V_1(0,a,x) = stay & V_2(0,a,x) = true & V_3(0,a,x) = stay \\
V_4(0,a,x) = up & V_5(0,a,x) = true & V_6(0,a,x) = false \\
V_7(0,a,x) = false & V_8(0,a,x) = down & V_9(0,a,x) = up \\
V_{10}(0,a,x) = true & V_{11}(0,a,x) = true & V_{12}(0,a,x) = false \\
V_{13}(0,a,x) = false & V_{14}(0,a,x) = false & V_{15}(0,a,x) = true \\
V_{16}(0,a,x) = false & V_{17}(0,a,x) = 90 & V_{18}(0,a,x) = true \\
V_{19}(0,a,x) = 0 & V_{20}(0,a,x) = true & V_{21}(0,a,x) = 0 \\
V_{22}(0,a,x) = false & V_{23}(0,a,x) = false & V_{24}(0,a,x) = false \\
V_{25}(0,a,x) = false & V_{26}(0,a,x) = false & V_{27}(0,a,x) = false \\
V_{28}(0,a,x) = false & V_{29}(0,a,x) = 0 & V_{30}(0,a,x) = 0 \\
V_{31}(0,a,x) = 0 & V_{32}(0,a,x) = 0 & V_{33}(0,a,x) = 0 \\
V_{34}(0,a,x) = 0 & V_{35}(0,a,x) = 0 & V_{36}(0,a,x) = 0
\end{array}
$$

ST Equations

$$
\begin{array}{l}
V_1(t+1,a,x) = cond(V_2(t,a,x), V_3(t,a,x), V_4(t,a,x)), \\
V_2(t+1,a,x) = or(V_5(t,a,x), V_6(t,a,x)), \\
V_3(t+1,a,x) = stay, \\
V_4(t+1,a,x) = cond(V_7(t,a,x), V_8(t,a,x), V_9(t,a,x)), \\
V_5(t+1,a,x) = and(V_{10}(t,a,x), V_{11}(t,a,x)), \\
V_6(t+1,a,x) = and(V_{12}(t,a,x), V_{13}(t,a,x)), \\
V_7(t+1,a,x) = and(V_{14}(t,a,x), V_{15}(t,a,x)), \\
V_8(t+1,a,x) = down, \\
V_9(t+1,a,x) = up, \\
V_{10}(t+1,a,x) = eq(V_{22}(t,a,x), V_{16}(t,a,x)), \\
V_{11}(t+1,a,x) = eq(a_9(t), V_{17}(t,a,x)), \\
V_{12}(t+1,a,x) = eq(V_{22}(t,a,x), V_{18}(t,a,x)), \\
V_{13}(t+1,a,x) = eq(a_9(t), V_{19}(t,a,x)), \\
V_{14}(t+1,a,x) = eq(V_{22}(t,a,x), V_{20}(t,a,x)), \\
V_{15}(t+1,a,x) = gt(a_9(t), V_{21}(t,a,x)), \\
V_{16}(t+1,a,x) = false, \\
V_{17}(t+1,a,x) = 90, \\
V_{18}(t+1,a,x) = true, \\
V_{19}(t+1,a,x) = 0,
\end{array}
$$

$$
\begin{aligned}
&V_{20}(t+1,a,x) = true,\\
&V_{21}(t+1,a,x) = 0,\\
&V_{22}(t+1,a,x) = or(V_{23}(t,a,x), V_{24}(t,a,x)),\\
&V_{23}(t+1,a,x) = or(V_{25}(t,a,x), V_{26}(t,a,x)),\\
&V_{24}(t+1,a,x) = or(V_{27}(t,a,x), V_{28}(t,a,x)),\\
&V_{25}(t+1,a,x) = gt(V_{29}(t,a,x), V_{30}(t,a,x)),\\
&V_{26}(t+1,a,x) = gt(V_{31}(t,a,x), V_{32}(t,a,x)),\\
&V_{27}(t+1,a,x) = gt(V_{33}(t,a,x), V_{34}(t,a,x)),\\
&V_{28}(t+1,a,x) = gt(V_{35}(t,a,x), V_{36}(t,a,x)),\\
&V_{29}(t+1,a,x) = sub(a_1(t), a_2(t)),\\
&V_{30}(t+1,a,x) = 0,\\
&V_{31}(t+1,a,x) = sub(a_3(t), a_4(t)),\\
&V_{32}(t+1,a,x) = 0,\\
&V_{33}(t+1,a,x) = sub(a_5(t), a_6(t)),\\
&V_{34}(t+1,a,x) = 0,\\
&V_{35}(t+1,a,x) = sub(a_7(t), a_8(t)),
\end{aligned}
$$

End

Appendix C

Abstract dSCA Definition of GRCP (Form 1)

Begin

Specification	acvSCA
Import	M_A, T
Sorts	SCA_Algebra
Constant Symbols	
VF Function Names	$V_i : T \times M_A^n \times M_A^k \rightarrow M_A$
β **Function Names**	$\beta_{pc} : N \times N \rightarrow N$
γ **Function Names**	$\gamma_{pc} : N \times N \rightarrow \{S, M\}$
δ **Function Names**	$\delta_{i,j,pc} : T \times M_A^n \times M_A^k \rightarrow T$
γ **Equations**	

$$
\begin{array}{llll}
\gamma_0(1,0) = M, & \gamma_0(1,1) = M, & \gamma_0(1,2) = M, & \gamma_0(1,3) = M, \\
\gamma_0(2,0) = M, & \gamma_0(2,1) = M, & \gamma_0(2,2) = M, & \gamma_0(2,3) = U, \\
\gamma_0(3,0) = M, & \gamma_0(3,1) = U, & \gamma_0(3,2) = U, & \gamma_0(3,3) = U, \\
\gamma_0(4,0) = M, & \gamma_0(4,1) = M, & \gamma_0(4,2) = M, & \gamma_0(4,3) = M, \\
\gamma_0(5,0) = M, & \gamma_0(5,1) = M, & \gamma_0(5,2) = M, & \gamma_0(5,3) = U, \\
\gamma_0(6,0) = M, & \gamma_0(6,1) = M, & \gamma_0(6,2) = M, & \gamma_0(6,3) = U, \\
\gamma_0(7,0) = M, & \gamma_0(7,1) = M, & \gamma_0(7,2) = M, & \gamma_0(7,3) = U, \\
\gamma_0(8,0) = M, & \gamma_0(8,1) = U, & \gamma_0(8,2) = U, & \gamma_0(8,3) = U, \\
\gamma_0(9,0) = M, & \gamma_0(9,1) = U, & \gamma_0(9,2) = U, & \gamma_0(9,3) = U, \\
\gamma_0(10,0) = M, & \gamma_0(10,1) = M, & \gamma_0(10,2) = M, & \gamma_0(10,3) = U, \\
\gamma_0(11,0) = M, & \gamma_0(11,1) = S, & \gamma_0(11,2) = M, & \gamma_0(11,3) = U, \\
\gamma_0(12,0) = M, & \gamma_0(12,1) = M & \gamma_0(12,2) = M, & \gamma_0(12,3) = U, \\
\gamma_0(13,0) = M, & \gamma_0(13,1) = S, & \gamma_0(13,2) = M, & \gamma_0(13,3) = U, \\
\gamma_0(14,0) = M, & \gamma_0(14,1) = M, & \gamma_0(14,2) = M, & \gamma_0(14,3) = U, \\
\gamma_0(15,0) = M, & \gamma_0(15,1) = S, & \gamma_0(15,2) = M, & \gamma_0(15,3) = U, \\
\gamma_0(16,0) = M, & \gamma_0(16,1) = U, & \gamma_0(16,2) = U, & \gamma_0(16,3) = U, \\
\gamma_0(17,0) = M, & \gamma_0(17,1) = U, & \gamma_0(17,2) = U, & \gamma_0(17,3) = U, \\
\gamma_0(18,0) = M, & \gamma_0(18,1) = U, & \gamma_0(18,2) = U, & \gamma_0(18,3) = U, \\
\gamma_0(19,0) = M, & \gamma_0(19,1) = U, & \gamma_0(19,2) = U, & \gamma_0(19,3) = U, \\
\gamma_0(20,0) = M, & \gamma_0(20,1) = U, & \gamma_0(20,2) = U, & \gamma_0(20,3) = U, \\
\gamma_0(21,0) = M, & \gamma_0(21,1) = U, & \gamma_0(21,2) = U, & \gamma_0(21,3) = U, \\
\gamma_0(22,0) = M, & \gamma_0(22,1) = M, & \gamma_0(22,2) = M, & \gamma_0(22,3) = U,
\end{array}
$$

$\gamma_0(23,0) = M, \quad \gamma_0(23,1) = M, \quad \gamma_0(23,2) = M, \quad \gamma_0(23,3) = U,$
$\gamma_0(24,0) = M, \quad \gamma_0(24,1) = M, \quad \gamma_0(24,2) = M, \quad \gamma_0(24,3) = U,$
$\gamma_0(25,0) = M, \quad \gamma_0(25,1) = M, \quad \gamma_0(25,2) = M, \quad \gamma_0(25,3) = U,$
$\gamma_0(26,0) = M, \quad \gamma_0(26,1) = M, \quad \gamma_0(26,2) = M, \quad \gamma_0(26,3) = U,$
$\gamma_0(27,0) = M, \quad \gamma_0(27,1) = M, \quad \gamma_0(27,2) = M, \quad \gamma_0(27,3) = U,$
$\gamma_0(28,0) = M, \quad \gamma_0(28,1) = M, \quad \gamma_0(28,2) = M, \quad \gamma_0(28,3) = U,$
$\gamma_0(29,0) = M, \quad \gamma_0(29,1) = S, \quad \gamma_0(29,2) = S, \quad \gamma_0(29,3) = U.$
$\gamma_0(30,0) = M, \quad \gamma_0(30,1) = U, \quad \gamma_0(30,2) = U, \quad \gamma_0(30,3) = U,$
$\gamma_0(31,0) = M, \quad \gamma_0(31,1) = S, \quad \gamma_0(31,2) = S, \quad \gamma_0(31,3) = U,$
$\gamma_0(32,0) = M, \quad \gamma_0(32,1) = U, \quad \gamma_0(32,2) = U, \quad \gamma_0(32,3) = U,$
$\gamma_0(33,0) = M, \quad \gamma_0(33,1) = S, \quad \gamma_0(33,2) = S, \quad \gamma_0(33,3) = U,$
$\gamma_0(34,0) = M, \quad \gamma_0(34,1) = U, \quad \gamma_0(34,2) = U, \quad \gamma_0(34,3) = U,$
$\gamma_0(35,0) = M, \quad \gamma_0(35,1) = S, \quad \gamma_0(35,2) = S, \quad \gamma_0(35,3) = U,$
$\gamma_0(36,0) = M, \quad \gamma_0(36,1) = U, \quad \gamma_0(36,2) = U, \quad \gamma_0(36,3) = U,$
$\gamma_0(pc,0) = M$

β Equations

$\beta_0(1,0) = pc, \quad \beta_0(1,1) = 2, \quad \beta_0(1,2) = 3, \quad \beta_0(1,3) = 4,$
$\beta_0(2,0) = pc, \quad \beta_0(2,1) = 5, \quad \beta_0(2,2) = 6, \quad \beta_0(2,3) = \omega,$
$\beta_0(3,0) = pc, \quad \beta_0(3,1) = \omega, \quad \beta_0(3,2) = \omega, \quad \beta_0(3,3) = \omega,$
$\beta_0(4,0) = pc, \quad \beta_0(4,1) = 7, \quad \beta_0(4,2) = 8, \quad \beta_0(4,3) = 9,$
$\beta_0(5,0) = pc, \quad \beta_0(5,1) = 10, \quad \beta_0(5,2) = 11, \quad \beta_0(5,3) = \omega,$
$\beta_0(6,0) = pc, \quad \beta_0(6,1) = 12, \quad \beta_0(6,2) = 13, \quad \beta_0(6,3) = \omega,$
$\beta_0(7,0) = pc, \quad \beta_0(7,1) = 14, \quad \beta_0(7,2) = 15, \quad \beta_0(7,3) = \omega,$
$\beta_0(8,0) = pc, \quad \beta_0(8,1) = \omega, \quad \beta_0(8,2) = \omega, \quad \beta_0(8,3) = \omega,$
$\beta_0(9,0) = pc, \quad \beta_0(9,1) = \omega, \quad \beta_0(9,2) = \omega, \quad \beta_0(9,3) = \omega,$
$\beta_0(10,0) = pc, \quad \beta_0(10,1) = 22, \quad \beta_0(10,2) = 16, \quad \beta_0(10,3) = \omega,$
$\beta_0(11,0) = pc, \quad \beta_0(11,1) = 9, \quad \beta_0(11,2) = 17, \quad \beta_0(11,3) = \omega,$
$\beta_0(12,0) = pc, \quad \beta_0(12,1) = 22, \quad \beta_0(12,2) = 18, \quad \beta_0(12,3) = \omega,$
$\beta_0(13,0) = pc, \quad \beta_0(13,1) = 9, \quad \beta_0(13,2) = 19, \quad \beta_0(13,3) = \omega,$
$\beta_0(14,0) = pc, \quad \beta_0(14,1) = 22, \quad \beta_0(14,2) = 20, \quad \beta_0(14,3) = \omega,$
$\beta_0(15,0) = pc, \quad \beta_0(15,1) = 9, \quad \beta_0(15,2) = 21, \quad \beta_0(15,3) = \omega,$
$\beta_0(16,0) = pc, \quad \beta_0(16,1) = \omega, \quad \beta_0(16,2) = \omega, \quad \beta_0(16,3) = \omega,$
$\beta_0(17,0) = pc, \quad \beta_0(17,1) = \omega, \quad \beta_0(17,2) = \omega, \quad \beta_0(17,3) = \omega,$
$\beta_0(18,0) = pc, \quad \beta_0(18,1) = \omega, \quad \beta_0(18,2) = \omega, \quad \beta_0(18,3) = \omega,$
$\beta_0(19,0) = pc, \quad \beta_0(19,1) = \omega, \quad \beta_0(19,2) = \omega, \quad \beta_0(19,3) = \omega,$
$\beta_0(20,0) = pc, \quad \beta_0(20,1) = \omega, \quad \beta_0(20,2) = \omega, \quad \beta_0(20,3) = \omega,$
$\beta_0(21,0) = pc, \quad \beta_0(21,1) = \omega, \quad \beta_0(21,2) = \omega, \quad \beta_0(21,3) = \omega,$
$\beta_0(22,0) = pc, \quad \beta_0(22,1) = 23, \quad \beta_0(22,2) = 24, \quad \beta_0(22,3) = \omega,$
$\beta_0(23,0) = pc, \quad \beta_0(23,1) = 25, \quad \beta_0(23,2) = 26, \quad \beta_0(23,3) = \omega,$
$\beta_0(24,0) = pc, \quad \beta_0(24,1) = 27, \quad \beta_0(24,2) = 28, \quad \beta_0(24,3) = \omega,$
$\beta_0(25,0) = pc, \quad \beta_0(25,1) = 29, \quad \beta_0(25,2) = 30, \quad \beta_0(25,3) = \omega,$
$\beta_0(26,0) = pc, \quad \beta_0(26,1) = 31, \quad \beta_0(26,2) = 32, \quad \beta_0(26,3) = \omega,$
$\beta_0(27,0) = pc, \quad \beta_0(27,1) = 33, \quad \beta_0(27,2) = 34, \quad \beta_0(27,3) = \omega,$
$\beta_0(28,0) = pc, \quad \beta_0(28,1) = 35, \quad \beta_0(28,2) = 36, \quad \beta_0(28,3) = \omega,$
$\beta_0(29,0) = pc, \quad \beta_0(29,1) = 1, \quad \beta_0(29,2) = 2, \quad \beta_0(29,3) = \omega.$
$\beta_0(30,0) = pc, \quad \beta_0(30,1) = \omega, \quad \beta_0(30,2) = \omega, \quad \beta_0(30,3) = \omega,$
$\beta_0(31,0) = pc, \quad \beta_0(31,1) = 3, \quad \beta_0(31,2) = 4, \quad \beta_0(31,3) = \omega,$
$\beta_0(32,0) = pc, \quad \beta_0(32,1) = \omega, \quad \beta_0(32,2) = \omega, \quad \beta_0(32,3) = \omega,$
$\beta_0(33,0) = pc, \quad \beta_0(33,1) = 5, \quad \beta_0(33,2) = 6, \quad \beta_0(33,3) = \omega,$
$\beta_0(34,0) = pc, \quad \beta_0(34,1) = \omega, \quad \beta_0(34,2) = \omega, \quad \beta_0(34,3) = \omega,$
$\beta_0(35,0) = pc, \quad \beta_0(35,1) = 7, \quad \beta_0(35,2) = 8, \quad \beta_0(35,3) = \omega,$
$\beta_0(36,0) = pc, \quad \beta_0(36,1) = \omega, \quad \beta_0(36,2) = \omega, \quad \beta_0(36,3) = \omega,$
$\beta_0(pc,1) = pc$

δ Equations

$$\delta_{1,0,0}(t,a,x) = t-1, \quad \delta_{1,1,0}(t,a,x) = t-1, \quad \delta_{1,2,0}(t,a,x) = t-1,$$
$$\delta_{1,3,0}(t,a,x) = t-1,$$
$$\delta_{2,0,0}(t,a,x) = t-1, \quad \delta_{2,1,0}(t,a,x) = t-1, \quad \delta_{2,2,0}(t,a,x) = t-1,$$
$$\delta_{2,3,0}(t,a,x) = t-1,$$
$$\delta_{3,0,0}(t,a,x) = t-1, \quad \delta_{3,1,0}(t,a,x) = t-1, \quad \delta_{3,2,0}(t,a,x) = t-1,$$
$$\delta_{3,3,0}(t,a,x) = t-1,$$
$$\delta_{4,0,0}(t,a,x) = t-1, \quad \delta_{4,1,0}(t,a,x) = t-1, \quad \delta_{4,2,0}(t,a,x) = t-1,$$
$$\delta_{4,3,0}(t,a,x) = t-1,$$
$$\delta_{5,0,0}(t,a,x) = t-1, \quad \delta_{5,1,0}(t,a,x) = t-1, \quad \delta_{5,2,0}(t,a,x) = t-1,$$
$$\delta_{5,3,0}(t,a,x) = t-1,$$
$$\delta_{6,0,0}(t,a,x) = t-1, \quad \delta_{6,1,0}(t,a,x) = t-1, \quad \delta_{6,2,0}(t,a,x) = t-1,$$
$$\delta_{6,3,0}(t,a,x) = t-1,$$
$$\delta_{7,0,0}(t,a,x) = t-1, \quad \delta_{7,1,0}(t,a,x) = t-1, \quad \delta_{7,2,0}(t,a,x) = t-1,$$
$$\delta_{7,3,0}(t,a,x) = t-1,$$
$$\delta_{8,0,0}(t,a,x) = t-1, \quad \delta_{8,1,0}(t,a,x) = t-1, \quad \delta_{8,2,0}(t,a,x) = t-1,$$
$$\delta_{8,3,0}(t,a,x) = t-1,$$
$$\delta_{9,0,0}(t,a,x) = t-1, \quad \delta_{9,1,0}(t,a,x) = t-1, \quad \delta_{9,2,0}(t,a,x) = t-1,$$
$$\delta_{9,3,0}(t,a,x) = t-1,$$
$$\delta_{10,0,0}(t,a,x) = t-1, \quad \delta_{10,1,0}(t,a,x) = t-1, \quad \delta_{10,2,0}(t,a,x) = t-1,$$
$$\delta_{10,3,0}(t,a,x) = t-1,$$
$$\delta_{11,0,0}(t,a,x) = t-1, \quad \delta_{11,1,0}(t,a,x) = t-1, \quad \delta_{11,2,0}(t,a,x) = t-1,$$
$$\delta_{11,3,0}(t,a,x) = t-1,$$
$$\delta_{12,0,0}(t,a,x) = t-1, \quad \delta_{12,1,0}(t,a,x) = t-1 \quad \delta_{12,2,0}(t,a,x) = t-1,$$
$$\delta_{12,3,0}(t,a,x) = t-1,$$
$$\delta_{13,0,0}(t,a,x) = t-1, \quad \delta_{13,1,0}(t,a,x) = t-1, \quad \delta_{13,2,0}(t,a,x) = t-1,$$
$$\delta_{13,3,0}(t,a,x) = t-1,$$
$$\delta_{14,0,0}(t,a,x) = t-1, \quad \delta_{14,1,0}(t,a,x) = t-1, \quad \delta_{14,2,0}(t,a,x) = t-1,$$
$$\delta_{14,3,0}(t,a,x) = t-1,$$
$$\delta_{15,0,0}(t,a,x) = t-1, \quad \delta_{15,1,0}(t,a,x) = t-1, \quad \delta_{15,2,0}(t,a,x) = t-1,$$
$$\delta_{15,3,0}(t,a,x) = t-1,$$
$$\delta_{16,0,0}(t,a,x) = t-1, \quad \delta_{16,1,0}(t,a,x) = t-1, \quad \delta_{16,2,0}(t,a,x) = t-1,$$
$$\delta_{16,3,0}(t,a,x) = t-1,$$
$$\delta_{17,0,0}(t,a,x) = t-1, \quad \delta_{17,1,0}(t,a,x) = t-1, \quad \delta_{17,2,0}(t,a,x) = t-1,$$
$$\delta_{17,3,0}(t,a,x) = t-1,$$
$$\delta_{18,0,0}(t,a,x) = t-1, \quad \delta_{18,1,0}(t,a,x) = t-1, \quad \delta_{18,2,0}(t,a,x) = t-1,$$
$$\delta_{18,3,0}(t,a,x) = t-1,$$
$$\delta_{19,0,0}(t,a,x) = t-1, \quad \delta_{19,1,0}(t,a,x) = t-1, \quad \delta_{19,2,0}(t,a,x) = t-1,$$
$$\delta_{19,3,0}(t,a,x) = t-1,$$
$$\delta_{20,0,0}(t,a,x) = t-1, \quad \delta_{20,1,0}(t,a,x) = t-1, \quad \delta_{20,2,0}(t,a,x) = t-1,$$
$$\delta_{20,3,0}(t,a,x) = t-1,$$
$$\delta_{21,0,0}(t,a,x) = t-1, \quad \delta_{21,1,0}(t,a,x) = t-1, \quad \delta_{21,2,0}(t,a,x) = t-1,$$
$$\delta_{21,3,0}(t,a,x) = t-1,$$
$$\delta_{22,0,0}(t,a,x) = t-1, \quad \delta_{22,1,0}(t,a,x) = t-1, \quad \delta_{22,2,0}(t,a,x) = t-1,$$
$$\delta_{22,3,0}(t,a,x) = t-1,$$
$$\delta_{23,0,0}(t,a,x) = t-1, \quad \delta_{23,1,0}(t,a,x) = t-1, \quad \delta_{23,2,0}(t,a,x) = t-1,$$
$$\delta_{23,3,0}(t,a,x) = t-1,$$
$$\delta_{24,0,0}(t,a,x) = t-1, \quad \delta_{24,1,0}(t,a,x) = t-1, \quad \delta_{24,2,0}(t,a,x) = t-1,$$
$$\delta_{24,3,0}(t,a,x) = t-1,$$
$$\delta_{25,0,0}(t,a,x) = t-1, \quad \delta_{25,1,0}(t,a,x) = t-1, \quad \delta_{25,2,0}(t,a,x) = t-1,$$
$$\delta_{25,3,0}(t,a,x) = t-1,$$

$\delta_{26,0,0}(t,a,x) = t-1, \quad \delta_{26,1,0}(t,a,x) = t-1, \quad \delta_{26,2,0}(t,a,x) = t-1,$
$\delta_{26,3,0}(t,a,x) = t-1,$
$\delta_{27,0,0}(t,a,x) = t-1, \quad \delta_{27,1,0}(t,a,x) = t-1, \quad \delta_{27,2,0}(t,a,x) = t-1,$
$\delta_{27,3,0}(t,a,x) = t-1,$
$\delta_{28,0,0}(t,a,x) = t-1, \quad \delta_{28,1,0}(t,a,x) = t-1, \quad \delta_{28,2,0}(t,a,x) = t-1,$
$\delta_{28,3,0}(t,a,x) = t-1,$
$\delta_{29,0,0}(t,a,x) = t-1, \quad \delta_{29,1,0}(t,a,x) = t-1, \quad \delta_{29,2,0}(t,a,x) = t-1,$
$\delta_{29,3,0}(t,a,x) = t-1.$
$\delta_{30,0,0}(t,a,x) = t-1, \quad \delta_{30,1,0}(t,a,x) = t-1, \quad \delta_{30,2,0}(t,a,x) = t-1,$
$\delta_{30,3,0}(t,a,x) = t-1,$
$\delta_{31,0,0}(t,a,x) = t-1, \quad \delta_{31,1,0}(t,a,x) = t-1, \quad \delta_{31,2,0}(t,a,x) = t-1,$
$\delta_{31,3,0}(t,a,x) = t-1,$
$\delta_{32,0,0}(t,a,x) = t-1, \quad \delta_{32,1,0}(t,a,x) = t-1, \quad \delta_{32,2,0}(t,a,x) = t-1,$
$\delta_{32,3,0}(t,a,x) = t-1,$
$\delta_{33,0,0}(t,a,x) = t-1, \quad \delta_{33,1,0}(t,a,x) = t-1, \quad \delta_{33,2,0}(t,a,x) = t-1,$
$\delta_{33,3,0}(t,a,x) = t-1,$
$\delta_{34,0,0}(t,a,x) = t-1, \quad \delta_{34,1,0}(t,a,x) = t-1, \quad \delta_{34,2,0}(t,a,x) = t-1,$
$\delta_{34,3,0}(t,a,x) = t-1,$
$\delta_{35,0,0}(t,a,x) = t-1, \quad \delta_{35,1,0}(t,a,x) = t-1, \quad \delta_{35,2,0}(t,a,x) = t-1,$
$\delta_{35,3,0}(t,a,x) = t-1,$
$\delta_{36,0,0}(t,a,x) = t-1, \quad \delta_{36,1,0}(t,a,x) = t-1, \quad \delta_{36,2,0}(t,a,x) = t-1,$
$\delta_{36,3,0}(t,a,x) = t-1,$
$\delta_{pc,0,0}(t,a,x) = t-1$

IV Equations

$V_1(0,a,x) = stay \quad V_2(0,a,x) = true \quad V_3(0,a,x) = stay$
$V_4(0,a,x) = up \quad V_5(0,a,x) = true \quad V_6(0,a,x) = false$
$V_7(0,a,x) = false \quad V_8(0,a,x) = down \quad V_9(0,a,x) = up$
$V_{10}(0,a,x) = true \quad V_{11}(0,a,x) = true \quad V_{12}(0,a,x) = false$
$V_{13}(0,a,x) = false \quad V_{14}(0,a,x) = false \quad V_{15}(0,a,x) = true$
$V_{16}(0,a,x) = false \quad V_{17}(0,a,x) = 90 \quad V_{18}(0,a,x) = true$
$V_{19}(0,a,x) = 0 \quad V_{20}(0,a,x) = true \quad V_{21}(0,a,x) = 0$
$V_{22}(0,a,x) = false \quad V_{23}(0,a,x) = false \quad V_{24}(0,a,x) = false$
$V_{25}(0,a,x) = false \quad V_{26}(0,a,x) = false \quad V_{27}(0,a,x) = false$
$V_{28}(0,a,x) = false \quad V_{29}(0,a,x) = 0 \quad V_{30}(0,a,x) = 0$
$V_{31}(0,a,x) = 0 \quad V_{32}(0,a,x) = 0 \quad V_{33}(0,a,x) = 0$
$V_{34}(0,a,x) = 0 \quad V_{35}(0,a,x) = 0 \quad V_{36}(0,a,x) = 0$
$V_{pc}(0,a,x) = 0$

ST Equations

$$
\begin{array}{ll}
V_1(t+1,a,x) = cond\begin{pmatrix} V_2(t,a,x), \\ V_3(t,a,x), \\ V_4(t,a,x) \end{pmatrix} & \text{if } V_{pc}(t,a,x) = 0, \\
V_2(t+1,a,x) = or(V_5(t,a,x), V_6(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_3(t+1,a,x) = start & \text{if } V_{pc}(t,a,x) = 0, \\
V_4(t+1,a,x) = cond\begin{pmatrix} V_7(t,a,x), \\ V_8(t,a,x), \\ V_9(t,a,x) \end{pmatrix} & \text{if } V_{pc}(t,a,x) = 0, \\
V_5(t+1,a,x) = and(V_{10}(t,a,x), V_{11}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_6(t+1,a,x) = and(V_{12}(t,a,x), V_{13}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_7(t+1,a,x) = and(V_{14}(t,a,x), V_{15}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_8(t+1,a,x) = down & \text{if } V_{pc}(t,a,x) = 0, \\
V_9(t+1,a,x) = up & \text{if } V_{pc}(t,a,x) = 0, \\
V_{10}(t+1,a,x) = eq(V_{22}(t,a,x), V_{16}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{11}(t+1,a,x) = eq(a_9(t), V_{17}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{12}(t+1,a,x) = eq(V_{22}(t,a,x), V_{18}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{13}(t+1,a,x) = eq(a_9(t), V_{19}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{14}(t+1,a,x) = eq(V_{22}(t,a,x), V_{20}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{15}(t+1,a,x) = gt(a_9(t), V_{21}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{16}(t+1,a,x) = false & \text{if } V_{pc}(t,a,x) = 0, \\
V_{17}(t+1,a,x) = 90 & \text{if } V_{pc}(t,a,x) = 0, \\
V_{18}(t+1,a,x) = true & \text{if } V_{pc}(t,a,x) = 0, \\
V_{19}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0, \\
V_{20}(t+1,a,x) = true & \text{if } V_{pc}(t,a,x) = 0, \\
V_{21}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0, \\
V_{22}(t+1,a,x) = or(V_{23}(t,a,x), V_{24}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{23}(t+1,a,x) = or(V_{25}(t,a,x), V_{26}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{24}(t+1,a,x) = or(V_{27}(t,a,x), V_{28}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{25}(t+1,a,x) = gt(V_{29}(t,a,x), V_{30}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{26}(t+1,a,x) = gt(V_{31}(t,a,x), V_{32}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{27}(t+1,a,x) = gt(V_{33}(t,a,x), V_{34}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{28}(t+1,a,x) = gt(V_{35}(t,a,x), V_{36}(t,a,x)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{29}(t+1,a,x) = sub(a_1(t), a_2(t)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{30}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0, \\
V_{31}(t+1,a,x) = sub(a_3(t), a_4(t)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{32}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0, \\
V_{33}(t+1,a,x) = sub(a_5(t), a_6(t)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{34}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0, \\
V_{35}(t+1,a,x) = sub(a_7(t), a_8(t)) & \text{if } V_{pc}(t,a,x) = 0, \\
V_{36}(t+1,a,x) = 0 & \text{if } V_{pc}(t,a,x) = 0, \\
V_{pc}(t+1,a,x) = mod(add(V_{pc}(t,a,x), 1), 36) &
\end{array}
$$

End

Appendix D

Abstract dSCA Definition of GRCP (Form 2)

Begin

Specification	acvSCA
Import	M_A, T
Sorts	SCA_Algebra
Constant Symbols	
VF Function Names	$V_i : T \times M_A^n \times M_A^k \rightarrow M_A$
β **Function Names**	$\beta_{pc} : N \times N \rightarrow N$
γ **Function Names**	$\gamma_{pc} : N \times N \rightarrow \{S, M\}$
δ **Function Names**	$\delta_{i,j,pc} : T \times M_A^n \times M_A^k \rightarrow T$
γ **Equations**	

$$
\begin{array}{llll}
\gamma_0(pc,0) = M, & \gamma_9(pc,0) = M, & \gamma_{18}(pc,0) = M, & \gamma_{27}(pc,0) = M, \\
\gamma_1(pc,0) = M, & \gamma_{10}(pc,0) = M, & \gamma_{19}(pc,0) = M, & \gamma_{28}(pc,0) = M, \\
\gamma_2(pc,0) = M, & \gamma_{11}(pc,0) = M, & \gamma_{20}(pc,0) = M, & \gamma_{29}(pc,0) = M, \\
\gamma_3(pc,0) = M, & \gamma_{12}(pc,0) = M, & \gamma_{21}(pc,0) = M, & \gamma_{30}(pc,0) = M, \\
\gamma_4(pc,0) = M, & \gamma_{13}(pc,0) = M, & \gamma_{22}(pc,0) = M, & \gamma_{31}(pc,0) = M, \\
\gamma_5(pc,0) = M, & \gamma_{14}(pc,0) = M, & \gamma_{23}(pc,0) = M, & \gamma_{32}(pc,0) = M, \\
\gamma_6(pc,0) = M, & \gamma_{15}(pc,0) = M, & \gamma_{24}(pc,0) = M, & \gamma_{33}(pc,0) = M, \\
\gamma_7(pc,0) = M, & \gamma_{16}(pc,0) = M, & \gamma_{25}(pc,0) = M, & \gamma_{34}(pc,0) = M, \\
\gamma_8(pc,0) = M, & \gamma_{17}(pc,0) = M, & \gamma_{26}(pc,0) = M, & \gamma_{35}(pc,0) = M, \\
\gamma_0(1,0) = M, & \gamma_0(1,1) = M, & \gamma_0(1,2) = M, & \gamma_0(1,3) = M, \\
\gamma_1(1,0) = M, & \gamma_1(1,1) = M, & \gamma_1(1,2) = M, & \gamma_1(1,3) = U, \\
\gamma_2(1,0) = M, & \gamma_2(1,1) = U, & \gamma_2(1,2) = U, & \gamma_2(1,3) = U, \\
\gamma_3(1,0) = M, & \gamma_3(1,1) = M, & \gamma_3(1,2) = M, & \gamma_3(1,3) = M, \\
\gamma_4(1,0) = M, & \gamma_4(1,1) = M, & \gamma_4(1,2) = M, & \gamma_4(1,3) = U, \\
\gamma_5(1,0) = M, & \gamma_5(1,1) = M, & \gamma_5(1,2) = M, & \gamma_5(1,3) = U, \\
\gamma_6(1,0) = M, & \gamma_6(1,1) = M, & \gamma_6(1,2) = M, & \gamma_6(1,3) = U, \\
\gamma_7(1,0) = M, & \gamma_7(1,1) = U, & \gamma_7(1,2) = U, & \gamma_7(1,3) = U, \\
\gamma_8(1,0) = M, & \gamma_8(1,1) = U, & \gamma_8(1,2) = U, & \gamma_8(1,3) = U, \\
\gamma_9(1,0) = M, & \gamma_9(1,1) = M, & \gamma_9(10,2) = M, & \gamma_9(1,3) = U, \\
\gamma_{10}(1,0) = M, & \gamma_{10}(1,1) = S, & \gamma_{10}(1,2) = M, & \gamma_{10}(1,3) = U, \\
\gamma_{11}(1,0) = M, & \gamma_{11}(1,1) = M & \gamma_{11}(1,2) = M, & \gamma_{11}(1,3) = U, \\
\gamma_{12}(1,0) = M, & \gamma_{12}(1,1) = S, & \gamma_{12}(1,2) = M, & \gamma_{12}(1,3) = U, \\
\gamma_{13}(1,0) = M, & \gamma_{13}(1,1) = M, & \gamma_{13}(1,2) = M, & \gamma_{13}(1,3) = U,
\end{array}
$$

$$\gamma_{14}(1,0)=M,\quad \gamma_{14}(1,1)=S,\quad \gamma_{14}(1,2)=M,\quad \gamma_{14}(1,3)=U,$$
$$\gamma_{15}(1,0)=M,\quad \gamma_{15}(1,1)=U,\quad \gamma_{15}(1,2)=U,\quad \gamma_{15}(1,3)=U,$$
$$\gamma_{16}(1,0)=M,\quad \gamma_{16}(1,1)=U,\quad \gamma_{16}(1,2)=U,\quad \gamma_{16}(1,3)=U,$$
$$\gamma_{17}(1,0)=M,\quad \gamma_{17}(1,1)=U,\quad \gamma_{17}(1,2)=U,\quad \gamma_{17}(1,3)=U,$$
$$\gamma_{18}(1,0)=M,\quad \gamma_{18}(1,1)=U,\quad \gamma_{18}(1,2)=U,\quad \gamma_{18}(1,3)=U,$$
$$\gamma_{19}(1,0)=M,\quad \gamma_{19}(1,1)=U,\quad \gamma_{19}(1,2)=U,\quad \gamma_{19}(1,3)=U,$$
$$\gamma_{20}(1,0)=M,\quad \gamma_{20}(1,1)=U,\quad \gamma_{20}(1,2)=U,\quad \gamma_{20}(1,3)=U,$$
$$\gamma_{21}(1,0)=M,\quad \gamma_{21}(1,1)=M,\quad \gamma_{21}(1,2)=M,\quad \gamma_{21}(1,3)=U,$$
$$\gamma_{22}(1,0)=M,\quad \gamma_{22}(1,1)=M,\quad \gamma_{22}(1,2)=M,\quad \gamma_{22}(1,3)=U,$$
$$\gamma_{23}(1,0)=M,\quad \gamma_{23}(1,1)=M,\quad \gamma_{23}(1,2)=M,\quad \gamma_{23}(1,3)=U,$$
$$\gamma_{24}(1,0)=M,\quad \gamma_{24}(1,1)=M,\quad \gamma_{24}(1,2)=M,\quad \gamma_{24}(1,3)=U,$$
$$\gamma_{25}(1,0)=M,\quad \gamma_{25}(1,1)=M,\quad \gamma_{25}(1,2)=M,\quad \gamma_{25}(1,3)=U,$$
$$\gamma_{26}(1,0)=M,\quad \gamma_{26}(1,1)=M,\quad \gamma_{26}(1,2)=M,\quad \gamma_{26}(1,3)=U,$$
$$\gamma_{27}(1,0)=M,\quad \gamma_{27}(1,1)=M,\quad \gamma_{27}(1,2)=M,\quad \gamma_{27}(1,3)=U,$$
$$\gamma_{28}(1,0)=M,\quad \gamma_{28}(1,1)=S,\quad \gamma_{28}(1,2)=S,\quad \gamma_{28}(1,3)=U.$$
$$\gamma_{29}(1,0)=M,\quad \gamma_{29}(1,1)=U,\quad \gamma_{29}(1,2)=U,\quad \gamma_{29}(1,3)=U,$$
$$\gamma_{30}(1,0)=M,\quad \gamma_{30}(1,1)=S,\quad \gamma_{30}(1,2)=S,\quad \gamma_{30}(1,3)=U,$$
$$\gamma_{31}(1,0)=M,\quad \gamma_{31}(1,1)=U,\quad \gamma_{31}(1,2)=U,\quad \gamma_{31}(1,3)=U,$$
$$\gamma_{32}(1,0)=M,\quad \gamma_{32}(1,1)=S,\quad \gamma_{32}(1,2)=S,\quad \gamma_{32}(1,3)=U,$$
$$\gamma_{33}(1,0)=M,\quad \gamma_{33}(1,1)=U,\quad \gamma_{33}(1,2)=U,\quad \gamma_{33}(1,3)=U,$$
$$\gamma_{34}(1,0)=M,\quad \gamma_{34}(1,1)=S,\quad \gamma_{34}(1,2)=S,\quad \gamma_{34}(1,3)=U,$$
$$\gamma_{35}(1,0)=M,\quad \gamma_{35}(1,1)=U,\quad \gamma_{35}(1,2)=U,\quad \gamma_{35}(1,3)=U$$

β Equations

$$\beta_0(pc,0)=pc,\quad \beta_9(pc,0)=pc,\quad \beta_{18}(pc,0)=pc,\quad \beta_{27}(pc,0)=pc,$$
$$\beta_1(pc,0)=pc,\quad \beta_{10}(pc,0)=pc,\quad \beta_{19}(pc,0)=pc,\quad \beta_{28}(pc,0)=pc,$$
$$\beta_2(pc,0)=pc,\quad \beta_{11}(pc,0)=pc,\quad \beta_{20}(pc,0)=pc,\quad \beta_{29}(pc,0)=pc,$$
$$\beta_3(pc,0)=pc,\quad \beta_{12}(pc,0)=pc,\quad \beta_{21}(pc,0)=pc,\quad \beta_{30}(pc,0)=pc,$$
$$\beta_4(pc,0)=pc,\quad \beta_{13}(pc,0)=pc,\quad \beta_{22}(pc,0)=pc,\quad \beta_{31}(pc,0)=pc,$$
$$\beta_5(pc,0)=pc,\quad \beta_{14}(pc,0)=pc,\quad \beta_{23}(pc,0)=pc,\quad \beta_{32}(pc,0)=pc,$$
$$\beta_6(pc,0)=pc,\quad \beta_{15}(pc,0)=pc,\quad \beta_{24}(pc,0)=pc,\quad \beta_{33}(pc,0)=pc,$$
$$\beta_7(pc,0)=pc,\quad \beta_{16}(pc,0)=pc,\quad \beta_{25}(pc,0)=pc,\quad \beta_{34}(pc,0)=pc,$$
$$\beta_8(pc,0)=pc,\quad \beta_{17}(pc,0)=pc,\quad \beta_{26}(pc,0)=pc,\quad \beta_{35}(pc,0)=pc,$$
$$\beta_0(1,0)=pc,\quad \beta_0(1,1)=1,\quad \beta_0(1,2)=1,\quad \beta_0(1,3)=1,$$
$$\beta_1(1,0)=pc,\quad \beta_1(1,1)=1,\quad \beta_1(1,2)=1,\quad \beta_1(1,3)=\omega,$$
$$\beta_2(1,0)=pc,\quad \beta_2(1,1)=\omega,\quad \beta_2(1,2)=\omega,\quad \beta_2(1,3)=\omega,$$
$$\beta_3(1,0)=pc,\quad \beta_3(1,1)=1,\quad \beta_3(1,2)=1,\quad \beta_3(1,3)=1,$$
$$\beta_4(1,0)=pc,\quad \beta_4(1,1)=1,\quad \beta_4(1,2)=1,\quad \beta_4(1,3)=\omega,$$
$$\beta_5(1,0)=pc,\quad \beta_5(1,1)=1,\quad \beta_5(1,2)=1,\quad \beta_5(1,3)=\omega,$$
$$\beta_6(1,0)=pc,\quad \beta_6(1,1)=1,\quad \beta_6(1,2)=1,\quad \beta_6(1,3)=\omega,$$
$$\beta_7(1,0)=pc,\quad \beta_7(1,1)=\omega,\quad \beta_7(1,2)=\omega,\quad \beta_7(1,3)=\omega,$$
$$\beta_8(1,0)=pc,\quad \beta_8(1,1)=\omega,\quad \beta_8(1,2)=\omega,\quad \beta_8(1,3)=\omega,$$
$$\beta_9(1,0)=pc,\quad \beta_9(1,1)=1,\quad \beta_9(1,2)=1,\quad \beta_9(1,3)=\omega,$$
$$\beta_{10}(1,0)=pc,\quad \beta_{10}(1,1)=9,\quad \beta_{10}(1,2)=1,\quad \beta_{10}(1,3)=\omega,$$
$$\beta_{11}(1,0)=pc,\quad \beta_{11}(1,1)=1,\quad \beta_{11}(1,2)=1,\quad \beta_{11}(1,3)=\omega,$$
$$\beta_{12}(1,0)=pc,\quad \beta_{12}(1,1)=9,\quad \beta_{12}(1,2)=1,\quad \beta_{12}(1,3)=\omega,$$
$$\beta_{13}(1,0)=pc,\quad \beta_{13}(1,1)=1,\quad \beta_{13}(1,2)=1,\quad \beta_{13}(1,3)=\omega,$$
$$\beta_{14}(1,0)=pc,\quad \beta_{14}(1,1)=9,\quad \beta_{14}(1,2)=1,\quad \beta_{14}(1,3)=\omega,$$
$$\beta_{15}(1,0)=pc,\quad \beta_{15}(1,1)=\omega,\quad \beta_{15}(1,2)=\omega,\quad \beta_{15}(1,3)=\omega,$$
$$\beta_{16}(1,0)=pc,\quad \beta_{16}(1,1)=\omega,\quad \beta_{16}(1,2)=\omega,\quad \beta_{16}(1,3)=\omega,$$
$$\beta_{17}(1,0)=pc,\quad \beta_{17}(1,1)=\omega,\quad \beta_{17}(1,2)=\omega,\quad \beta_{17}(1,3)=\omega,$$
$$\beta_{18}(1,0)=pc,\quad \beta_{18}(1,1)=\omega,\quad \beta_{18}(1,2)=\omega,\quad \beta_{18}(1,3)=\omega,$$

$\beta_{19}(1,0) = pc, \quad \beta_{19}(1,1) = \omega, \quad \beta_{19}(1,2) = \omega, \quad \beta_{19}(1,3) = \omega,$
$\beta_{20}(1,0) = pc, \quad \beta_{20}(1,1) = \omega, \quad \beta_{20}(1,2) = \omega, \quad \beta_{20}(1,3) = \omega,$
$\beta_{21}(1,0) = pc, \quad \beta_{21}(1,1) = 1, \quad \beta_{21}(1,2) = 1, \quad \beta_{21}(1,3) = \omega,$
$\beta_{22}(1,0) = pc, \quad \beta_{22}(1,1) = 1, \quad \beta_{22}(1,2) = 1, \quad \beta_{22}(1,3) = \omega,$
$\beta_{23}(1,0) = pc, \quad \beta_{23}(1,1) = 1, \quad \beta_{23}(1,2) = 1, \quad \beta_{23}(1,3) = \omega,$
$\beta_{24}(1,0) = pc, \quad \beta_{24}(1,1) = 1, \quad \beta_{24}(1,2) = 1, \quad \beta_{24}(1,3) = \omega,$
$\beta_{25}(1,0) = pc, \quad \beta_{25}(1,1) = 1, \quad \beta_{25}(1,2) = 1, \quad \beta_{25}(1,3) = \omega,$
$\beta_{26}(1,0) = pc, \quad \beta_{26}(1,1) = 1, \quad \beta_{26}(1,2) = 1, \quad \beta_{26}(1,3) = \omega,$
$\beta_{27}(1,0) = pc, \quad \beta_{27}(1,1) = 1, \quad \beta_{27}(1,2) = 1, \quad \beta_{27}(1,3) = \omega,$
$\beta_{28}(1,0) = pc, \quad \beta_{28}(1,1) = 1, \quad \beta_{28}(1,2) = 2, \quad \beta_{28}(1,3) = \omega.$
$\beta_{29}(1,0) = pc, \quad \beta_{29}(1,1) = \omega, \quad \beta_{29}(1,2) = \omega, \quad \beta_{29}(1,3) = \omega,$
$\beta_{30}(1,0) = pc, \quad \beta_{30}(1,1) = 3, \quad \beta_{30}(1,2) = 4, \quad \beta_{30}(1,3) = \omega,$
$\beta_{31}(1,0) = pc, \quad \beta_{31}(1,1) = \omega, \quad \beta_{31}(1,2) = \omega, \quad \beta_{31}(1,3) = \omega,$
$\beta_{32}(1,0) = pc, \quad \beta_{32}(1,1) = 5, \quad \beta_{32}(1,2) = 6, \quad \beta_{32}(1,3) = \omega,$
$\beta_{33}(1,0) = pc, \quad \beta_{33}(1,1) = \omega, \quad \beta_{33}(1,2) = \omega, \quad \beta_{33}(1,3) = \omega,$
$\beta_{34}(1,0) = pc, \quad \beta_{34}(1,1) = 7, \quad \beta_{34}(1,2) = 8, \quad \beta_{34}(1,3) = \omega,$
$\beta_{35}(1,0) = pc, \quad \beta_{35}(1,1) = \omega, \quad \beta_{35}(1,2) = \omega, \quad \beta_{35}(1,3) = \omega,$
$\beta_0(pc,1) = pc$

δ Equations

$\delta_{pc,0,0}(t,a,x) = t-1, \quad \delta_{pc,0,9}(t,a,x) = t-1, \quad \delta_{pc,0,18}(t,a,x) = t-1,$
$\delta_{pc,0,27}(t,a,x) = t-1,$
$\delta_{pc,0,1}(t,a,x) = t-1, \quad \delta_{pc,0,10}(t,a,x) = t-1, \quad \delta_{pc,0,19}(t,a,x) = t-1,$
$\delta_{pc,0,28}(t,a,x) = t-1,$
$\delta_{pc,0,2}(t,a,x) = t-1, \quad \delta_{pc,0,11}(t,a,x) = t-1, \quad \delta_{pc,0,20}(t,a,x) = t-1,$
$\delta_{pc,0,29}(t,a,x) = t-1,$
$\delta_{pc,0,3}(t,a,x) = t-1, \quad \delta_{pc,0,12}(t,a,x) = t-1, \quad \delta_{pc,0,21}(t,a,x) = t-1,$
$\delta_{pc,0,30}(t,a,x) = t-1,$
$\delta_{pc,0,4}(t,a,x) = t-1, \quad \delta_{pc,0,13}(t,a,x) = t-1, \quad \delta_{pc,0,22}(t,a,x) = t-1,$
$\delta_{pc,0,31}(t,a,x) = t-1,$
$\delta_{pc,0,5}(t,a,x) = t-1, \quad \delta_{pc,0,14}(t,a,x) = t-1, \quad \delta_{pc,0,23}(t,a,x) = t-1,$
$\delta_{pc,0,32}(t,a,x) = t-1,$
$\delta_{pc,0,6}(t,a,x) = t-1, \quad \delta_{pc,0,15}(t,a,x) = t-1, \quad \delta_{pc,0,24}(t,a,x) = t-1,$
$\delta_{pc,0,33}(t,a,x) = t-1,$
$\delta_{pc,0,7}(t,a,x) = t-1, \quad \delta_{pc,0,16}(t,a,x) = t-1, \quad \delta_{pc,0,25}(t,a,x) = t-1,$
$\delta_{pc,0,34}(t,a,x) = t-1,$
$\delta_{pc,0,8}(t,a,x) = t-1, \quad \delta_{pc,0,17}(t,a,x) = t-1, \quad \delta_{pc,0,26}(t,a,x) = t-1,$
$\delta_{pc,0,35}(t,a,x) = t-1,$
$\delta_{1,0,0}(t,a,x) = t-1, \quad \delta_{1,1,0}(t,a,x) = t-35, \quad \delta_{1,2,0}(t,a,x) = t-34,$
$\delta_{1,3,0}(t,a,x) = t-33,$
$\delta_{1,0,1}(t,a,x) = t-1, \quad \delta_{1,1,1}(t,a,x) = t-33, \quad \delta_{1,2,1}(t,a,x) = t-32,$
$\delta_{1,3,1}(t,a,x) = t-1,$
$\delta_{1,0,2}(t,a,x) = t-1, \quad \delta_{1,1,2}(t,a,x) = t-1, \quad \delta_{1,2,2}(t,a,x) = t-1,$
$\delta_{1,3,2}(t,a,x) = t-1,$
$\delta_{1,0,3}(t,a,x) = t-1, \quad \delta_{1,1,3}(t,a,x) = t-33, \quad \delta_{1,2,3}(t,a,x) = t-32,$
$\delta_{1,3,3}(t,a,x) = t-31,$
$\delta_{1,0,4}(t,a,x) = t-1, \quad \delta_{1,1,4}(t,a,x) = t-31, \quad \delta_{1,2,4}(t,a,x) = t-30,$
$\delta_{1,3,4}(t,a,x) = t-1,$
$\delta_{1,0,5}(t,a,x) = t-1, \quad \delta_{1,1,5}(t,a,x) = t-30, \quad \delta_{1,2,5}(t,a,x) = t-29,$
$\delta_{1,3,5}(t,a,x) = t-1,$
$\delta_{1,0,6}(t,a,x) = t-1, \quad \delta_{1,1,6}(t,a,x) = t-29, \quad \delta_{1,2,6}(t,a,x) = t-28,$
$\delta_{1,3,6}(t,a,x) = t-1,$
$\delta_{1,0,7}(t,a,x) = t-1, \quad \delta_{1,1,7}(t,a,x) = t-1, \quad \delta_{1,2,7}(t,a,x) = t-1,$
$\delta_{1,3,7}(t,a,x) = t-1,$

$\delta_{1,0,8}(t,a,x) = t-1$, $\delta_{1,1,8}(t,a,x) = t-1$, $\delta_{1,2,8}(t,a,x) = t-1$,
$\delta_{1,3,8}(t,a,x) = t-1$,
$\delta_{1,0,9}(t,a,x) = t-1$, $\delta_{1,1,9}(t,a,x) = t-24$, $\delta_{1,2,9}(t,a,x) = t-30$,
$\delta_{1,3,9}(t,a,x) = t-1$,
$\delta_{1,0,10}(t,a,x) = t-1$, $\delta_{1,1,10}(t,a,x) = t-1$, $\delta_{1,2,10}(t,a,x) = t-30$,
$\delta_{1,3,10}(t,a,x) = t-1$,
$\delta_{1,0,11}(t,a,x) = t-1$, $\delta_{1,1,11}(t,a,x) = t-26$, $\delta_{1,2,11}(t,a,x) = t-30$,
$\delta_{1,3,11}(t,a,x) = t-1$,
$\delta_{1,0,12}(t,a,x) = t-1$, $\delta_{1,1,12}(t,a,x) = t-1$, $\delta_{1,2,12}(t,a,x) = t-30$,
$\delta_{1,3,12}(t,a,x) = t-1$,
$\delta_{1,0,13}(t,a,x) = t-1$, $\delta_{1,1,13}(t,a,x) = t-28$, $\delta_{1,2,13}(t,a,x) = t-30$,
$\delta_{1,3,13}(t,a,x) = t-1$,
$\delta_{1,0,14}(t,a,x) = t-1$, $\delta_{1,1,14}(t,a,x) = t-1$, $\delta_{1,2,14}(t,a,x) = t-30$,
$\delta_{1,3,14}(t,a,x) = t-1$,
$\delta_{1,0,15}(t,a,x) = t-1$, $\delta_{1,1,15}(t,a,x) = t-1$, $\delta_{1,2,15}(t,a,x) = t-1$,
$\delta_{1,3,15}(t,a,x) = t-1$,
$\delta_{1,0,16}(t,a,x) = t-1$, $\delta_{1,1,16}(t,a,x) = t-1$, $\delta_{1,2,16}(t,a,x) = t-1$,
$\delta_{1,3,16}(t,a,x) = t-1$,
$\delta_{1,0,17}(t,a,x) = t-1$, $\delta_{1,1,17}(t,a,x) = t-1$, $\delta_{1,2,17}(t,a,x) = t-1$,
$\delta_{1,3,17}(t,a,x) = t-1$,
$\delta_{1,0,18}(t,a,x) = t-1$, $\delta_{1,1,18}(t,a,x) = t-1$, $\delta_{1,2,18}(t,a,x) = t-1$,
$\delta_{1,3,18}(t,a,x) = t-1$,
$\delta_{1,0,19}(t,a,x) = t-1$, $\delta_{1,1,19}(t,a,x) = t-1$, $\delta_{1,2,19}(t,a,x) = t-1$,
$\delta_{1,3,19}(t,a,x) = t-1$,
$\delta_{1,0,20}(t,a,x) = t-1$, $\delta_{1,1,20}(t,a,x) = t-1$, $\delta_{1,2,20}(t,a,x) = t-1$,
$\delta_{1,3,20}(t,a,x) = t-1$,
$\delta_{1,0,21}(t,a,x) = t-1$, $\delta_{1,1,21}(t,a,x) = t-35$, $\delta_{1,2,21}(t,a,x) = t-34$,
$\delta_{1,3,21}(t,a,x) = t-1$,
$\delta_{1,0,22}(t,a,x) = t-1$, $\delta_{1,1,22}(t,a,x) = t-34$, $\delta_{1,2,22}(t,a,x) = t-33$,
$\delta_{1,3,22}(t,a,x) = t-1$,
$\delta_{1,0,23}(t,a,x) = t-1$, $\delta_{1,1,23}(t,a,x) = t-33$, $\delta_{1,2,23}(t,a,x) = t-32$,
$\delta_{1,3,23}(t,a,x) = t-1$,
$\delta_{1,0,24}(t,a,x) = t-1$, $\delta_{1,1,24}(t,a,x) = t-32$, $\delta_{1,2,24}(t,a,x) = t-31$,
$\delta_{1,3,24}(t,a,x) = t-1$,
$\delta_{1,0,25}(t,a,x) = t-1$, $\delta_{1,1,25}(t,a,x) = t-31$, $\delta_{1,2,25}(t,a,x) = t-30$,
$\delta_{1,3,25}(t,a,x) = t-1$,
$\delta_{1,0,26}(t,a,x) = t-1$, $\delta_{1,1,26}(t,a,x) = t-30$, $\delta_{1,2,26}(t,a,x) = t-29$,
$\delta_{1,3,26}(t,a,x) = t-1$,
$\delta_{1,0,27}(t,a,x) = t-1$, $\delta_{1,1,27}(t,a,x) = t-29$, $\delta_{1,2,27}(t,a,x) = t-28$,
$\delta_{1,3,27}(t,a,x) = t-1$,
$\delta_{1,0,28}(t,a,x) = t-1$, $\delta_{1,1,28}(t,a,x) = t-1$, $\delta_{1,2,28}(t,a,x) = t-1$,
$\delta_{1,3,28}(t,a,x) = t-1$,
$\delta_{1,0,29}(t,a,x) = t-1$, $\delta_{1,1,29}(t,a,x) = t-1$, $\delta_{1,2,29}(t,a,x) = t-1$,
$\delta_{1,3,29}(t,a,x) = t-1$,
$\delta_{1,0,30}(t,a,x) = t-1$, $\delta_{1,1,30}(t,a,x) = t-1$, $\delta_{1,2,30}(t,a,x) = t-1$,
$\delta_{1,3,30}(t,a,x) = t-1$,
$\delta_{1,0,31}(t,a,x) = t-1$, $\delta_{1,1,31}(t,a,x) = t-1$, $\delta_{1,2,31}(t,a,x) = t-1$,
$\delta_{1,3,31}(t,a,x) = t-1$,
$\delta_{1,0,32}(t,a,x) = t-1$, $\delta_{1,1,32}(t,a,x) = t-1$, $\delta_{1,2,32}(t,a,x) = t-1$,
$\delta_{1,3,32}(t,a,x) = t-1$,

$\delta_{1,0,33}(t,a,x) = t-1$, $\delta_{1,1,33}(t,a,x) = t-1$, $\delta_{1,2,33}(t,a,x) = t-1$,
$\delta_{1,3,33}(t,a,x) = t-1$,
$\delta_{1,0,34}(t,a,x) = t-1$, $\delta_{1,1,34}(t,a,x) = t-1$, $\delta_{1,2,34}(t,a,x) = t-1$,
$\delta_{1,3,34}(t,a,x) = t-1$,
$\delta_{1,0,35}(t,a,x) = t-1$, $\delta_{1,1,35}(t,a,x) = t-1$, $\delta_{1,2,35}(t,a,x) = t-1$,
$\delta_{1,3,35}(t,a,x) = t-1$,

IV Equations

$V_1(0,a,x) = stay$, $V_1(1,a,x) = true$, $V_1(2,a,x) = stay$,
$V_1(3,a,x) = up$, $V_1(4,a,x) = true$, $V_1(5,a,x) = false$,
$V_1(6,a,x) = false$, $V_1(7,a,x) = down$, $V_1(8,a,x) = up$,
$V_1(9,a,x) = true$, $V_1(10,a,x) = true$, $V_1(11,a,x) = false$,
$V_1(12,a,x) = false$, $V_1(13,a,x) = false$, $V_1(14,a,x) = true$,
$V_1(15,a,x) = false$, $V_1(16,a,x) = 90$, $V_1(17,a,x) = true$,
$V_1(18,a,x) = 0$, $V_1(19,a,x) = true$, $V_1(20,a,x) = 0$,
$V_1(21,a,x) = false$, $V_1(22,a,x) = false$, $V_1(23,a,x) = false$,
$V_1(24,a,x) = false$, $V_1(25,a,x) = false$, $V_1(26,a,x) = false$,
$V_1(27,a,x) = false$, $V_1(28,a,x) = 0$, $V_1(29,a,x) = 0$,
$V_1(30,a,x) = 0$, $V_1(31,a,x) = 0$, $V_1(32,a,x) = 0$,
$V_1(33,a,x) = 0$, $V_1(34,a,x) = 0$, $V_1(35,a,x) = 0$,
$V_{pc}(0,a,x) = 1$, $V_{pc}(1,a,x) = 2$, $V_{pc}(2,a,x) = 3$,
$V_{pc}(3,a,x) = 4$, $V_{pc}(4,a,x) = 5$, $V_{pc}(5,a,x) = 6$,
$V_{pc}(6,a,x) = 7$, $V_{pc}(7,a,x) = 8$, $V_{pc}(8,a,x) = 9$,
$V_{pc}(9,a,x) = 10$, $V_{pc}(10,a,x) = 11$, $V_{pc}(11,a,x) = 12$,
$V_{pc}(12,a,x) = 13$, $V_{pc}(13,a,x) = 14$, $V_{pc}(14,a,x) = 15$,
$V_{pc}(15,a,x) = 16$, $V_{pc}(16,a,x) = 17$, $V_{pc}(17,a,x) = 18$,
$V_{pc}(18,a,x) = 19$, $V_{pc}(19,a,x) = 20$, $V_{pc}(20,a,x) = 21$,
$V_{pc}(21,a,x) = 22$, $V_{pc}(22,a,x) = 23$, $V_{pc}(23,a,x) = 24$,
$V_{pc}(24,a,x) = 25$, $V_{pc}(25,a,x) = 26$, $V_{pc}(26,a,x) = 27$,
$V_{pc}(27,a,x) = 28$, $V_{pc}(28,a,x) = 29$, $V_{pc}(29,a,x) = 30$,
$V_{pc}(30,a,x) = 31$, $V_{pc}(31,a,x) = 32$, $V_{pc}(32,a,x) = 33$,
$V_{pc}(33,a,x) = 34$, $V_{pc}(34,a,x) = 35$, $V_{pc}(35,a,x) = 0$

ST Equations

$$V_{pc}(t+1,a,x) = \begin{cases} mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 0 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 1 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 2 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 3 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 4 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 5 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 6 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 7 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 8 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 9 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 10 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 11 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 12 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 13 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 14 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 15 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 16 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 17 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 18 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 19 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 20 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 21 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 22 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 23 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 24 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 25 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 26 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 27 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 28 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 29 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 30 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 31 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 32 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 33 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 34 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 35 \end{cases}$$

$$V_1(t,a,x) = \begin{cases} cond\begin{pmatrix} V_1(t-35,a,x), \\ V_1(t-34,a,x), \\ V_1(t-33,a,x) \end{pmatrix} & \text{if } V_{pc}(t-1,a,x) = 0 \\ or(V_1(t-33,a,x), V_6(t-32,a,x)) & \text{if } V_{pc}(t-1,a,x) = 1 \\ start & \text{if } V_{pc}(t-1,a,x) = 2 \\ cond\begin{pmatrix} V_1(t-33,a,x), \\ V_1(t-32,a,x), \\ V_1(t-31,a,x) \end{pmatrix} & \text{if } V_{pc}(t-1,a,x) = 3 \\ and(V_1(t-31,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 4 \\ and(V_1(t-30,a,x), V_1(t-29,a,x)) & \text{if } V_{pc}(t-1,a,x) = 5 \\ and(V_1(t-29,a,x), V_1(t-28,a,x)) & \text{if } V_{pc}(t-1,a,x) = 6 \\ down & \text{if } V_{pc}(t-1,a,x) = 7 \\ up & \text{if } V_{pc}(t-1,a,x) = 8 \\ eq(V_1(t-24,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 9 \\ eq(a_9(t), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 10 \\ eq(V_1(t-26,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 11 \\ eq(a_9(t), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 12 \\ eq(V_1(t-28,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 13 \\ gt(a_9(t), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 14 \\ false & \text{if } V_{pc}(t-1,a,x) = 15 \\ 90 & \text{if } V_{pc}(t-1,a,x) = 16 \\ true & \text{if } V_{pc}(t-1,a,x) = 17 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 18 \\ true & \text{if } V_{pc}(t-1,a,x) = 19 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 20 \\ or(V_1(t-35,a,x), V_1(t-34,a,x)) & \text{if } V_{pc}(t-1,a,x) = 21 \\ or(V_1(t-34,a,x), V_1(t-33,a,x)) & \text{if } V_{pc}(t-1,a,x) = 22 \\ or(V_1(t-33,a,x), V_1(t-32,a,x)) & \text{if } V_{pc}(t-1,a,x) = 23 \\ gt(V_1(t-33,a,x), V_1(t-31,a,x)) & \text{if } V_{pc}(t-1,a,x) = 24 \\ gt(V_1(t-31,a,x), V_1(t-30,a,x)) & \text{if } V_{pc}(t-1,a,x) = 25 \\ gt(V_1(t-30,a,x), V_1(t-29,a,x)) & \text{if } V_{pc}(t-1,a,x) = 26 \\ gt(V_1(t-29,a,x), V_1(t-28,a,x)) & \text{if } V_{pc}(t-1,a,x) = 27 \\ sub(a_1(t), a_2(t)) & \text{if } V_{pc}(t-1,a,x) = 28 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 29 \\ sub(a_3(t), a_4(t)) & \text{if } V_{pc}(t-1,a,x) = 30 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 31 \\ sub(a_5(t), a_6(t)) & \text{if } V_{pc}(t-1,a,x) = 32 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 33 \\ sub(a_7(t), a_8(t)) & \text{if } V_{pc}(t-1,a,x) = 34 \\ 0 & \text{if } V_{pc}(t-1,a,x) = 35 \end{cases}$$

End

Appendix E

Concrete dSCA definition of GRCP

Begin

Specification	cdSCA
Import	M_A, T
Sorts	SCA_Algebra
Constant Symbols	
VF Function Names	$V_i : T \times A^n \times M^k_{Atup} \rightarrow M_{Atup}$
β Function Names	$\beta_{pc} : N \times N \rightarrow N$
γ Function Names	$\gamma_{pc} : N \times N \rightarrow \{S, M\}$
δ Function Names	$\delta_{i,j,pc} : T \times A^n \times M^k_{Atup} \rightarrow T$
γ Equations	

$$
\begin{array}{llll}
\gamma_0(1,0) = M, & \gamma_0(1,2) = M, & \gamma_0(1,3) = M, & \gamma_0(1,4) = M, \\
\gamma_0(1,1) = M, & & & \\
\gamma_1(1,0) = M, & \gamma_1(1,2) = M, & \gamma_1(1,3) = M, & \gamma_1(1,4) = U, \\
\gamma_1(1,1) = M, & & & \\
\gamma_2(1,0) = M, & \gamma_2(1,2) = U, & \gamma_2(1,3) = U, & \gamma_2(1,4) = U, \\
\gamma_2(1,1) = M, & & & \\
\gamma_3(1,0) = M, & \gamma_3(1,2) = M, & \gamma_3(1,3) = M, & \gamma_3(1,4) = M, \\
\gamma_3(1,1) = M, & & & \\
\gamma_4(1,0) = M, & \gamma_4(1,2) = M, & \gamma_4(1,3) = M, & \gamma_4(1,4) = U, \\
\gamma_4(1,1) = M, & & & \\
\gamma_5(1,0) = M, & \gamma_5(1,2) = M, & \gamma_5(1,3) = M, & \gamma_5(1,4) = U, \\
\gamma_5(1,1) = M, & & & \\
\gamma_6(1,0) = M, & \gamma_6(1,2) = M, & \gamma_6(1,3) = M, & \gamma_6(1,4) = U, \\
\gamma_6(1,1) = M, & & & \\
\gamma_7(1,0) = M, & \gamma_7(1,2) = U, & \gamma_7(1,3) = U, & \gamma_7(1,4) = U, \\
\gamma_7(1,1) = M, & & & \\
\gamma_8(1,0) = M, & \gamma_8(1,2) = U, & \gamma_8(1,3) = U, & \gamma_8(1,4) = U, \\
\gamma_8(1,1) = M, & & & \\
\gamma_9(1,0) = M, & \gamma_9(1,2) = M, & \gamma_9(10,3) = M, & \gamma_9(1,4) = U, \\
\gamma_9(1,1) = M, & & & \\
\gamma_{10}(1,0) = M, & \gamma_{10}(1,2) = S, & \gamma_{10}(1,3) = M, & \gamma_{10}(1,4) = U, \\
\gamma_{10}(1,1) = M, & & & \\
\gamma_{11}(1,0) = M, & \gamma_{11}(1,2) = M & \gamma_{11}(1,3) = M, & \gamma_{11}(1,4) = U, \\
\gamma_{11}(1,1) = M, & & &
\end{array}
$$

$\gamma_{12}(1,0) = M, \quad \gamma_{12}(1,2) = S, \quad \gamma_{12}(1,3) = M, \quad \gamma_{12}(1,4) = U,$
$\gamma_{12}(1,1) = M,$
$\gamma_{13}(1,0) = M, \quad \gamma_{13}(1,2) = M, \quad \gamma_{13}(1,3) = M, \quad \gamma_{13}(1,4) = U,$
$\gamma_{13}(1,1) = M,$
$\gamma_{14}(1,0) = M, \quad \gamma_{14}(1,2) = S, \quad \gamma_{14}(1,3) = M, \quad \gamma_{14}(1,4) = U,$
$\gamma_{14}(1,1) = M,$
$\gamma_{15}(1,0) = M, \quad \gamma_{15}(1,2) = U, \quad \gamma_{15}(1,3) = U, \quad \gamma_{15}(1,4) = U,$
$\gamma_{15}(1,1) = M,$
$\gamma_{16}(1,0) = M, \quad \gamma_{16}(1,2) = U, \quad \gamma_{16}(1,3) = U, \quad \gamma_{16}(1,4) = U,$
$\gamma_{16}(1,1) = M,$
$\gamma_{17}(1,0) = M, \quad \gamma_{17}(1,2) = U, \quad \gamma_{17}(1,3) = U, \quad \gamma_{17}(1,4) = U,$
$\gamma_{17}(1,1) = M,$
$\gamma_{18}(1,0) = M, \quad \gamma_{18}(1,2) = U, \quad \gamma_{18}(1,3) = U, \quad \gamma_{18}(1,4) = U,$
$\gamma_{18}(1,1) = M,$
$\gamma_{19}(1,0) = M, \quad \gamma_{19}(1,2) = U, \quad \gamma_{19}(1,3) = U, \quad \gamma_{19}(1,4) = U,$
$\gamma_{19}(1,1) = M,$
$\gamma_{20}(1,0) = M, \quad \gamma_{20}(1,2) = U, \quad \gamma_{20}(1,3) = U, \quad \gamma_{20}(1,4) = U,$
$\gamma_{20}(1,1) = M,$
$\gamma_{21}(1,0) = M, \quad \gamma_{21}(1,2) = M, \quad \gamma_{21}(1,3) = M, \quad \gamma_{21}(1,4) = U,$
$\gamma_{21}(1,1) = M,$
$\gamma_{22}(1,0) = M, \quad \gamma_{22}(1,2) = M, \quad \gamma_{22}(1,3) = M, \quad \gamma_{22}(1,4) = U,$
$\gamma_{22}(1,1) = M,$
$\gamma_{23}(1,0) = M, \quad \gamma_{23}(1,2) = M, \quad \gamma_{23}(1,3) = M, \quad \gamma_{23}(1,4) = U,$
$\gamma_{23}(1,1) = M,$
$\gamma_{24}(1,0) = M, \quad \gamma_{24}(1,2) = M, \quad \gamma_{24}(1,3) = M, \quad \gamma_{24}(1,4) = U,$
$\gamma_{24}(1,1) = M,$
$\gamma_{25}(1,0) = M, \quad \gamma_{25}(1,2) = M, \quad \gamma_{25}(1,3) = M, \quad \gamma_{25}(1,4) = U,$
$\gamma_{25}(1,1) = M,$
$\gamma_{26}(1,0) = M, \quad \gamma_{26}(1,2) = M, \quad \gamma_{26}(1,3) = M, \quad \gamma_{26}(1,4) = U,$
$\gamma_{26}(1,1) = M,$
$\gamma_{27}(1,0) = M, \quad \gamma_{27}(1,2) = M, \quad \gamma_{27}(1,3) = M, \quad \gamma_{27}(1,4) = U,$
$\gamma_{27}(1,1) = M,$
$\gamma_{28}(1,0) = M, \quad \gamma_{28}(1,2) = S, \quad \gamma_{28}(1,3) = S, \quad \gamma_{28}(1,4) = U.$
$\gamma_{28}(1,1) = M,$
$\gamma_{29}(1,0) = M, \quad \gamma_{29}(1,2) = U, \quad \gamma_{29}(1,3) = U, \quad \gamma_{29}(1,4) = U,$
$\gamma_{29}(1,1) = M,$
$\gamma_{30}(1,0) = M, \quad \gamma_{30}(1,2) = S, \quad \gamma_{30}(1,3) = S, \quad \gamma_{30}(1,4) = U,$
$\gamma_{30}(1,1) = M,$
$\gamma_{31}(1,0) = M, \quad \gamma_{31}(1,2) = U, \quad \gamma_{31}(1,3) = U, \quad \gamma_{31}(1,4) = U,$
$\gamma_{31}(1,1) = M,$
$\gamma_{32}(1,0) = M, \quad \gamma_{32}(1,2) = S, \quad \gamma_{32}(1,3) = S, \quad \gamma_{32}(1,4) = U,$
$\gamma_{32}(1,1) = M,$
$\gamma_{33}(1,0) = M, \quad \gamma_{33}(1,2) = U, \quad \gamma_{33}(1,3) = U, \quad \gamma_{33}(1,4) = U,$
$\gamma_{33}(1,1) = M,$
$\gamma_{34}(1,0) = M, \quad \gamma_{34}(1,2) = S, \quad \gamma_{34}(1,3) = S, \quad \gamma_{34}(1,4) = U,$
$\gamma_{34}(1,1) = M,$
$\gamma_{35}(1,0) = M, \quad \gamma_{35}(1,2) = U, \quad \gamma_{35}(1,3) = U, \quad \gamma_{35}(1,4) = U,$
$\gamma_{35}(1,1) = M,$
$\gamma_{0}(pc,0) = M, \quad \gamma_{9}(pc,0) = M, \quad \gamma_{18}(pc,0) = M, \quad \gamma_{27}(pc,0) = M,$
$\gamma_{1}(pc,0) = M, \quad \gamma_{10}(pc,0) = M, \quad \gamma_{19}(pc,0) = M, \quad \gamma_{28}(pc,0) = M,$
$\gamma_{2}(pc,0) = M, \quad \gamma_{11}(pc,0) = M, \quad \gamma_{20}(pc,0) = M, \quad \gamma_{29}(pc,0) = M,$
$\gamma_{3}(pc,0) = M, \quad \gamma_{12}(pc,0) = M, \quad \gamma_{21}(pc,0) = M, \quad \gamma_{30}(pc,0) = M,$
$\gamma_{4}(pc,0) = M, \quad \gamma_{13}(pc,0) = M, \quad \gamma_{22}(pc,0) = M, \quad \gamma_{31}(pc,0) = M,$

$$\gamma_5(pc,0)=M,\quad \gamma_{14}(pc,0)=M,\quad \gamma_{23}(pc,0)=M,\quad \gamma_{32}(pc,0)=M,$$
$$\gamma_6(pc,0)=M,\quad \gamma_{15}(pc,0)=M,\quad \gamma_{24}(pc,0)=M,\quad \gamma_{33}(pc,0)=M,$$
$$\gamma_7(pc,0)=M,\quad \gamma_{16}(pc,0)=M,\quad \gamma_{25}(pc,0)=M,\quad \gamma_{34}(pc,0)=M,$$
$$\gamma_8(pc,0)=M,\quad \gamma_{17}(pc,0)=M,\quad \gamma_{26}(pc,0)=M,\quad \gamma_{35}(pc,0)=M$$

β Equations

$$\beta_0(1,0)=pc,\quad \beta_0(1,2)=1,\quad \beta_0(1,3)=1,\quad \beta_0(1,4)=1,$$
$$\beta_0(1,1)=1,$$
$$\beta_1(1,0)=pc,\quad \beta_1(1,2)=1,\quad \beta_1(1,3)=1,\quad \beta_1(1,4)=\omega,$$
$$\beta_1(1,1)=1,$$
$$\beta_2(1,0)=pc,\quad \beta_2(1,2)=\omega,\quad \beta_2(1,3)=\omega,\quad \beta_2(1,4)=\omega,$$
$$\beta_2(1,1)=1,$$
$$\beta_3(1,0)=pc,\quad \beta_3(1,2)=1,\quad \beta_3(1,3)=1,\quad \beta_3(1,4)=1,$$
$$\beta_3(1,1)=1,$$
$$\beta_4(1,0)=pc,\quad \beta_4(1,2)=1,\quad \beta_4(1,3)=1,\quad \beta_4(1,4)=\omega,$$
$$\beta_4(1,1)=1,$$
$$\beta_5(1,0)=pc,\quad \beta_5(1,2)=1,\quad \beta_5(1,3)=1,\quad \beta_5(1,4)=\omega,$$
$$\beta_5(1,1)=1,$$
$$\beta_6(1,0)=pc,\quad \beta_6(1,2)=1,\quad \beta_6(1,3)=1,\quad \beta_6(1,4)=\omega,$$
$$\beta_6(1,1)=1,$$
$$\beta_7(1,0)=pc,\quad \beta_7(1,2)=\omega,\quad \beta_7(1,3)=\omega,\quad \beta_7(1,4)=\omega,$$
$$\beta_7(1,1)=1,$$
$$\beta_8(1,0)=pc,\quad \beta_8(1,2)=\omega,\quad \beta_8(1,3)=\omega,\quad \beta_8(1,4)=\omega,$$
$$\beta_8(1,1)=1,$$
$$\beta_9(1,0)=pc,\quad \beta_9(1,2)=1,\quad \beta_9(1,3)=1,\quad \beta_9(1,4)=\omega,$$
$$\beta_9(1,1)=1,$$
$$\beta_{10}(1,0)=pc,\quad \beta_{10}(1,2)=9,\quad \beta_{10}(1,3)=1,\quad \beta_{10}(1,4)=\omega,$$
$$\beta_{10}(1,1)=1,$$
$$\beta_{11}(1,0)=pc,\quad \beta_{11}(1,2)=1,\quad \beta_{11}(1,3)=1,\quad \beta_{11}(1,4)=\omega,$$
$$\beta_{11}(1,1)=1,$$
$$\beta_{12}(1,0)=pc,\quad \beta_{12}(1,2)=9,\quad \beta_{12}(1,3)=1,\quad \beta_{12}(1,4)=\omega,$$
$$\beta_{12}(1,1)=1,$$
$$\beta_{13}(1,0)=pc,\quad \beta_{13}(1,2)=1,\quad \beta_{13}(1,3)=1,\quad \beta_{13}(1,4)=\omega,$$
$$\beta_{13}(1,1)=1,$$
$$\beta_{14}(1,0)=pc,\quad \beta_{14}(1,2)=9,\quad \beta_{14}(1,3)=1,\quad \beta_{14}(1,4)=\omega,$$
$$\beta_{14}(1,1)=1,$$
$$\beta_{15}(1,0)=pc,\quad \beta_{15}(1,2)=\omega,\quad \beta_{15}(1,3)=\omega,\quad \beta_{15}(1,4)=\omega,$$
$$\beta_{15}(1,1)=1,$$
$$\beta_{16}(1,0)=pc,\quad \beta_{16}(1,2)=\omega,\quad \beta_{16}(1,3)=\omega,\quad \beta_{16}(1,4)=\omega,$$
$$\beta_{16}(1,1)=1,$$
$$\beta_{17}(1,0)=pc,\quad \beta_{17}(1,2)=\omega,\quad \beta_{17}(1,3)=\omega,\quad \beta_{17}(1,4)=\omega,$$
$$\beta_{17}(1,1)=1,$$
$$\beta_{18}(1,0)=pc,\quad \beta_{18}(1,2)=\omega,\quad \beta_{18}(1,3)=\omega,\quad \beta_{18}(1,4)=\omega,$$
$$\beta_{18}(1,1)=1,$$
$$\beta_{19}(1,0)=pc,\quad \beta_{19}(1,2)=\omega,\quad \beta_{19}(1,3)=\omega,\quad \beta_{19}(1,4)=\omega,$$
$$\beta_{19}(1,1)=1,$$
$$\beta_{20}(1,0)=pc,\quad \beta_{20}(1,2)=\omega,\quad \beta_{20}(1,3)=\omega,\quad \beta_{20}(1,4)=\omega,$$
$$\beta_{20}(1,1)=1,$$
$$\beta_{21}(1,0)=pc,\quad \beta_{21}(1,2)=1,\quad \beta_{21}(1,3)=1,\quad \beta_{21}(1,4)=\omega,$$
$$\beta_{21}(1,1)=1,$$
$$\beta_{22}(1,0)=pc,\quad \beta_{22}(1,2)=1,\quad \beta_{22}(1,3)=1,\quad \beta_{22}(1,4)=\omega,$$
$$\beta_{22}(1,1)=1,$$
$$\beta_{23}(1,0)=pc,\quad \beta_{23}(1,2)=1,\quad \beta_{23}(1,3)=1,\quad \beta_{23}(1,4)=\omega,$$
$$\beta_{23}(1,1)=1,$$
$$\beta_{24}(1,0)=pc,\quad \beta_{24}(1,2)=1,\quad \beta_{24}(1,3)=1,\quad \beta_{24}(1,4)=\omega,$$
$$\beta_{24}(1,1)=1,$$

$\beta_{25}(1,0) = pc, \quad \beta_{25}(1,2) = 1, \quad \beta_{25}(1,3) = 1, \quad \beta_{25}(1,4) = \omega,$
$\beta_{25}(1,1) = 1,$
$\beta_{26}(1,0) = pc, \quad \beta_{26}(1,2) = 1, \quad \beta_{26}(1,3) = 1, \quad \beta_{26}(1,4) = \omega,$
$\beta_{26}(1,1) = 1,$
$\beta_{27}(1,0) = pc, \quad \beta_{27}(1,2) = 1, \quad \beta_{27}(1,3) = 1, \quad \beta_{27}(1,4) = \omega,$
$\beta_{27}(1,1) = 1,$
$\beta_{28}(1,0) = pc, \quad \beta_{28}(1,2) = 1, \quad \beta_{28}(1,3) = 2, \quad \beta_{28}(1,4) = \omega.$
$\beta_{28}(1,1) = 1,$
$\beta_{29}(1,0) = pc, \quad \beta_{29}(1,2) = \omega, \quad \beta_{29}(1,3) = \omega, \quad \beta_{29}(1,4) = \omega,$
$\beta_{29}(1,1) = 1,$
$\beta_{30}(1,0) = pc, \quad \beta_{30}(1,2) = 3, \quad \beta_{30}(1,3) = 4, \quad \beta_{30}(1,4) = \omega,$
$\beta_{30}(1,1) = 1,$
$\beta_{31}(1,0) = pc, \quad \beta_{31}(1,2) = \omega, \quad \beta_{31}(1,3) = \omega, \quad \beta_{31}(1,4) = \omega,$
$\beta_{31}(1,1) = 1,$
$\beta_{32}(1,0) = pc, \quad \beta_{32}(1,2) = 5, \quad \beta_{32}(1,3) = 6, \quad \beta_{32}(1,4) = \omega,$
$\beta_{32}(1,1) = 1,$
$\beta_{33}(1,0) = pc, \quad \beta_{33}(1,2) = \omega, \quad \beta_{33}(1,3) = \omega, \quad \beta_{33}(1,4) = \omega,$
$\beta_{33}(1,1) = 1,$
$\beta_{34}(1,0) = pc, \quad \beta_{34}(1,2) = 7, \quad \beta_{34}(1,3) = 8, \quad \beta_{34}(1,4) = \omega,$
$\beta_{34}(1,1) = 1,$
$\beta_{35}(1,0) = pc, \quad \beta_{35}(1,2) = \omega, \quad \beta_{35}(1,3) = \omega, \quad \beta_{35}(1,4) = \omega,$
$\beta_{35}(1,1) = 1,$
$\beta_0(pc,0) = pc, \quad \beta_9(pc,0) = pc, \quad \beta_{18}(pc,0) = pc, \quad \beta_{27}(pc,0) = pc,$
$\beta_1(pc,0) = pc, \quad \beta_{10}(pc,0) = pc, \quad \beta_{19}(pc,0) = pc, \quad \beta_{28}(pc,0) = pc,$
$\beta_2(pc,0) = pc, \quad \beta_{11}(pc,0) = pc, \quad \beta_{20}(pc,0) = pc, \quad \beta_{29}(pc,0) = pc,$
$\beta_3(pc,0) = pc, \quad \beta_{12}(pc,0) = pc, \quad \beta_{21}(pc,0) = pc, \quad \beta_{30}(pc,0) = pc,$
$\beta_4(pc,0) = pc, \quad \beta_{13}(pc,0) = pc, \quad \beta_{22}(pc,0) = pc, \quad \beta_{31}(pc,0) = pc,$
$\beta_5(pc,0) = pc, \quad \beta_{14}(pc,0) = pc, \quad \beta_{23}(pc,0) = pc, \quad \beta_{32}(pc,0) = pc,$
$\beta_6(pc,0) = pc, \quad \beta_{15}(pc,0) = pc, \quad \beta_{24}(pc,0) = pc, \quad \beta_{33}(pc,0) = pc,$
$\beta_7(pc,0) = pc, \quad \beta_{16}(pc,0) = pc, \quad \beta_{25}(pc,0) = pc, \quad \beta_{34}(pc,0) = pc,$
$\beta_8(pc,0) = pc, \quad \beta_{17}(pc,0) = pc, \quad \beta_{26}(pc,0) = pc, \quad \beta_{35}(pc,0) = pc$

δ Equations

$\delta_{1,0,0}(t,a,x) = t-1, \quad \delta_{1,1,0}(t,a,x) = t-1, \quad \delta_{1,2,0}(t,a,x) = t-1,$
$\delta_{1,3,0}(t,a,x) = t-1, \quad \delta_{1,4,0}(t,a,x) = t-1,$
$\delta_{1,0,1}(t,a,x) = t-1, \quad \delta_{1,1,1}(t,a,x) = t-1, \quad \delta_{1,2,1}(t,a,x) = t-1,$
$\delta_{1,3,1}(t,a,x) = t-1, \quad \delta_{1,4,1}(t,a,x) = t-1,$
$\delta_{1,0,2}(t,a,x) = t-1, \quad \delta_{1,1,2}(t,a,x) = t-1, \quad \delta_{1,2,2}(t,a,x) = t-1,$
$\delta_{1,3,2}(t,a,x) = t-1, \quad \delta_{1,4,2}(t,a,x) = t-1,$
$\delta_{1,0,3}(t,a,x) = t-1, \quad \delta_{1,1,3}(t,a,x) = t-1, \quad \delta_{1,2,3}(t,a,x) = t-1,$
$\delta_{1,3,3}(t,a,x) = t-1, \quad \delta_{1,4,3}(t,a,x) = t-1,$
$\delta_{1,0,4}(t,a,x) = t-1, \quad \delta_{1,1,4}(t,a,x) = t-1, \quad \delta_{1,2,4}(t,a,x) = t-1,$
$\delta_{1,3,4}(t,a,x) = t-1, \quad \delta_{1,4,4}(t,a,x) = t-1,$
$\delta_{1,0,5}(t,a,x) = t-1, \quad \delta_{1,1,5}(t,a,x) = t-1, \quad \delta_{1,2,5}(t,a,x) = t-1,$
$\delta_{1,3,5}(t,a,x) = t-1, \quad \delta_{1,4,5}(t,a,x) = t-1,$
$\delta_{1,0,6}(t,a,x) = t-1, \quad \delta_{1,1,6}(t,a,x) = t-1, \quad \delta_{1,2,6}(t,a,x) = t-1,$
$\delta_{1,3,6}(t,a,x) = t-1, \quad \delta_{1,4,6}(t,a,x) = t-1,$
$\delta_{1,0,7}(t,a,x) = t-1, \quad \delta_{1,1,7}(t,a,x) = t-1, \quad \delta_{1,2,7}(t,a,x) = t-1,$
$\delta_{1,3,7}(t,a,x) = t-1, \quad \delta_{1,4,7}(t,a,x) = t-1,$
$\delta_{1,0,8}(t,a,x) = t-1, \quad \delta_{1,1,8}(t,a,x) = t-1, \quad \delta_{1,2,8}(t,a,x) = t-1,$
$\delta_{1,3,8}(t,a,x) = t-1, \quad \delta_{1,4,8}(t,a,x) = t-1,$
$\delta_{1,0,9}(t,a,x) = t-1, \quad \delta_{1,1,9}(t,a,x) = t-1, \quad \delta_{1,2,9}(t,a,x) = t-1,$
$\delta_{1,3,9}(t,a,x) = t-1, \quad \delta_{1,4,9}(t,a,x) = t-1,$

$$\delta_{1,0,10}(t,a,x) = t-1, \quad \delta_{1,1,10}(t,a,x) = t-1, \quad \delta_{1,2,10}(t,a,x) = t-1,$$
$$\delta_{1,3,10}(t,a,x) = t-1, \quad \delta_{1,4,10}(t,a,x) = t-1,$$
$$\delta_{1,0,11}(t,a,x) = t-1, \quad \delta_{1,1,11}(t,a,x) = t-1, \quad \delta_{1,2,11}(t,a,x) = t-1,$$
$$\delta_{1,3,11}(t,a,x) = t-1, \quad \delta_{1,4,11}(t,a,x) = t-1,$$
$$\delta_{1,0,12}(t,a,x) = t-1, \quad \delta_{1,1,12}(t,a,x) = t-1, \quad \delta_{1,2,12}(t,a,x) = t-1,$$
$$\delta_{1,3,12}(t,a,x) = t-1, \quad \delta_{1,4,12}(t,a,x) = t-1,$$
$$\delta_{1,0,13}(t,a,x) = t-1, \quad \delta_{1,1,13}(t,a,x) = t-1, \quad \delta_{1,2,13}(t,a,x) = t-1,$$
$$\delta_{1,3,13}(t,a,x) = t-1, \quad \delta_{1,4,13}(t,a,x) = t-1,$$
$$\delta_{1,0,14}(t,a,x) = t-1, \quad \delta_{1,1,14}(t,a,x) = t-1, \quad \delta_{1,2,14}(t,a,x) = t-1,$$
$$\delta_{1,3,14}(t,a,x) = t-1, \quad \delta_{1,4,14}(t,a,x) = t-1,$$
$$\delta_{1,0,15}(t,a,x) = t-1, \quad \delta_{1,1,15}(t,a,x) = t-1, \quad \delta_{1,2,15}(t,a,x) = t-1,$$
$$\delta_{1,3,15}(t,a,x) = t-1, \quad \delta_{1,4,15}(t,a,x) = t-1,$$
$$\delta_{1,0,16}(t,a,x) = t-1, \quad \delta_{1,1,16}(t,a,x) = t-1, \quad \delta_{1,2,16}(t,a,x) = t-1,$$
$$\delta_{1,3,16}(t,a,x) = t-1, \quad \delta_{1,4,16}(t,a,x) = t-1,$$
$$\delta_{1,0,17}(t,a,x) = t-1, \quad \delta_{1,1,17}(t,a,x) = t-1, \quad \delta_{1,2,17}(t,a,x) = t-1,$$
$$\delta_{1,3,17}(t,a,x) = t-1, \quad \delta_{1,4,17}(t,a,x) = t-1,$$
$$\delta_{1,0,18}(t,a,x) = t-1, \quad \delta_{1,1,18}(t,a,x) = t-1, \quad \delta_{1,2,18}(t,a,x) = t-1,$$
$$\delta_{1,3,18}(t,a,x) = t-1, \quad \delta_{1,4,18}(t,a,x) = t-1,$$
$$\delta_{1,0,19}(t,a,x) = t-1, \quad \delta_{1,1,19}(t,a,x) = t-1, \quad \delta_{1,2,19}(t,a,x) = t-1,$$
$$\delta_{1,3,19}(t,a,x) = t-1, \quad \delta_{1,4,19}(t,a,x) = t-1,$$
$$\delta_{1,0,20}(t,a,x) = t-1, \quad \delta_{1,1,20}(t,a,x) = t-1, \quad \delta_{1,2,20}(t,a,x) = t-1,$$
$$\delta_{1,3,20}(t,a,x) = t-1, \quad \delta_{1,4,20}(t,a,x) = t-1,$$
$$\delta_{1,0,21}(t,a,x) = t-1, \quad \delta_{1,1,21}(t,a,x) = t-1, \quad \delta_{1,2,21}(t,a,x) = t-1,$$
$$\delta_{1,3,21}(t,a,x) = t-1, \quad \delta_{1,4,21}(t,a,x) = t-1,$$
$$\delta_{1,0,22}(t,a,x) = t-1, \quad \delta_{1,1,22}(t,a,x) = t-1, \quad \delta_{1,2,22}(t,a,x) = t-1,$$
$$\delta_{1,3,22}(t,a,x) = t-1, \quad \delta_{1,4,22}(t,a,x) = t-1,$$
$$\delta_{1,0,23}(t,a,x) = t-1, \quad \delta_{1,1,23}(t,a,x) = t-1, \quad \delta_{1,2,23}(t,a,x) = t-1,$$
$$\delta_{1,3,23}(t,a,x) = t-1, \quad \delta_{1,4,23}(t,a,x) = t-1,$$
$$\delta_{1,0,24}(t,a,x) = t-1, \quad \delta_{1,1,24}(t,a,x) = t-1, \quad \delta_{1,2,24}(t,a,x) = t-1,$$
$$\delta_{1,3,24}(t,a,x) = t-1, \quad \delta_{1,4,24}(t,a,x) = t-1,$$
$$\delta_{1,0,25}(t,a,x) = t-1, \quad \delta_{1,1,25}(t,a,x) = t-1, \quad \delta_{1,2,25}(t,a,x) = t-1,$$
$$\delta_{1,3,25}(t,a,x) = t-1, \quad \delta_{1,4,25}(t,a,x) = t-1,$$
$$\delta_{1,0,26}(t,a,x) = t-1, \quad \delta_{1,1,26}(t,a,x) = t-1, \quad \delta_{1,2,26}(t,a,x) = t-1,$$
$$\delta_{1,3,26}(t,a,x) = t-1, \quad \delta_{1,4,26}(t,a,x) = t-1,$$
$$\delta_{1,0,27}(t,a,x) = t-1, \quad \delta_{1,1,27}(t,a,x) = t-1, \quad \delta_{1,2,27}(t,a,x) = t-1,$$
$$\delta_{1,3,27}(t,a,x) = t-1, \quad \delta_{1,4,27}(t,a,x) = t-1,$$
$$\delta_{1,0,28}(t,a,x) = t-1, \quad \delta_{1,1,28}(t,a,x) = t-1, \quad \delta_{1,2,28}(t,a,x) = t-1,$$
$$\delta_{1,3,28}(t,a,x) = t-1, \quad \delta_{1,4,28}(t,a,x) = t-1,$$
$$\delta_{1,0,29}(t,a,x) = t-1, \quad \delta_{1,1,29}(t,a,x) = t-1, \quad \delta_{1,2,29}(t,a,x) = t-1,$$
$$\delta_{1,3,29}(t,a,x) = t-1, \quad \delta_{1,4,29}(t,a,x) = t-1,$$
$$\delta_{1,0,30}(t,a,x) = t-1, \quad \delta_{1,1,30}(t,a,x) = t-1, \quad \delta_{1,2,30}(t,a,x) = t-1,$$
$$\delta_{1,3,30}(t,a,x) = t-1, \quad \delta_{1,4,30}(t,a,x) = t-1,$$
$$\delta_{1,0,31}(t,a,x) = t-1, \quad \delta_{1,1,31}(t,a,x) = t-1, \quad \delta_{1,2,31}(t,a,x) = t-1,$$
$$\delta_{1,3,31}(t,a,x) = t-1, \quad \delta_{1,4,31}(t,a,x) = t-1,$$
$$\delta_{1,0,32}(t,a,x) = t-1, \quad \delta_{1,1,32}(t,a,x) = t-1, \quad \delta_{1,2,32}(t,a,x) = t-1,$$
$$\delta_{1,3,32}(t,a,x) = t-1, \quad \delta_{1,4,32}(t,a,x) = t-1,$$
$$\delta_{1,0,33}(t,a,x) = t-1, \quad \delta_{1,1,33}(t,a,x) = t-1, \quad \delta_{1,2,33}(t,a,x) = t-1,$$
$$\delta_{1,3,33}(t,a,x) = t-1, \quad \delta_{1,4,33}(t,a,x) = t-1,$$
$$\delta_{1,0,34}(t,a,x) = t-1, \quad \delta_{1,1,34}(t,a,x) = t-1, \quad \delta_{1,2,34}(t,a,x) = t-1,$$
$$\delta_{1,3,34}(t,a,x) = t-1, \quad \delta_{1,4,34}(t,a,x) = t-1,$$
$$\delta_{1,0,35}(t,a,x) = t-1, \quad \delta_{1,1,35}(t,a,x) = t-1, \quad \delta_{1,2,35}(t,a,x) = t-1,$$
$$\delta_{1,3,35}(t,a,x) = t-1, \quad \delta_{1,4,35}(t,a,x) = t-1,$$

d Equations

$$
\begin{array}{lll}
d(1,1,0)=1, & d(1,2,0)=2, & d(1,3,0)=3,\\
d(1,1,1)=4, & d(1,2,1)=5, & d(1,3,1)=36,\\
d(1,1,2)=37, & d(1,2,2)=37, & d(1,3,2)=37,\\
d(1,1,3)=6, & d(1,2,3)=7, & d(1,3,3)=8,\\
d(1,1,4)=9, & d(1,2,4)=10, & d(1,3,4)=39,\\
d(1,1,5)=11, & d(1,2,5)=12, & d(1,3,5)=40,\\
d(1,1,6)=13, & d(1,2,6)=14, & d(1,3,6)=41,\\
d(1,1,7)=42, & d(1,2,7)=42, & d(1,3,7)=42,\\
d(1,1,8)=43, & d(1,2,8)=43, & d(1,3,8)=43,\\
d(1,1,9)=21, & d(1,2,9)=15, & d(1,3,9)=44,\\
d(1,1,10)=45, & d(1,2,10)=16, & d(1,3,10)=45,\\
d(1,1,11)=21, & d(1,2,11)=17, & d(1,3,11)=46,\\
d(1,1,12)=47, & d(1,2,12)=18, & d(1,3,12)=47,\\
d(1,1,13)=21, & d(1,2,13)=19, & d(1,3,13)=48,\\
d(1,1,14)=49, & d(1,2,14)=20, & d(1,3,14)=49,\\
d(1,1,15)=50, & d(1,2,15)=50, & d(1,3,15)=50,\\
d(1,1,16)=51, & d(1,2,16)=51, & d(1,3,16)=51,\\
d(1,1,17)=52, & d(1,2,17)=52, & d(1,3,17)=52,\\
d(1,1,18)=53, & d(1,2,18)=53, & d(1,3,18)=53,\\
d(1,1,19)=54, & d(1,2,19)=54, & d(1,3,19)=54,\\
d(1,1,20)=55, & d(1,2,20)=55, & d(1,3,20)=55,\\
d(1,1,21)=22, & d(1,2,21)=23, & d(1,3,21)=56,\\
d(1,1,22)=24, & d(1,2,22)=25, & d(1,3,22)=57,\\
d(1,1,23)=26, & d(1,2,23)=27, & d(1,3,23)=58,\\
d(1,1,24)=28, & d(1,2,24)=29, & d(1,3,24)=59,\\
d(1,1,25)=30, & d(1,2,25)=31, & d(1,3,25)=60,\\
d(1,1,26)=32, & d(1,2,26)=33, & d(1,3,26)=61,\\
d(1,1,27)=34, & d(1,2,27)=35, & d(1,3,27)=62,\\
d(1,1,28)=63, & d(1,2,28)=63, & d(1,3,28)=63,\\
d(1,1,29)=64, & d(1,2,29)=64, & d(1,3,29)=64,\\
d(1,1,30)=65, & d(1,2,30)=65, & d(1,3,30)=65,\\
d(1,1,31)=66, & d(1,2,31)=66, & d(1,3,31)=66,\\
d(1,1,32)=67, & d(1,2,32)=67, & d(1,3,32)=67,\\
d(1,1,33)=68, & d(1,2,33)=68, & d(1,3,33)=68,\\
d(1,1,34)=69, & d(1,2,34)=69, & d(1,3,34)=69,\\
d(1,1,35)=70, & d(1,2,35)=70, & d(1,3,35)=70
\end{array}
$$

IV Equations

$V_{pc}(0,a,x)=1,\quad V_{pc}(1,a,x)=2,\quad V_{pc}(2,a,x)=3,$
$V_{pc}(3,a,x)=4,\quad V_{pc}(4,a,x)=5,\quad V_{pc}(5,a,x)=6,$
$V_{pc}(6,a,x)=7,\quad V_{pc}(7,a,x)=8,\quad V_{pc}(8,a,x)=9,$
$V_{pc}(9,a,x)=10,\quad V_{pc}(10,a,x)=11,\quad V_{pc}(11,a,x)=12,$
$V_{pc}(12,a,x)=13,\quad V_{pc}(13,a,x)=14,\quad V_{pc}(14,a,x)=15,$
$V_{pc}(15,a,x)=16,\quad V_{pc}(16,a,x)=17,\quad V_{pc}(17,a,x)=18,$
$V_{pc}(18,a,x)=19,\quad V_{pc}(19,a,x)=20,\quad V_{pc}(20,a,x)=21,$
$V_{pc}(21,a,x)=22,\quad V_{pc}(22,a,x)=23,\quad V_{pc}(23,a,x)=24,$
$V_{pc}(24,a,x)=25,\quad V_{pc}(25,a,x)=26,\quad V_{pc}(26,a,x)=27,$
$V_{pc}(27,a,x)=28,\quad V_{pc}(28,a,x)=29,\quad V_{pc}(29,a,x)=30,$
$V_{pc}(30,a,x)=31,\quad V_{pc}(31,a,x)=32,\quad V_{pc}(32,a,x)=33,$
$V_{pc}(33,a,x)=34,\quad V_{pc}(34,a,x)=351,\quad V_{pc}(35,a,x)=0,$

$$V_1(0,a,x)=\begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(1,a,x)=\begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(2,a,x)=\begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(3,a,x)=\begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(4,a,x)=\begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(5,a,x) = \left(\begin{array}{cccccccc} \mathit{stay} & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array}\right),$$

$$V_1(6,a,x) = \left(\begin{array}{cccccccc} \mathit{stay} & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array}\right),$$

$$V_7(0,a,x) = \left(\begin{array}{cccccccc} \mathit{stay} & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array}\right),$$

$$V_1(8,a,x) = \left(\begin{array}{cccccccc} \mathit{stay} & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array}\right),$$

$$V_1(9,a,x) = \left(\begin{array}{cccccccc} \mathit{stay} & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array}\right),$$

$$V_1(10,a,x) = \left(\begin{array}{cccccccc} \mathit{stay} & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array}\right),$$

$$V_1(11,a,x) = \left(\begin{array}{cccccccc} \mathit{stay} & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array}\right),$$

$$V_1(12,a,x) = \left(\begin{array}{cccccccc} \mathit{stay} & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array}\right),$$

$$V_1(13,a,x) = \left(\begin{array}{cccccccc} \mathit{stay} & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array}\right),$$

$$V_1(14,a,x) = \left(\begin{array}{cccccccc} \mathit{stay} & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array}\right),$$

$$V_1(15,a,x) = \left\{ \begin{array}{cccccccc} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array} \right\},$$

$$V_1(16,a,x) = \left\{ \begin{array}{cccccccc} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array} \right\},$$

$$V_1(17,a,x) = \left\{ \begin{array}{cccccccc} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array} \right\},$$

$$V_1(18,a,x) = \left\{ \begin{array}{cccccccc} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array} \right\},$$

$$V_1(19,a,x) = \left\{ \begin{array}{cccccccc} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array} \right\},$$

$$V_1(20,a,x) = \left\{ \begin{array}{cccccccc} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array} \right\},$$

$$V_1(21,a,x) = \left\{ \begin{array}{cccccccc} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array} \right\},$$

$$V_1(22,a,x) = \left\{ \begin{array}{cccccccc} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array} \right\},$$

$$V_1(23,a,x) = \left\{ \begin{array}{cccccccc} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array} \right\},$$

$$V_1(24,a,x) = \left\{ \begin{array}{cccccccc} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{array} \right\},$$

$$V_1(25,a,x) = \begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(26,a,x) = \begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(27,a,x) = \begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(28,a,x) = \begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(29,a,x) = \begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(30,a,x) = \begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(31,a,x) = \begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(32,a,x) = \begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(33,a,x) = \begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(34,a,x) = \begin{pmatrix} stay & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & u & u & u & u \\ u & u & u & u & & & & \end{pmatrix},$$

$$V_1(35,a,x) = \begin{pmatrix} stay & true & stay & up & true & false & false & down \\ up & true & true & false & false & false & true & false \\ 90 & true & 0 & true & 0 & false & false & false \\ false & false & false & false & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & & & & \end{pmatrix}$$

ST Equations

$$V_{pc}(t+1,a,x) = \begin{cases} mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 0 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 1 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 2 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 3 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 4 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 5 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 6 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 7 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 8 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 9 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 10 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 11 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 12 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 13 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 14 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 15 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 16 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 17 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 18 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 19 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 20 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 21 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 22 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 23 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 24 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 25 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 26 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 27 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 28 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 29 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 30 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 31 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 32 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 33 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 34 \\ mod\left(add\left(V_{pc}(t,a,x),1\right),36\right) & \text{if } V_{pc}(t-1,a,x) = 35 \end{cases}$$

$$
V_1(t+1,a,x) = \begin{cases}
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), cond\left(\Pi_1^{35}(V_1(t,a,x)), \Pi_2^{35}(V_1(t,a,x))\right), \Pi_3^{35}(V_1(t,a,x))\right)) & \text{if } V_{pc}(t,a,x) = 0 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), or\left(\Pi_4^{35}(V_1(t,a,x)), \Pi_5^{35}(V_1(t,a,x))\right)\right) & \text{if } V_{pc}(t,a,x) = 1 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), start\right) & \text{if } V_{pc}(t,\mathrm{a},\mathrm{x}) = 2 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), cond\left(\Pi_6^{35}(V_1(t,a,x)), \Pi_7^{35}(V_1(t,a,x)), \Pi_8^{35}(V_1(t,a,x))\right)\right) & \text{if } V_{pc}(t,a,x) = 3 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), and\left(\Pi_9^{35}(V_1(t,a,x)), \Pi_{10}^{35}(V_1(t,a,x))\right)\right) & \text{if } V_{pc}(t,a,x) = 4 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), and\left\langle\Pi_{11}^{35}(V_1(t,a,x)), \Pi_{12}^{35}(V_1(t,a,x))\right\rangle\right) & \text{if } V_{pc}(t,a,x) = 5 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), and\left\langle\Pi_{13}^{35}(V_1(t,a,x)), \Pi_{14}^{35}(V_1(t,a,x))\right\rangle\right) & \text{if } V_{pc}(t,a,x) = 6 \\
\Upsilon\left(pc, V_1(t,a,x), down\right) & \text{if } V_{pc}(t,a,x) = 7 \\
\Upsilon\left(pc, V_1(t,a,x), up\right) & \text{if } V_{pc}(t,a,x) = 8 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), eq\left(\Pi_{21}^{35}(V_1(t,a,x)), \Pi_{15}^{35}(V_1(t,a,x))\right)\right) & \text{if } V_{pc}(t,a,x) = 9 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), eq\left(a_9(t), \Pi_{16}^{35}(V_1(t,a,x))\right)\right) & \text{if } V_{pc}(t,a,x) = 10 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), eq\left(\Pi_{21}^{35}(V_1(t,a,x)), \Pi_{17}^{35}(V_1(t,a,x))\right)\right) & \text{if } V_{pc}(t,a,x) = 11 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), eq\left\langle a_9(t), \Pi_{18}^{35}(V_1(t,a,x))\right\rangle\right) & \text{if } V_{pc}(t,a,x) = 12 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), eq\left\langle\Pi_{21}^{35}(V_1(t,a,x)), \Pi_{19}^{35}(V_1(t,a,x))\right\rangle\right) & \text{if } V_{pc}(t,a,x) = 13 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), gt\left(a_9(t), \Pi_{20}^{35}(V_1(t,a,x))\right)\right) & \text{if } V_{pc}(t,a,x) = 14 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), false\right) & \text{if } V_{pc}(t,a,x) = 15 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), 90\right) & \text{if } V_{pc}(t,a,x) = 16 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), true\right) & \text{if } V_{pc}(t,a,x) = 17 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), 0\right) & \text{if } V_{pc}(t,a,x) = 18 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), true\right) & \text{if } V_{pc}(t,a,x) = 19 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), 0\right) & \text{if } V_{pc}(t,a,x) = 20 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), or\left(\Pi_{22}^{35}(V_1(t,a,x)), \Pi_{23}^{35}(V_1(t,a,x))\right)\right) & \text{if } V_{pc}(t,a,x) = 21 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), or\left\langle\Pi_{24}^{35}(V_1(t,a,x)), \Pi_{25}^{35}(V_1(t,a,x))\right\rangle\right) & \text{if } V_{pc}(t,a,x) = 22 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), or\left\langle\Pi_{26}^{35}(V_1(t,a,x)), \Pi_{27}^{35}(V_1(t,a,x))\right\rangle\right) & \text{if } V_{pc}(t,a,x) = 23 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), gt\left\langle\Pi_{28}^{35}(V_1(t,a,x)), \Pi_{29}^{35}(V_1(t,a,x))\right\rangle\right) & \text{if } V_{pc}(t,a,x) = 24 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), gt\left\langle\Pi_{30}^{35}(V_1(t,a,x)), \Pi_{31}^{35}(V_1(t,a,x))\right\rangle\right) & \text{if } V_{pc}(t,a,x) = 25 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), gt\left\langle\Pi_{32}^{35}(V_1(t,a,x)), \Pi_{33}^{35}(V_1(t,a,x))\right\rangle\right) & \text{if } V_{pc}(t,a,x) = 26 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), gt\left\langle\Pi_{34}^{35}(V_1(t,a,x)), \Pi_{35}^{35}(V_1(t,a,x))\right\rangle\right) & \text{if } V_{pc}(t,a,x) = 27 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), sub\left(a_1(t), a_2(t)\right)\right) & \text{if } V_{pc}(t,a,x) = 28 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), 0\right) & \text{if } V_{pc}(t,a,x) = 29 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), sub\left(a_3(t), a_4(t)\right)\right) & \text{if } V_{pc}(t,a,x) = 30 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), 0\right) & \text{if } V_{pc}(t,a,x) = 31 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), sub\left(a_5(t), a_6(t)\right)\right) & \text{if } V_{pc}(t,a,x) = 32 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), 0\right) & \text{if } V_{pc}(t,a,x) = 33 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), sub\left(a_7(t), a_8(t)\right)\right) & \text{if } V_{pc}(t,a,x) = 34 \\
\Upsilon\left(V_{pc}(t,a,x), V_1(t,a,x), 0\right) & \text{if } V_{pc}(t,a,x) = 35
\end{cases}
$$

$$V_{pc}(t+1,a,x) = mod(add(V_{pc}(t,a,x),1),1),null)$$

End

Appendix F

SCA to Abstract dSCA Transformation

Begin

Specification SCA_To_acvSCA

Import $SCAAlgebra, adSCAAlgebra,$
$\gamma EqList, \beta EqList, \delta sEqList, \gamma SCAEqList, \beta SCAEqList, \delta dSCAEqList,$
$\beta OpList, \gamma OpList, \delta OpList$
$dSCAISVEqList, dSCAEqList, STVEqList, dSCASTVEqList, IVEquation$

Sorts

Constant Symbols

Function Names

$Transform : SCAAlgebra \rightarrow adSCAAlgebra$
$Create\gamma s : SCAAlgebra \rightarrow \gamma EqList$
$Create\beta s : SCAAlgebra \rightarrow \beta EqList$
$Create\delta s : SCAAlgebra \rightarrow \delta sEqList$
$CreateISVFs : SCAAlgebra \rightarrow dSCAISVEqList$
$CreateSTVFs : SCAAlgebra \rightarrow dSCASTVEqList$
$B\gamma s : N^2 \times \gamma SCAEqList \times \gamma dSCAEqList \rightarrow \gamma dSCAEqList$
$B\gamma : N^2 \times \gamma SCAEqList \rightarrow \gamma dSCAEqList$
$B\gamma Args : N^2 \times \gamma SCAEqList \times \gamma dSCAEqList \rightarrow \gamma dSCAEqList$
$B\gamma Arg : N^2 \times \gamma SCAEqList\times \rightarrow \gamma dSCAEqList$
$B\beta s : N^2 \times \beta SCAEqList \times \beta dSCAEqList \rightarrow \beta dSCAEqList$
$B\beta : N^2 \times \beta SCAEqList \rightarrow \beta dSCAEqList$
$B\beta Args : N^2 \times \beta SCAEqList \times \beta dSCAEqList \rightarrow \beta dSCAEqList$
$B\beta Arg : N^2 \times \beta SCAEqList\times \rightarrow \beta dSCAEqList$
$B\delta s : N^2\delta dSCAEqList \rightarrow \delta dSCAEqList$
$B\delta Args : N^2 \times \delta dSCAEqList \rightarrow dSCAEqList$
$BSTs : \ STVEqList \times dSCASTVEqList \times \beta dSCAEqList\times$
$\gamma dSCAEqList \times \delta dSCAEqList \rightarrow dSCASTVEqList$
$BST : IVEquation \times \beta OpList \times \gamma OpList \times \delta OpList \rightarrow IVEquation$
$rewire : Term \times N \times \beta OpList \times \gamma OpList \times \delta OpList \rightarrow Term$

$$
\begin{array}{ll}
\textbf{Equations} & Transform(SCA_{src}) = CreateadSCA \left(\begin{array}{l} GetName(SCA_{src}), \\ SCAAlgebra\ , \\ [], \\ [], \\ VFOp, \\ \gamma_0 : N^2 \rightarrow \{M, S, U\}, \\ \beta_0 : N^2 \rightarrow N, \\ \delta Op\ , \\ CreateISVFs(SCA_{src}), \\ CreateSTVFs(SCA_{src}), \\ Create\gamma s(SCA_{src}), \\ Create\beta s(SCA_{src}), \\ Create\delta s(SCA_{src}) \end{array} \right) \\
& Create\gamma s\ (\ source\_SCA\ ) = B\gamma s \left(\begin{array}{l} num\_mod(source\_SCA), \\ GetMaxA(source\_SCA), \\ Get\gamma Eqs(source\_SCA), \\ [] \end{array} \right) \\
& Create\beta s\ (\ source\_SCA\ ) = B\beta s \left(\begin{array}{l} num\_mod(source\_SCA), \\ GetMaxA(source\_SCA), \\ Get\beta Eqs(source\_SCA), \\ [] \end{array} \right) \\
& Create\delta s(source\_SCA) = B\delta s \left(\begin{array}{l} nummod(source\_SCA), \\ GetMaxA(source\_SCA), \\ [], \end{array} \right) \\
& CreateISVFs(Source\_SCA) = (GetEqIV(Source\_SCA), BuildVF(V_{pc}(t, a, x), 0)) \\
& CreateSTVFs(Source\_SCA) = BSTs \left(\begin{array}{l} GetEqIV(Source\_SCA), \\ [], \\ Create\beta s\ (Get\beta Ops(Source\_SCA))\ , \\ Create\gamma s\ (Get\gamma Ops(Source\_SCA))\ , \\ Create\delta s\ (Get\delta Ops(Source\_SCA))\ , \end{array} \right)
\end{array}
$$

$$
\begin{aligned}
B\gamma s\left(\begin{array}{l} num\_mod, \\ Max_A, \\ eqs, \\ neqs, \end{array}\right) &= B\gamma s\left(\begin{array}{l} num\_mod - 1, \\ Max_A, \\ eqs, \\ \left(B\gamma\left(\begin{array}{l} Max_A, \\ num\_mod, \\ eqs, \end{array}\right), neqs\right) \end{array}\right) \\
B\gamma s\left(\begin{array}{l} num\_mod, \\ Max_A, \\ eqs, \\ neqs, \end{array}\right) &= \left(\begin{array}{l} neqs, Build\gamma\left(\begin{array}{l} \gamma_0, \\ pc, \\ 0, \\ M \end{array}\right) \end{array}\right) \\
B\gamma\left(\begin{array}{l} Max_A, \\ num\_mod, \\ eqs, \end{array}\right) &= \left(\begin{array}{l} Build\gamma\left(\begin{array}{l} \gamma_0, \\ num\_mod, \\ 0, \\ M \end{array}\right), B\gamma Args\left(\begin{array}{l} Max_A, \\ num\_mod, \\ eqs, \\ [] \end{array}\right) \end{array}\right) \\
B\gamma Args\left(\begin{array}{l} arg\_val, \\ num\_mod, \\ eqs, \\ [] \end{array}\right) &= \left(\begin{array}{l} B\gamma Args\left(\begin{array}{l} arg\_val - 1, \\ num\_mod, \\ eqs, \\ [] \end{array}\right), B\gamma Arg\left(\begin{array}{l} arg\_val, \\ num\_mod, \\ eqs \end{array}\right) \end{array}\right) \\
B\gamma Args\left(\begin{array}{l} 0, \\ num\_mod, \\ eqs, \\ neqs \end{array}\right) &= neqs \\
B\gamma Arg\left(\begin{array}{l} arg\_val, \\ num\_mod, \\ eqs \end{array}\right) &= \begin{cases} Build\gamma\left(\begin{array}{l} \gamma_0, \\ num\_mod, \\ arg\_val, \\ M \end{array}\right) & if\ GetEl\left(\begin{array}{l} eqs, \\ num\_mod, \\ arg\_val \end{array}\right) \neq [] \\ Build\gamma\left(\begin{array}{l} \gamma_0, \\ num\_mod, \\ arg\_val, \\ U \end{array}\right) & otherwise \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
B\beta s\left(\begin{array}{l} num\_mod, \\ Max_A, \\ eqs, \\ neqs, \end{array}\right) &= B\beta s\left(\begin{array}{l} num\_mod-1, \\ Max_A, \\ eqs, \\ B\beta\left(\begin{array}{l} Max_A, \\ num\_mod, \\ eqs, \end{array}\right), neqs \end{array}\right) \\
B\beta s\left(\begin{array}{l} num\_mod, \\ Max_A, \\ eqs, \\ neqs, \end{array}\right) &= \left(\begin{array}{l} neqs, Build\beta\left(\begin{array}{l} \beta_0, \\ pc, \\ 0, \\ pc \end{array}\right) \end{array}\right) \\
B\beta\left(\begin{array}{l} Max_A, \\ num\_mod, \\ eqs, \end{array}\right) &= \left(\begin{array}{l} Build\beta\left(\begin{array}{l} \beta_0, \\ num\_mod, \\ 0, \\ pc \end{array}\right), B\beta Args\left(\begin{array}{l} Max_A, \\ num\_mod, \\ eqs, \\ [] \end{array}\right) \end{array}\right) \\
B\beta Args\left(\begin{array}{l} arg\_val, \\ num\_mod, \\ eqs, \\ [] \end{array}\right) &= \left(\begin{array}{l} B\beta Args\left(\begin{array}{l} arg\_val-1, \\ num\_mod, \\ eqs, \\ [] \end{array}\right), B\beta Arg\left(\begin{array}{l} arg\_val, \\ num\_mod, \\ eqs \end{array}\right) \end{array}\right) \\
B\beta Args\left(\begin{array}{l} 0, \\ num\_mod, \\ eqs, \\ neqs \end{array}\right) &= neqs \\
B\beta Arg\left(\begin{array}{l} av, \\ nm, \\ eqs \end{array}\right) &= \begin{cases} Build\beta\left(\begin{array}{l} \beta_0, \\ nm, \\ av, \\ RetTerm(GetEl\left(\begin{array}{l} eqs, \\ nm, \\ av \end{array}\right), 2) \end{array}\right) & ifGetEl\left(\begin{array}{l} eqs, \\ nm, \\ av \end{array}\right) \neq [] \\ Build\beta\left(\begin{array}{l} \beta_0, \\ nm, \\ av, \\ \omega \end{array}\right) & otherwise \end{cases}
\end{aligned}
$$

$$
\begin{array}{l}
B\delta s\left(\begin{array}{l} num\_mod, \\ Max_A, \\ neqs \end{array}\right) = B\delta s\left(\begin{array}{l} num\_mod - 1, \\ Max_A, \\ B\delta Args\left(\begin{array}{l} Max_A, \\ num\_mod, \\ [] \end{array}\right) \end{array}\right) \\
B\delta s\left(\begin{array}{l} 0, \\ Max_A, \\ neqs \end{array}\right) = \left(\begin{array}{l} neqs, Build\delta\left(\begin{array}{l} num\_mod, \\ 0, \\ 0, \\ t-1 \end{array}\right) \end{array}\right) \\
B\delta Args\left(\begin{array}{l} arg\_val, \\ num\_mod, \\ neqs \end{array}\right) = \left(\begin{array}{l} B\delta Args\left(\begin{array}{l} arg\_val - 1, \\ num\_mod, \\ neqs \end{array}\right), Build\delta\left(\begin{array}{l} num\_mod, \\ arg_val, \\ 0, \\ t-1 \end{array}\right) \end{array}\right) \\
B\delta Args\left(\begin{array}{l} 0, \\ num\_mod, \\ neqs \end{array}\right) = Build\delta\left(\begin{array}{l} num\_mod, \\ 0, \\ 0, \\ t-1 \end{array}\right) \\
BSTs\left(\begin{array}{l} (e, eqs), \\ neqs, \\ \beta s, \\ \gamma s, \\ \delta s \end{array}\right) = BSTs\left(\begin{array}{l} eqs, \\ \left(\begin{array}{l} BST\left(\begin{array}{l} e, \\ \beta s, \\ \gamma s, \\ \delta s \end{array}\right), neqs \end{array}\right), \\ \beta s, \\ \gamma s, \\ \delta s \end{array}\right) \\
BSTs\left(\begin{array}{l} [], \\ neqs, \\ \beta s, \\ \gamma s, \\ \delta s \end{array}\right) = \left(\begin{array}{l} neqs, \\ CreateVF\left(\begin{array}{l} V_{pc}(t+1, a, x), \\ V_{pc}(t, a, x) = 0, \\ cond\left(\begin{array}{l} mod(add(V_{pc}(t, a, x), 1), 1), \\ null \end{array}\right) \end{array}\right) \end{array}\right)
\end{array}
$$

$$
BST\begin{pmatrix} e, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = CreateVF\begin{pmatrix} RetTerm(e,1), \\ V_{pc}(t,a,x) = 0, \\ \begin{pmatrix} cond & rewire & \begin{pmatrix} RetTerm(e,2), \\ GetIndex(RetTerm(e,1)), \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} \\ & & \\ & & null \end{pmatrix}, \\ \begin{pmatrix} op, \\ i, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = op\begin{pmatrix} t_1, \\ i, \\ 1, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_2 \\ i, \\ 2, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_3 \\ i, \\ 3, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} \end{pmatrix}
$$

$$
rewire\begin{pmatrix} op, \\ i, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = op\begin{pmatrix} t_1, \\ i, \\ 1, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_2 \\ i, \\ 2, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_3 \\ i, \\ 3, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}
$$

$$
rewire\begin{pmatrix} op(t_1), \\ i, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = op\begin{pmatrix} t_1, \\ i, \\ 1, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_2 \\ i, \\ 2, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_3 \\ i, \\ 3, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}
$$

$$
rewire\begin{pmatrix} op(t_1,t_2), \\ i, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = op\begin{pmatrix} t_1, \\ i, \\ 1, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_2 \\ i, \\ 2, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_3 \\ i, \\ 3, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}
$$

$$
rewire\begin{pmatrix} op(t_1,t_2,t_3), \\ i, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix} = op\begin{pmatrix} t_1, \\ i, \\ 1, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_2 \\ i, \\ 2, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}, wire\begin{pmatrix} t_3 \\ i, \\ 3, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}
$$

$$
\begin{pmatrix} t, \\ i, \\ j, \\ \beta s, \\ \gamma s, \\ \delta s \end{pmatrix}_{wire} = \begin{cases} V_{RetTerm}\left(\begin{pmatrix} \beta s, \\ i, \\ j, \\ 0 \end{pmatrix}_{GetEl}, 2\right), & \left(\begin{pmatrix} \delta s, \\ i, \\ j, \\ 0 \end{pmatrix}_{GetEl}, 2\right)_{(RetTerm}, a, x) \quad if\ cond_1 \\ a_{RetTerm}\left(\begin{pmatrix} \beta s, \\ i, \\ j, \\ 0 \end{pmatrix}_{GetEl}, 2\right), & \left(\begin{pmatrix} \delta s, \\ i, \\ j, \\ 0 \end{pmatrix}_{GetEl}, 2\right)_{(RetTerm}) \quad if\ cond_2 \end{cases}
$$

End

where

$$cond_1 = RetTerm(GetEl(\gamma s, i, j, 0), 2) = M$$
$$cond_2 = RetTerm(GetEl(\gamma s, i, j, 0), 2) = S$$

and

$$VFOp = \begin{pmatrix} V_0 : T \times M_A^n \times M_A^{k+1} \to M_A, \\ \vdots, \\ V_{k+1} : T \times M_A^n \times M_A^{k+1} \to M_A \end{pmatrix}$$

$$\delta Op = \begin{pmatrix} \delta_{0,0,0} : T \times M_A^n \times M_A^{k+1} \to T, \\ \vdots, \\ \delta_{i,j,0} : T \times M_A^n \times M_A^{k+1} \to T \end{pmatrix}$$

with

$$k = num\_mod(Src\_SCA)$$
$$j = Get\_MaxA(Src\_SCA)$$
$$n = num\_inp(Src\_SCA)$$

Appendix G

Abstract dSCA to Abstract dSCA Transformation Details

G.1 Process

This appendix describes the process of transforming an abstract dSCA with defining shape $\nabla = (n_1, m_1)$ to an abstract dSCA with an defining shape of $\nabla = (n_2, m_2)$. The transformations required for the following equation lists within a supplied abstract SCA specification are covered:

1. Wiring Functions;
2. Delay Functions;
3. Initial State Equations; and
4. State Transition Equations.

After discussing the necessary transformations they are used to transform the abstract dSCA produced in the last chapter to an abstract dSCA with defining shape of $\nabla = (1, k)$.

G.1.1 Prerequisites

- The source network, N_1 has $k_1 > 1$ modules and $Max_{n_1} > 0$ component specifications in its modules definitions;

- The object network, N_2 has $k_2 > 1$ modules and $Max_{n_2} > 0$ component specifications in its modules definitions;
- The defining size of N_2 must be equal to or greater than the defining size of N_1, i.e. $\triangle(N_2) \geq \triangle(1)$;
- There exists the total mapping, Ξ given as:

$$\Xi : \mathbb{N}_{k_1} \times \mathbb{N}_{pc_1} \rightarrow \mathbb{N}_{k_2} \times \mathbb{N}_{pc_2}$$

  that maps modules and execution orders of N_1 to modules and execution orders of N_2; and
- There exists the inverse mapping Ξ^{-1}, given as:

$$\Xi^{-1} : \mathbb{N}_{k_2} \times \mathbb{N}_{pc_2} \rightsquigarrow \mathbb{N}_{k_1} \times \mathbb{N}_{pc_1}$$

  (Note that this mapping may not be total, since some functional components of N_2 may be the undefined operation used to ensure synchronicity of the network).

G.1.2 Mapping Function

The provision of a mapping function is a fundamental prerequisite before this transformation can occur. Its purpose is to provide a total mapping between when a particular function executed on a particular module in the source network and what module and when it will execute on the target network. It is a simple list of equations containing two pairs:

$$(i_{src}, pc\_val_{src}) = (i_{tgt}, pc\_val_{tgt})$$

and must be defined for all values $i_{src} = 1, \ldots, k$ of the k-module source abstract dSCA and $pc\_val_{src} = 0, \ldots, Max_N - 1$. The mapping is denoted as Ξ, and has the (partial) inverse Ξ^{-1}. There is no need to map the program counter module.

G.1.3 Wiring Functions

Unlike the previous transformation, wiring functions will alter values radically to provide the dynamic retiming and structure necessary to support a re-shaped abstract dSCA.

γ-wiring Operations

Consider the source abstract dSCA γ-wiring function:

$$\gamma_{pc\_val_1}(i_1, j_2) = z_2$$

the corresponding target abstract dSCA γ-wiring function will be:

$$\gamma_{pc\_val_2}(i_2, j_2) = z_2$$

where $j_1 = j_2$, and $\Xi\,(i_1, pc\_val_1) = (i_2, pc\_val_2)$

The *informal* process of generating target abstract dSCA γ-wiring functions is to walk the structure of the target architecture creating wiring functions for all modules at all values of the program counter for the number of inputs to each module.

- For each module m_i where $i \in \mathbb{N}_{k_2}$ and $i > 0$:
  - For each pc_val where $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$:
    - For the o^{th} argument of each module create:

$$\gamma'_{pc\_val}(i, 0) = M$$

    - For each argument where $j \in \{1, \ldots, n_2(i)\}$ create a new γ-wiring function

$$\gamma'_{pv\_val}(i, j) = \begin{cases} Value\ from\ source & \text{if } \Xi^{-1}(i, j) \downarrow \\ U & \text{otherwise} \end{cases}$$

      with the intended meaning that the undefined connection is given if the inverse mapping is not defined, otherwise the appropriate value from the source network is used.

- For module 0 create Max_{N_2} γ-wiring functions wiring m_0 back to itself.

Formally, the $Create\gamma s$ operation:

$$Create\gamma s : dSCAAlgebra \times N^2 \times MapEqList \rightarrow \gamma dSCAEqList$$

is introduced and is defined as:

$$Create\gamma s \begin{pmatrix} Source\_SCA, \\ num\_mod, \\ Max_N, \\ \Xi^{-1} \end{pmatrix} = B\gamma s \begin{pmatrix} nm, \\ Get\gamma Eqs(source\_SCA), \\ [], \\ Max_N, \\ GetMaxA(source\_SCA), \\ \Xi^{-1} \end{pmatrix}$$

It takes as arguments the source abstract SCA, the number of modules in the target abstract SCA and then value of Max_N for that network as well as the inverse mapping function equation list. The maximum number of arguments that all modules take in the source is extracted from the source specification - since this cannot change through transformation.

The $B\gamma s$ operation, given as:

$$B\gamma s : N \times \gamma dSCAEqList^2 \times N \times N \times MapEqList \rightarrow \gamma dSCAEqList$$

is defined recursively over the number of modules in the target SCA in two cases, the first representing the case where the module number is 0, and the second case where it is not. When the module under consideration is the 0^{th} module, $B\gamma s$ is defined as the recursive call to itself:

$$B\gamma s \begin{pmatrix} mod\_val, \\ old\gamma s, \\ new\gamma s, \\ Max_N, \\ Max_A, \\ \Xi^{-1} \end{pmatrix} = B\gamma s \begin{pmatrix} mod\_val - 1, \\ old\gamma s, \\ \begin{pmatrix} B\gamma pc \begin{pmatrix} Max_N - 1, \\ old\gamma s, \\ [], \\ mod\_val, \\ Max_A, \\ \Xi^{-1} \end{pmatrix}, new\gamma s \end{pmatrix}, \\ Max_N, \\ Max_A, \\ \Xi^{-1} \end{pmatrix}$$

The recursive call contains an argument where a list is appended to the newly generated γ-wiring functions for a module. This list is created by calling the $B\gamma pc$ operation:

$$B\gamma pc : N \times \gamma sSCAEqList^2 \times N \times N \times MapEqList \rightarrow \gamma dSCAEqList$$

which is itself defined recursively over the values that the program counter may take in two cases: firstly where the program counter is equal to 0 and secondly where the program counter is greater

than zero. In the second case $B\gamma pc$ is defined as:

$$B\gamma pc\begin{pmatrix} pc\_val, \\ old\gamma s, \\ new\gamma s, \\ mod\_val, \\ Max_A, \\ \Xi^{-1} \end{pmatrix} = B\gamma pc\begin{pmatrix} pc\_val - 1, \\ old\gamma s, \\ \begin{pmatrix} B\gamma arg\begin{pmatrix} Max_A - 1, \\ old\gamma s, \\ [], \\ mod\_val, \\ pc\_val, \\ \Xi^{-1} \end{pmatrix}, new\gamma s \end{pmatrix}, \\ mod\_val, \\ Max_A, \\ \Xi^{-1} \end{pmatrix}$$

The operation recurses on itself building a list of new γ-wiring functions for a module at a particular value of the program counter for all inputs to a module by calling the $B\gamma arg$ operation:

$$B\gamma arg : N \times \gamma dSCAEqList^2 \times N^2 \times MapEqList \rightarrow \gamma dSCAEqList$$

$B\gamma arg$ is itself also defined recursively, this time over the argument number under consideration. It has two cases, the first where the argument index is zero, and the second where it is not. For the second case it is defined as:

$$B\gamma arg\begin{pmatrix} arg\_num, \\ old\gamma s, \\ new\gamma s, \\ mod\_val, \\ pc\_val, \\ \Xi^{-1} \end{pmatrix} = B\gamma arg\begin{pmatrix} arg\_num - 1, \\ old\gamma s, \\ \begin{pmatrix} B\gamma\begin{pmatrix} mod\_val, \\ arg\_num, \\ pc\_val, \\ old\gamma s, \\ \Xi^{-1} \end{pmatrix}, neqs \end{pmatrix}, \\ mod\_val, \\ pc\_val, \\ \Xi^{-1} \end{pmatrix}$$

where the $B\gamma$ operation is used to construct the γ-wiring function for this particular argument index at a particular program counter for a particular module. It is given as:

$$B\gamma : N^3 \times \gamma dSCAEqList \times MapEqList \rightarrow \gamma Equation$$

To construct the γ-wiring for a function it is first identified whether there exists a corresponding element in the source abstract dSCA. This is achieved by examining the inverse mapping function.

If the inverse mapping is undefined for the module and program counter values under consideration then the wiring in both the source and target abstract SCAs are unimportant. In such a case, the output of the $B\gamma$ operation is defined to be the creation of a γ-wiring function to the unconnected value U. Therefore, where $\Xi^{-1}(mod\_val, pc\_val) \uparrow$ we define:

$$B\gamma \begin{pmatrix} mod\_val, \\ arg\_val, \\ pc\_val, \\ old\gamma s, \\ \Xi^{-1} \end{pmatrix} = Build\gamma \begin{pmatrix} \gamma_{pc\_val}, \\ mod\_val, \\ arg\_val, \\ U \end{pmatrix}$$

Similarly, it may be the case that the inverse mapping function is defined, but in the source abstract dSCA there is no γ-wiring function defined for this combination of module number, program counter value and argument number. It can be easily identified what the corresponding wiring function was in the source abstract dSCA, since it will be:

$$\gamma_{snd(\Xi^{-1}(mod\_val, pc\_val))}(fst(\Xi^{-1}(mod\_val, pc\_val)), arg\_val)$$

Where this γ-wiring function does not exist then the result of $B\gamma$ is:

$$B\gamma \begin{pmatrix} mod\_val, \\ arg\_val, \\ pc\_val, \\ old\gamma s, \\ \Xi^{-1} \end{pmatrix} = Build\gamma \begin{pmatrix} \gamma_{pc\_val}, \\ mod\_val, \\ arg\_val, \\ U \end{pmatrix}$$

The final case is where the inverse mapping is defined and a corresponding β-wiring function exists in the source network. For this situation the γ-wiring operation in the target dSCA is constructed as:

$$\gamma_{pc\_val}(mod\_val, arg\_val) = RetTerm \left(GetEl \begin{pmatrix} old\gamma s, \\ fst(\Xi^{-1}(mod\_val, pc\_val)), \\ arg\_val, \\ snd(\Xi^{-1}(mod\_val, pc\_val)) \end{pmatrix}, 2 \right)$$

the $B\gamma$ operation can therefore be defined, when $\Xi^{-1}(mod\_val, pc\_Val) \downarrow$ as:

$$B\gamma \begin{pmatrix} mod\_val, \\ arg\_val, \\ pc\_val, \\ old\gamma s, \\ \Xi^{-1} \end{pmatrix} = Build\gamma \begin{pmatrix} \gamma_{pc\_val}, \\ mod\_val, \\ arg\_val, \\ RetTerm \left(GetEl \begin{pmatrix} old\gamma s, \\ i_{old}, \\ arg\_val, \\ pc_{old} \end{pmatrix}, 2 \right) \end{pmatrix}$$

where:

$$i_{old} = fst(RetTerm(GetEl(\Xi^{-1}, mod\_val, pc\_val)), 2)$$
$$pc_{old} = snd(RetTerm(GetEl(\Xi^{-1}, mod\_val, pc\_val)), 2)$$

The second case of the $B\gamma arg$ operation, where the argument index is zero, simply generates a γ-wiring function for the 0^{th} argument. This wiring, by definition, will be to the program counter module, and is appended to the list of functions generated for that module. The function returns this new list, and is defined as:

$$B\gamma arg \begin{pmatrix} 0, \\ old\gamma s, \\ new\gamma s, \\ mod\_val, \\ pc\_val, \\ \Xi^{-1} \end{pmatrix} = \begin{pmatrix} Build\gamma \begin{pmatrix} \gamma_{pc\_val} \\ mod\_val, \\ 0, \\ M \end{pmatrix}, new\gamma s \end{pmatrix}$$

The second case of the $B\gamma pc$ operation, where the program counter is 0, simply generates the γ-wiring functions for module mn at $pc\_val = 0$, and appends them to the list of already generated γ-wiring functions for module mn at all other values of the program counter. It is defined as:

$$B\gamma pc \begin{pmatrix} 0, \\ old\gamma s, \\ new\gamma s, \\ mod\_val, \\ Max_A, \\ \Xi^{-1} \end{pmatrix} = \begin{pmatrix} B\gamma arg \begin{pmatrix} Max_A - 1, \\ old\gamma s, \\ [], \\ mod\_val, \\ 0, \\ \Xi^{-1} \end{pmatrix}, new\gamma s \end{pmatrix}$$

Finally, the second case definition of $B\gamma s$ operation is defined for the case of module zero as:

$$B\gamma s \begin{pmatrix} 0, \\ old\gamma s, \\ new\gamma s, \\ Max_N, \\ Max_A, \\ \Xi^{-1} \end{pmatrix} = \begin{pmatrix} Build\gamma \begin{pmatrix} \gamma_0, \\ 0, \\ 0, \\ M \end{pmatrix}, \ldots, Build\gamma \begin{pmatrix} \gamma_{Max_N - 1}, \\ 0, \\ 0, \\ M \end{pmatrix}, new\gamma s \end{pmatrix}$$

β-wiring Operations

Consider the source abstract dSCA β-wiring function:

$$\beta_{pc\_val_1}(i_1, j_1) = z_1$$

the corresponding target abstract dSCA β-wiring function will be:

$$\beta_{pc\_val_2}(i_2, j_2) = z_2$$

where $j_1 = j_2$ and $\Xi(i_1, pc\_val_1) = (i_2, pc\_val_2)$ The *informal* process of generating target abstract dSCA β-wiring functions is to walk the structure of the target architecture creating wiring functions for all modules at all values of the program counter for the number of inputs to each module:

- For each module m_i where $i \in \mathbb{N}_{k_2}$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$:
    - For the o^{th} argument of each module create:

$$\beta'_{pc\_val}(i, 0) = M$$

    - For each argument where $j \in \{1, \ldots, n_2(i)\}$ create a new β-wiring function

$$\beta'_{pv\_val}(i, j) = \begin{cases} Value\ from\ source & \text{if } \Xi^{-1}(i, j) \downarrow \\ \omega & \text{otherwise} \end{cases}$$

      with the intended meaning that the undefined index is given if the inverse mapping is not defined, otherwise the appropriate value from the source network is used.

- For module 0 create Max_N β-wiring functions to wire m_0 back to itself.

Formally, the $Create\beta s$ operation is introduced:

$$Create\beta s : SCAAlgebra \times N^2 \times MapEqList \rightarrow \beta dSCAEqList$$

and it is defined as:

$$Create\beta s \begin{pmatrix} Source\_SCA, \\ k, \\ Max_N, \\ \Xi^{-1} \end{pmatrix} = B\beta s \begin{pmatrix} k, \\ Get\beta Eqs(source\_SCA), \\ [], \\ Max_N, \\ GetMaxA(source\_SCA), \\ \Xi^{-1} \end{pmatrix}$$

It takes as arguments the source abstract SCA, the number of modules in the target abstract SCA and then value of Max_N for that network as well as the inverse mapping function equation list. The maximum number of arguments that any module can take, Max_A, is calculated by the call to the $GetMaxA$ function - the impact of this is that more wiring functions may be generated than are

necessary but as these are wired to the unconnected module they will not partake in the functionality of the resultant target abstract dSCA.

The purpose of the $Create\beta s$ operation is to extract the relevant details out of the source abstract SCA and call the $B\beta s$ operation. The values that are extracted are the number of modules in the source abstract SCA, the source β-wiring functions and Max_A.

The $B\beta s$ operation:

$$B\beta s : N \times \beta dSCAEqList^2 \times N^2 \times MapEqList \rightarrow \beta dSCAEqList$$

is defined recursively over the number of modules in the target SCA in two cases, the first represents the case when the module number is 0 and the second case is where the module number is greater than 0. In the second case the $B\beta s$ is defined with the recursive call to itself:

$$B\beta s \begin{pmatrix} mod\_num, \\ old\beta s, \\ new\beta s, \\ Max_N, \\ Max_A, \\ \Xi^{-1} \end{pmatrix} = B\beta s \begin{pmatrix} mod\_num - 1, \\ old\beta s, \\ \begin{pmatrix} B\beta pc \begin{pmatrix} Max_N - 1, \\ old\beta s, \\ [], \\ mod\_num, \\ Max_A, \\ \Xi^{-1} \end{pmatrix}, new\beta s \end{pmatrix}, \\ Max_N, \\ Max_A, \\ \Xi^{-1} \end{pmatrix}$$

The recursive call contains an argument where a list is appended to the newly generated β-wiring functions for a module. This list is created by calling the $B\beta pc$ operation:

$$B\beta pc : N \times \beta dSCAEqList^2 \times N^2 \times MapEqList \rightarrow \beta dSCAEqList$$

which is defined recursively over the values that the program counter may take in two cases. The first case is where the program counter is 0 and the second is where the program counter is greater

than zero. In the second case $B\beta pc$ is defined as:

$$B\beta pc\begin{pmatrix} pc\_val, \\ old\beta s, \\ new\beta s, \\ mod\_num, \\ Max_A, \\ \Xi^{-1} \end{pmatrix} = B\beta pc\begin{pmatrix} pc\_val - 1, \\ old\beta s, \\ \begin{pmatrix} B\beta arg\begin{pmatrix} Max_A - 1, \\ old\beta s, \\ [], \\ mod\_num, \\ pc\_val, \\ \Xi^{-1} \end{pmatrix}, new\beta s \end{pmatrix}, \\ mod\_num, \\ Max_A, \\ \Xi^{-1} \end{pmatrix}$$

This operation recurses on itself building a list of new β-wiring functions for a module at a particular value of the program counter for all inputs to a module by calling the $B\beta arg$ operation. The $B\beta arg$ operation is given as:

$$B\beta arg : N \times \beta dSCAEqList^2 \times N^2 \times MapEqList \rightarrow \beta dSCAEqList$$

and is also defined recursively, this time over the argument number under consideration in two cases - where the argument index is zero, and where it is not. For the second case it is defined as:

$$B\beta arg\begin{pmatrix} arg\_val, \\ old\beta s, \\ new\beta s, \\ mod\_num, \\ pc\_val, \\ \Xi^{-1} \end{pmatrix} = B\beta arg\begin{pmatrix} arg\_num - 1, \\ old\beta s, \\ \begin{pmatrix} B\beta\begin{pmatrix} mod\_num, \\ arg_val, \\ pc\_val, \\ old\beta s, \\ \Xi^{-1} \end{pmatrix}, new\beta s \end{pmatrix}, \\ mod\_num, \\ pc\_val, \\ \Xi^{-1} \end{pmatrix}$$

The $B\beta$ operation is used to construct the β-wiring function for this particular argument index at a particular program counter for a particular module. It is given as:

$$B\beta : N^3 \times \beta dSCAEqList \times MapEqList \rightarrow \beta dSCAEquation$$

To construct the β-wiring function it is first identified whether there exists a corresponding element in the source abstract dSCA. This is achieved by considering the inverse mapping function, if it is undefined for the values under consideration then its wiring in both the source and target abstract

SCAs are unimportant. In such a case the output of the $B\beta$ operation is defined to be the creation of a wiring function to the unconnected value ω, where $\Xi^{-1}(mod\_val, pc\_val)\uparrow$, as:

$$B\beta\begin{pmatrix} mod\_num, \\ arg\_val, \\ pc\_val, \\ old\beta s, \\ \Xi^{-1} \end{pmatrix} = Build\beta\begin{pmatrix} \beta_{pc\_val}, \\ mod\_num, \\ arg\_val, \\ \omega \end{pmatrix}$$

Where the inverse mapping is defined, the corresponding wiring function in the source abstract dSCA can be identified as:

$$\beta_{snd(\Xi^{-1}(mod\_num,pc\_val))}(fst(\Xi^{-1}(mod\_num, pc\_val)), arg\_val)$$

The β-wiring operation in the target dSCA is therefore constructed as:

$$\beta_{pc\_val}(mod\_num, arg\_val) = RetTerm\left(GetEl\begin{pmatrix} old\beta s, \\ fst(\Xi^{-1}(mod\_num, pc\_val)), \\ arg\_val, \\ snd(\Xi^{-1}(mod\_num, pc\_val)) \end{pmatrix}, 2\right)$$

the $B\beta$ operation, where $\Xi^{-1}(mod\_val, pc\_val)\downarrow$ can therefore be defined as:

$$B\beta\begin{pmatrix} mod\_num, \\ arg\_val, \\ pc\_val, \\ old\beta s, \\ \Xi^{-1} \end{pmatrix} = Build\beta\begin{pmatrix} \beta_{pc\_val}, \\ mod\_num, \\ arg\_val, \\ RetTerm\left(GetEl\begin{pmatrix} old\beta s, \\ i_{old}, \\ arg\_val, \\ pc_{old} \end{pmatrix}, 2\right) \end{pmatrix}$$

where:

$$i_{old} = fst(RetTerm(GetEl(\Xi^{-1}, mod\_num, pc\_val)), 2)$$
$$pc_{old} = snd(RetTerm(GetEl(\Xi^{-1}, mod\_num, pc\_val)), 2)$$

The second case of the $B\beta arg$ operation, where the argument index is zero, simply generates a β-wiring function for the 0^{th} argument, which will be to the program counter, and appends it to the list of functions generated for that module and returns the new list. It is defined as:

$$B\beta arg\begin{pmatrix} 0, \\ old\beta s, \\ new\beta s, \\ mod\_val, \\ pc\_val, \\ \Xi^{-1} \end{pmatrix} = \left(Build\beta\begin{pmatrix} \beta_{pc\_val}, \\ mod\_num, \\ 0, \\ pc \end{pmatrix}, new\beta s\right)$$

The second case of the $B\beta pc$ operation, where the program counter is 0, simply generates the β-wiring functions for module mn at $pc\_val = 0$ and appends them to the list of already generated β-wiring functions for module mn at all other values of the program counter. It is defined as:

$$B\beta pc \begin{pmatrix} 0, \\ old\beta s, \\ new\beta s, \\ mod\_val, \\ Max_A, \\ \Xi^{-1} \end{pmatrix} = \begin{pmatrix} B\beta arg \begin{pmatrix} Max_A - 1, \\ old\beta s, \\ [], \\ mod\_val, \\ 0, \\ \Xi^{-1} \end{pmatrix}, n\beta s \end{pmatrix}$$

Finally, the second definition of $B\beta s$ operation is defined for the case of module zero, or the program counter. In this case there is only one β-wiring function for each value of the program counter:

$$B\beta s \begin{pmatrix} 0, \\ old\beta s, \\ new\beta s, \\ Max_N, \\ Max_A, \\ \Xi^{-1} \end{pmatrix} = \begin{pmatrix} Build\beta \begin{pmatrix} \beta_0, \\ 0, \\ 0, \\ pc \end{pmatrix}, \ldots, Build\beta \begin{pmatrix} \beta_{Max_N - 1}, \\ 0, \\ 0, \\ pc \end{pmatrix}, new\beta s \end{pmatrix}$$

G.1.4 Delay Functions

The delay functions for the source and target abstract dSCA are of the same format, however the derivation of the delay is more complicated than the simple generation of the wiring functions, and thus a more detailed explanation of the derivation is given.

In both networks, it is the intention of the delay function to indicate the time delay between now and the time the result was calculated. In the source abstract dSCA this is given by the defined delay function. For the object abstract dSCA this value needs to be derived from the data available.

Informally, target abstract dSCA functions are produced as follows:

- For each module m_i where $i \in \mathbb{N}_{k_2}$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$:
    - We define, for the 0^{th} argument, the unit delay:

$$\delta'_{i,0,pc\_val}(t, a, x) = t - 1$$

* For each argument where $j \in \{1, \ldots, n_2(i)\}$ create a new δ-wiring function

$$\delta_{pc\_val}(i,j) = \begin{cases} t - new\_value & \text{if } (\Xi^{-1}(i, pc\_val)) \downarrow \wedge \\ & (\gamma_{pc\_val}(i,j) = M) \\ t-1 & \text{otherwise} \end{cases}$$

- For module 0 create Max_N delay operations of unit length delay to represent the wiring of m_0 back to itself.

Formally, the new delay functions are created by calling the *Createδs* operation, given as:

$$Create\delta s : dSCAAlgebra \times N^2 \times MapList^2 \rightarrow \delta dSCAEqList$$

where the first argument is the specification defining the source abstract dSCA, the second and third argument describe the defining shape of the target abstract dSCA, and the final 2 arguments the mapping and its inverse. *Createδs* is defined as:

$$Create\delta s \begin{pmatrix} Source\_SCA, \\ k, \\ Max_N, \\ \Xi, \\ \Xi^{-1}, \end{pmatrix} = B\delta s \begin{pmatrix} k, \\ Get\delta Eqs(Source\_SCA), \\ GetMaxA(Source\_SCA)[], \\ Get\gamma Eqs(Source\_SCA), \\ Create\beta s \begin{pmatrix} Source\_SCA, \\ k, \\ Max_N, \\ \Xi^{-1} \end{pmatrix}, \\ GetMax_N(Source\_SCA), \\ Max_N, \\ \Xi, \\ \Xi^{-1}, \end{pmatrix}$$

The $B\delta s$ operation is defined recursively over the number of modules in the target abstract dSCA. There are two cases, the first where the module index under consideration is greater than 0 and the second case where the index is 0. $B\delta s$ is given as:

$$\begin{aligned} B\delta s: \quad & N \times \delta dSCAEqList^2 \times N \times \gamma dSCAEqList \times \\ & \beta dSCAEqList \times N^2 \times MapEqList^2 \rightarrow \delta dSCAEqList \end{aligned}$$

In the first case, $B\delta s$ is defined as:

$$B\delta s \begin{pmatrix} mod\_val, \\ old\delta s, \\ new\delta s, \\ Max_A, \\ old\gamma s, \\ new\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} = B\delta s \begin{pmatrix} mod\_val - 1, \\ old\delta s, \\ \begin{pmatrix} B\delta pc \begin{pmatrix} Max_N - 1, \\ old\delta s, \\ [], \\ mod\_val, \\ Max_A, \\ old\gamma s, \\ new\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi \end{pmatrix}, new\delta s \end{pmatrix}, \\ Max_A, \\ old\gamma s, \\ new\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi \end{pmatrix}$$

The internal call to $B\delta pc$ creates a list of delay functions for a particular value of the program counter for module *mod_val*. $B\delta pc$ is given as:

$$\begin{aligned} B\delta pc: \quad & N \times \delta dSCAEqList^2 \times N^2 \times \gamma dSCAEqList \times \\ & \beta dSCAEqList \times N^2 \times MapEqList^2 \rightarrow \delta EqList \end{aligned}$$

It is defined recursively over values of the program counter in two cases: where the program counter

is not zero and where it is. For the first case $B\delta pc$ is define as:

$$B\delta pc\begin{pmatrix} pc\_val, \\ old\delta s, \\ new\delta s, \\ mod\_val, \\ Max_A, \\ old\gamma s, \\ new\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} = B\delta pc\begin{pmatrix} pc\_val - 1, \\ old\delta s, \\ \begin{pmatrix} B\delta arg\begin{pmatrix} Max_A - 1, \\ old\delta s, \\ [], \\ mod\_val, \\ pc\_val, \\ old\gamma s, \\ new\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi \end{pmatrix}, new\delta s \end{pmatrix}, \\ mod\_val, \\ Max_A, \\ old\gamma s, \\ new\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi \end{pmatrix}$$

The call to $B\delta arg$ enables the construction of a list of delay functions for the arguments of module mn at program counter value pc, it is given as:

$$\begin{aligned} B\delta arg: \quad & N \times \delta dSCAEqList^2 \times N^2 \times \gamma dSCAEqList \times \\ & \beta dSCAEqList \times N^2 \times MapEqList^2 \rightarrow \delta EqList \end{aligned}$$

It is defined recursively over the number of arguments for the module in two cases - where the argument index is not 0, and where it is 0. For the case where the argument index is not 0 then

$B\delta arg$ is defined as:

$$B\delta arg\begin{pmatrix} arg\_num, \\ old\delta s, \\ new\delta s, \\ mod\_val, \\ pc\_val, \\ old\gamma s, \\ new\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} = B\delta arg\begin{pmatrix} arg\_num - 1, \\ old\delta s, \\ \begin{pmatrix} B\delta\begin{pmatrix} mod\_val, \\ arg\_num, \\ pc\_val, \\ old\delta s, \\ old\gamma s, \\ new\beta s, \\ Max_N^{s}, \\ Max_N^{o}, \\ \Xi^{-1}, \\ \Xi, \end{pmatrix}, new\delta s \end{pmatrix}, \\ mod\_val, \\ pc\_val, \\ old\gamma s, \\ new\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi, \end{pmatrix}$$

Finally, the $B\delta$ operation, which is responsible for creating the new delay function for the j^{th} argument of module mn at program counter pc, is called and it is given as:

$$\begin{aligned} B\delta: \quad & N^3 \times \delta dSCAEqList \times \gamma dSCAEqList \times \beta dSCAEqList \times \\ & N^2 \times MapEqList^2 \rightarrow \delta dSCAEquation \end{aligned}$$

To provide a definition of $B\delta$ the new value of the delay needs to be generated from the existing knowledge of the two abstract SCAs. To understand what the delay should be, an understanding of the particular delay required is needed. If the wiring is to a source, or is unconnected, then the unit delay is generated. This case is identified by considering the target abstract dSCA β-wiring functions:

$$B\delta\begin{pmatrix} mod\_val, \\ arg\_val, \\ pc\_val, \\ old\delta s \\ old\gamma s, \\ new\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} = Build\delta\begin{pmatrix} mod\_val, \\ arg\_val, \\ pc\_val, \\ t-1 \end{pmatrix} \; if \; cond_1$$

where:

$$cond_1 = \left(RetTerm \left(GetEl \left(\begin{array}{l} old\gamma s, \\ pc\_val, \\ mod\_val, \\ arg\_val \end{array} \right), 2 \right) \neq M \right)$$

In the situation where this condition is not true, i.e. the wiring under consideration is to another module, then the value of the new delay function needs to be calculated. To calculate the new value, the following process is followed:

1. Find the module and program counter value in the source abstract dSCA that relates to the current module and program counter value in the target abstract dSCA, using the inverse mapping function;
2. Identify the module in the source abstract dSCA that produces the value we are interested in from the β-wiring function;
3. Identify the program counter value in the source abstract dSCA that the value we are interested in is calculated from the delay functions;
4. Find the module and program counter in the target dSCA that produces the value we are interested in, using the mapping function; and
5. Calculate the delay between the current value of the program counter and the program counter value from (4).

The module and program counter in the source abstract dSCA is given directly by the inverse mapping function:

$$\Xi^{-1}(mod\_val_2, pc\_val_2) = (mod\_val_1, pc\_val_1)$$

The position of arguments in the functional specification cannot change in the transformation. Thus if arg_val is the argument number under consideration in the target abstract dSCA, then it will also be in the source abstract dSCA. This fact and the β-wiring function in the source abstract dSCA are used to determine the module that produces the value for that argument, in the source SCA:

$$mod\_val_1^{res} = \beta_{pc\_val_1}(mod\_val_1, arg\_val)$$

Using the delay function from the source dSCA, the value of the program counter that the result was calculated at can be determined. It will be the current source program counter value minus the delay value for this argument modulus the value of Max_N in the source abstract dSCA:

$$pc\_val_1^{res} = (pc\_val_1 - (t - \delta^1_{mod\_val_1, arg\_val, pc\_val_1}(t, a, x))) \bmod Max_N^1$$

It is now possible to determine the value of the program counter in the target abstract dSCA by applying the mapping function to the values $pc\_val_1^{res}$ just determined, and $mod\_val_1^{res}$, and taking the second element of the returned tuple:

$$pc\_val_2^{res} = snd(\Xi(mod\_val_1^{res}, pc\_val_1^{res}))$$

The value of the delay can be worked out from the difference between the program counter in the target abstract dSCA now, and the value of $pc\_val_2^{res}$:

$$(pc\_val - pc\_val_2^{res}) \bmod Max_N^2$$

$B\delta$ is therefore defined as:

$$B\delta \begin{pmatrix} mod\_val, \\ arg\_val, \\ pc\_val, \\ old\delta s \\ old\gamma s, \\ old\beta s, \\ Max_N^{src}, \\ Max_N^{tgt}, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} = Build\delta \begin{pmatrix} mod\_val, \\ arg\_val, \\ pc\_val, \\ t - ((pc\_val - pc\_val_{tgt}^{res}) \bmod Max_N^{tgt}) \end{pmatrix}$$

and:

$$pc\_val_{tgt}^{res} = snd\left(RetTerm\left(GetEl\begin{pmatrix} \Xi, \\ mod\_val_{src}^{res}, \\ pc\_val_{src}^{res} \end{pmatrix}, 2\right)\right)$$

with:

$$mod\_val_{src}^{res} = fst\left(RetTerm\left(GetEl\begin{pmatrix} old\beta s, \\ pc\_val_{src}, \\ mod\_val_{src}, \\ arg\_val \end{pmatrix}, 2\right)\right)$$

and:

$$pc\_val_{src}^{res} = pc\_val_{src} - \left(t - GetEl\begin{pmatrix} old\delta s, \\ mod\_val_{src}, \\ arg\_val, \\ pc\_val_{src} \end{pmatrix}\right) \bmod Max_N^{src}$$

where mod_val_{src} and pc_val_{src} are:

$$pc\_val_{src} = snd\left(RetTerm\left(GetEl\left(\begin{array}{l}\Xi^{-1},\\ mod\_val,\\ pc\_val\end{array}\right),2\right)\right)$$

$$mod\_val_{src} = fst\left(RetTerm\left(GetEl\left(\begin{array}{l}\Xi^{-1},\\ mod\_val,\\ pc\_val\end{array}\right),2\right)\right)$$

The second case of $B\delta arg$, where $arg\_num = 0$, is a case of returning the list of delay functions already generated with the delay for the channel to the program counter module added:

$$B\delta arg\left(\begin{array}{l}0,\\ old\delta s,\\ new\delta s,\\ mn,\\ pc,\\ \Xi^{-1},\\ \Xi,\\ old\gamma s,\\ old\beta s,\\ Max_N^{src},\\ Max_N^{tgt}\end{array}\right) = \left(Build\delta\left(\begin{array}{l}mn,\\ 0,\\ pc,\\ t-1\end{array}\right), new\delta s\right)$$

The second case of $B\delta pc$, where $pc\_val = 0$, is where the delay functions for module mn at program counter 0 are appended to the already constructed delay functions:

$$B\delta pc\left(\begin{array}{l}0,\\ old\delta s,\\ new\delta s,\\ mod\_val,\\ Max_A,\\ \Xi^{-1},\\ \Xi,\\ old\gamma s,\\ Max_N^{src},\\ Max_N^{tgt}\end{array}\right) = \left(\begin{array}{l}B\delta arg\left(\begin{array}{l}Max_A - 1,\\ old\delta s,\\ [],\\ mod\_val,\\ 0,\\ \Xi^{-1},\\ \Xi,\\ old\gamma s,\\ Max_N^{src},\\ Max_N^{tgt}\end{array}\right),\\ neqs\end{array}\right)$$

Finally, the second case of $B\delta s$ manages the situation where all the modules have been addressed, except for module 0. In this circumstance, the delay functions already constructed are returned, in

addition to a delay function for the program counter, m_0, as follows:

$$B\delta s \begin{pmatrix} 0, \\ old\delta s, \\ new\delta s, \\ Max_N, \\ Max_A, \\ \Xi^{-1}, \\ \Xi, \\ old\gamma s, \\ Max_N^s, \\ Max_N^o \end{pmatrix} = \left(Build\delta \begin{pmatrix} 0, \\ 0, \\ 0, \\ t-1 \end{pmatrix}, \ldots, Build\delta \begin{pmatrix} 0, \\ 0, \\ Max_N - 1, \\ t-1 \end{pmatrix}, new\delta s \right)$$

G.1.5 Initial State Equations

Consider the target abstract dSCA module m_i, its Initial State equations, will be of the form:

$$\begin{aligned} V_i(0, a, x) &= x_{i,0} \\ V_i(0, a, x) &= x_{i,1} \\ &\vdots \\ V_i(0, a, x) &= x_{i,Max_N-1} \end{aligned}$$

where each value x_{i,pc_val}, where $pc\_val = 0, 1, \ldots, Max_N - 1$, will either be the undefined element, or will come from some particular module and value of the source abstract dSCA program counter. Values of the source program counter and module are given directly from the mapping function, Ξ.

Informally, the set of Initial State equations is created as follows:

- For each module m_i where $i \in \mathbb{N}_{k_2}$ and $i > 0$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$ create a new Initial State equation:

$$V_i(pc\_val, a, x) = \begin{cases} new\_value & \text{if } \Xi^{-1}(i, pc)) \downarrow \\ u & \text{otherwise} \end{cases}$$

- For m_0, the program counter:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$ create a new Initial State equation:

$$V_0(pc\_val, a, x) = (pc\_val + 1) \bmod Max_{N_2}$$

Formally, the $CreateIVs$ operation is introduced as:

$$CreateIVs : dSCAAlgebra \times N^2 \times MapEqList \rightarrow dSCAISVEqList$$

where the arguments are such that it takes a source abstract dSCA specification and values giving the defining shape of the target abstract dSCA to produce the Initial State equations for the target dSCA. We define the operation as:

$$CreateIVs \left(\begin{array}{l} Source\_SCA, \\ k, \\ Max_N, \\ \Xi^{-1} \end{array} \right) = BIVs \left(\begin{array}{l} k, \\ Max_N, \\ GetEqIV(Source\_SCA), \\ [], \\ \Xi^{-1} \end{array} \right)$$

The purpose of the operation, $BIVs$, called by $CreateIVs$, is to build the new Initial State equations for the network. It is given as:

$$BIVs : N^2 \times dSCAISVEqList^2 \times MapEqList \rightarrow dSCAISVEqList$$

and is defined recursively over the number of modules in the target abstract dSCA. There are two cases: where the module number under consideration is greater than 0, and where the module number is zero. In the first case $BIVs$ is defined to recurse on itself, decrementing the module number and adding the result of calling the BIV operation to the list of new Initial State equations:

$$BIVs \left(\begin{array}{l} num\_mod, \\ Max_N, \\ oeqs, \\ neqs, \\ \Xi^{-1} \end{array} \right) = BIVs \left(\begin{array}{l} num\_mod - 1, \\ Max_N, \\ oeqs, \\ \left(BIVpc \left(\begin{array}{l} Max_N - 1, \\ num\_mod, \\ oeqs, \\ [], \\ \Xi^{-1} \end{array} \right), neqs \right), \\ \Xi^{-1} \end{array} \right)$$

This operation makes a call to the $BIVpc$ operation to generate the list of Initial State equations for all values of the program counter for module *num_mod*. Where $BIVpc$ is given by:

$$BIVpc : N^2 \times dSCAISVEqList^2 \times MapEqList \rightarrow dSCAISVEqList$$

where the first argument is the program counter value, the second argument the module number under consideration, the third argument the list of Initial State equations from the source abstract

dSCA, the fourth argument is the list of new Initial State equations that are being recursively created and the final argument is the inverse mapping function.

BIVpc is defined recursively over program counter values with two cases, the first representing the case where the program counter is greater than zero and the second case is where the program counter is zero. The first case is defined for two situations, where the inverse mapping is defined (in which case a new equation is created from values in the source abstract dSCA) and where it is not (in which case an equation is created that returns the undefined value u):

$$BIVpc\begin{pmatrix} pc, \\ i, \\ oeqs, \\ neqs, \\ \Xi^{-1} \end{pmatrix} = \begin{cases} BIVpc\begin{pmatrix} pc-1, \\ i, \\ \begin{pmatrix} new\_val, \\ neqs \end{pmatrix}, \\ \Xi^{-1} \end{pmatrix} & \text{if } \Xi^{-1}(i,pc)\downarrow \\ BIVpc\begin{pmatrix} pc-1, \\ i, \\ \begin{pmatrix} BuildIV\begin{pmatrix} i, \\ pc, \\ u \end{pmatrix}, \\ neqs \end{pmatrix}, \\ \Xi^{-1} \end{pmatrix} & \text{if } \Xi^{-1}(i,pc)\uparrow \end{cases}$$

where:

$$new\_val = BuildIV\begin{pmatrix} i, \\ pc, \\ RetTerm\left(GetEl\left(oeqs, RetTerm(GetEl(\Xi^{-1}, i, pc), 2)\right), 2\right) \end{pmatrix}$$

The second case of *BIVpc*, where the program counter is zero is the simple case of creating the equation for that value of the program counter and appending it to the list of already generated Initial State equations:

$$BIVpc\begin{pmatrix} 0, \\ i, \\ oeqs, \\ neqs, \\ \Xi^{-1} \end{pmatrix} = \begin{pmatrix} BuildIV\begin{pmatrix} i, \\ RetTerm\begin{pmatrix} GetEl(oeqs, \Xi^{-1}(i,0)), \\ 2 \end{pmatrix} \end{pmatrix}, \\ neqs \end{pmatrix}$$

The second case of *BIVs* returns the list of already generated Initial State equations, with the list of functions for the program counter appended to the front of them:

$$BIVs\begin{pmatrix} 0, \\ Max_N, \\ oeqs, \\ neqs, \\ \Xi^{-1} \end{pmatrix} = (BpcIVs(Max_N, [], Max_N), neqs)$$

$BpcIVs$ is given as:

$$BpcIVs : N \times dSCAISVEqList \times N \rightarrow dSCAISVEqList$$

and is recursively defined using two cases over the program counter values as:

$$BpcIVs\left(\begin{array}{l} 0, \\ neqs, \\ Max_N \end{array}\right) = \left(BuildIV\left(\begin{array}{l} 0, \\ 0, \\ 1 \end{array}\right), eqs\right)$$

$$BpcIVs\left(\begin{array}{l} pc\_val, \\ neqs, \\ Max_N \end{array}\right) = BpcIVs\left(\begin{array}{l} pc\_val - 1, \\ \left(BuildIV\left(\begin{array}{l} 0, \\ pc\_val, \\ mod\left(\begin{array}{l} pc\_val + 1, \\ Max_N \end{array}\right) \end{array}\right), eqs\right), \\ Max_N \end{array}\right)$$

G.1.6 State Transition Equations

Consider the target abstract dSCA module m_i, its State Transition equations, will be of the form:

$$V_i(t+1, a, x) = \begin{cases} f_{i,0}(\ldots) & \text{if } pc = 0 \\ \ldots & \\ f_{i,Max_N-1}(\ldots) & \text{if } pc = Max_N - 1 \end{cases}$$

where each functional specification component f_{i,pc_val}, for values of $pc\_val = 0, 1, \ldots, Max_N - 1$, will either be the undefined element, or will be the component specification extracted from some particular module and value of the source abstract dSCA program counter in the source abstract dSCA. In a similar manner to creating the Initial State equations, values of the program counter and module number in the source abstract dSCA for values in the target abstract dSCA are provided by the inverse mapping function, Ξ^{-1}.

- For each module m_i where $i \in \mathbb{N}_{k_2}$ and $i > 0$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$ in abstract dSCA extract and rewire the relevant functional specifications from the source abstract dSCA, if one exists, otherwise use the undefined constant u.
  - Create a new State Transition equation from the previous result.
- For m_0, the program counter:

– Create the program counter State Transition equation:

$$V_{pc}(t+1,a,x) = \begin{cases} mod(add(V_{pc}(t,a,x),1),Max_N) \;\; if \; V_{pc}(t-1,a,x) = 0 \\ \vdots \\ mod(add(V_{pc}(t,a,x),1),Max_N) \;\; \vdots \end{cases}$$

Formally the *CreateSTs* function is introduced as:

$$CreateSTs : SCAAlgebra \times N^2 \times MapEqList^2 \rightarrow dSCASTVEqList$$

CreateSTs takes a source abstract dSCA specification and values for the defining shape of the target abstract dSCA and produces the State Transition equations of that target dSCA. It is defined as:

$$CreateSTs\begin{pmatrix} Source\_SCA, \\ k, \\ Max_N, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} = BSTs\begin{pmatrix} k, \\ Max_N, \\ GetEqSTVF(Source\_SCA), \\ [], \\ \Xi^{-1}, \\ Create\beta s\begin{pmatrix} Source\_SCA, \\ k, \\ Max_N, \\ \Xi^{-1} \end{pmatrix}, \\ Create\delta s\begin{pmatrix} Source\_SCA, \\ k, \\ Max_N, \\ \Xi, \\ \Xi^{-1}, \end{pmatrix} \end{pmatrix}$$

The operation called by *CreateSTs* is the *BSTs* operation which is given as:

$$\begin{aligned} BSTs: \;\; & N^2 \times dSCASTVEqList^2 \times MapList \times \\ & \beta dSCAEqList \times \delta dSCAEqList \rightarrow dSCASTVEqList \end{aligned}$$

and is defined recursively over the set of module numbers. In keeping with a number of these transformation operations it has two cases, the first where the module number is greater than 0, and the second where it is 0. The first case takes as arguments, the module number under consideration, the value of Max_N, the list of source abstract dSCA State Transition equations, the target State Transition equations, the inverse mapping, and finally the target abstract dSCAs β-wiring and delay

functions. It is defined as:

$$BSTs\begin{pmatrix} mod\_num, \\ Max_N, \\ STVFs, \\ neqs, \\ \Xi^{-1}, \\ tgt\beta s, \\ tgt\delta s \end{pmatrix} = BSTs\begin{pmatrix} mod\_num - 1, \\ \begin{pmatrix} BST\begin{pmatrix} Max_N, \\ STVFs, \\ \Xi^{-1}, \\ tgt\beta s, \\ tgt\delta s, \\ mod\_num \end{pmatrix}, \\ neqs \end{pmatrix}, \\ Max_N \end{pmatrix}$$

The operation BST used in the above definition is given as:

$$\begin{aligned} BST: \quad & N^2 \times dSCASTVEqList \times MapList \times \\ & \beta dSCAEqList \times \delta dSCAEqList \rightarrow dSCASTVEqList \end{aligned}$$

and is defined such that a new equation is built up for module *mod_val* under consideration. It is defined as:

$$BST\begin{pmatrix} Max_N, \\ mod\_val, \\ STVFs, \\ \Xi^{-1}, \\ \beta s, \\ \delta s, \end{pmatrix} = BuildST\begin{pmatrix} mod\_val, \\ NewST\begin{pmatrix} rewire\begin{pmatrix} new\_vfopdef, \\ mod\_val, \\ pc\_val, \\ \beta eqs, \\ \delta eqs \end{pmatrix}, \\ Max_N, \\ null \end{pmatrix} \end{pmatrix}$$

where:

$$new\_vfopdef = NST\begin{pmatrix} Max_N, \\ mod\_val, \\ [], \\ STVFs, \\ \Xi^{-1} \end{pmatrix}$$

The $NewST(rewire(NST(\ldots)))$ component of the above definition needs some explaining. Consider that the VFOPDef term of a Value Function equation for an abstract dSCA is of the form:

$$f_i(pc, \ldots) = \begin{cases} f_{i,0}(\ldots) & \text{if pc} = 0 \\ f_{i,1}(\ldots) & \text{if pc} = 1 \\ \vdots & \\ f_{i,Max_N-1}(\ldots) & \text{if pc} = \text{Max}_\text{N} - 1 \end{cases}$$

It has already been noted that this is a convenient syntactic way of writing the conditional. If

written according to the machine algebra, M_A, it would appear as:

$$f_i(pc,\ldots) = cond\left(\begin{array}{l} pc = 0, \\ f_{i,0}(\ldots), \\ cond\left(\begin{array}{l} pc = 1, \\ f_{i,1}(\ldots), \\ cond\left(\ldots, cond\left(\begin{array}{l} pc = Max_N - 1, \\ f_{i,Max_N-1}(\ldots), \\ null \end{array}\right)\ldots\right) \end{array}\right) \end{array}\right)$$

It is this second form that is used to select the component specification based on a particular value of the program counter. To do so, the operation $GetFn$ is introduced:

$$GetFn : VFOpDefTerm \times N \rightarrow Term$$

and is defined recursively over the structure of the VFOpDef term definition:

$$\begin{array}{l} GetFn(cond(a,b,c),0) = b \\ GetFn(cond(a,b,c),pc_{req}) = GetFn(c,pc_{req}-1) \end{array}$$

To generate a target abstract dSCA State Transition equation for a module a list of the appropriate VFOpDef Terms, selected from the source abstract dSCA by means of the inverse mapping function Ξ^{-}, the $GetEl$ operation for STEqList specifications and the $GetFn$ operation defined above are used. Consider module m_i in the target abstract dSCA, at program counter value *pc_val* it is defined to be executing either the:

1. VFOpDef term in module $fst(\Xi^{-1}(i, pc\_val))$ at the source program counter value $snd(\Xi^{-1}(i, pc\_val))$ in the source abstract dSCA, if the mapping is defined; or
2. the output u, if the mapping is undefined.

The NST operation is introduced to determine which case is under consideration, and it is given as:

$$NST : N^2 \times dSCASTVEqList^2 \times MapEqList \rightarrow VFOpDefList$$

and recurses over the program counter values to produce a list of VFOpDef terms that are used for the definition of the State Transition phase of the Value Function for a particular module. It is

defined:

$$NST\left(\begin{array}{l} pc\_val, \\ mod\_val, \\ neqs, \\ oeqs, \\ \Xi^{-1} \end{array}\right) = NST\left(\begin{array}{l} pc\_val - 1, \\ mod\_val, \\ \left(neqs, Extract\left(\begin{array}{l} oeqs, \\ mod\_val_{src}, \\ pc\_val_{src} \end{array}\right)\right), \\ oeqs, \\ \Xi^{-1} \end{array}\right)$$

where:

$$mod\_val_{src} = fst\left(RetTerm\left(GetEl\left(\begin{array}{l} \Xi^{-1}, \\ mod\_val, \\ pc\_val \end{array}\right), 2\right)\right)$$

and:

$$pc\_val_{src} = snd\left(RetTerm\left(GetEl\left(\begin{array}{l} \Xi^{-1}, \\ mod\_val, \\ pc\_val \end{array}\right), 2\right)\right)$$

The *Extract* function used in the above definition is given as:

$$Extract : dSCASTVEqList \times N^2 \rightarrow VFOpDefTerm$$

and is defined as:

$$Extract\left(\begin{array}{l} oeqs, \\ mod\_val, \\ pc\_val \end{array}\right) = GetFn\left(\begin{array}{l} GetEl(oeqs, mod\_val), \\ pc\_val \end{array}\right)$$

The second case of the NST operation is defined as returning the list of VFOpDef terms constructed by appending the value for the program counter at 0 to those VFOpDef terms already obtained:

$$NST\left(\begin{array}{l} 0, \\ mod\_val, \\ neqs, \\ oeqs, \\ \Xi^{-1} \end{array}\right) = \left(neqs, Extract\left(\begin{array}{l} oeqs, \\ mod\_val_{src}, \\ pc\_val_{src} \end{array}\right)\right)$$

where mod_val_{src} and pc_val_{src} are as defined for the first case of NST. The result of NST is to produce a list of VFOpDef terms, however these terms will all be wired based on the values in the source abstract dSCA and must be rewired. Rewiring is accomplished with the *rewire* operation, whose purpose is to recurse down a list of VFOpDef terms, producing a new list of VFOpDef terms with wiring and delay functions put in place to reflect the target dSCA. Consistent with the definitions of the other transformations in this thesis simplification of the wiring and delay functions is applied in-situ.

The *rewire* operation is given as an operation that takes a VFOpDef list, the module number and the program counter value under consideration together with the list of *beta*-wiring and delay functions for the target abstract dSCA:

$$rewire : VFOpDefList \times N^2 \times \beta dSCAEqList \times \delta dSCAEqList \rightarrow VFOpDefList$$

The operation *rewire* is defined recursively over the list of VFOpDef terms with the first case being defined as:

$$rewire \begin{pmatrix} (e, es), \\ mod\_val, \\ pc\_val, \\ \beta eqs, \\ \delta eqs \end{pmatrix} = \begin{pmatrix} rw \begin{pmatrix} e, \\ mod\_val, \\ pc\_val, \\ \beta eqs, \\ \delta eqs \end{pmatrix}, rewire \begin{pmatrix} es, \\ mod\_val, \\ pc\_val - 1, \\ \beta eqs, \\ \delta eqs \end{pmatrix} \end{pmatrix}$$

and the second case is defined as:

$$rewire \begin{pmatrix} e, \\ mod\_val, \\ pc\_val, \\ \beta eqs, \\ \delta eqs \end{pmatrix} = rw \begin{pmatrix} e, \\ mod\_val, \\ pc\_val, \\ \beta eqs, \\ \delta eqs \end{pmatrix}$$

The operation *rw* used in *rewire* could be defined generically to take account of any number of arguments, but for clarity in this thesis, it is defined for the 4 cases that M_A will allow (zero to 3 arguments):

$$rw \begin{pmatrix} t, \\ \beta s, \\ \delta s, \\ mod\_val, \\ pc\_val \end{pmatrix} = t$$

$$rw \begin{pmatrix} t(t_1), \\ \beta s, \\ \delta s, \\ mod\_val, \\ pc\_val \end{pmatrix} = t \begin{pmatrix} wire \begin{pmatrix} t_1, \\ \beta s, \\ \delta s, \\ mod\_val, \\ 1, \\ pc\_val \end{pmatrix} \end{pmatrix}$$

$$rw \begin{pmatrix} t(t_1, t_2), \\ \beta s, \\ \delta s, \\ mod\_val, \\ pc\_val \end{pmatrix} = t \begin{pmatrix} wire \begin{pmatrix} t_1, \\ \beta s, \\ \delta s, \\ mod\_val, \\ 1, \\ pc\_val \end{pmatrix}, wire \begin{pmatrix} t_2, \\ \beta s, \\ \delta s, \\ mod\_val, \\ 2, \\ pc\_val \end{pmatrix} \end{pmatrix}$$

$$rw\left(\begin{array}{l} t(t_1,t_2,t_3), \\ \beta s, \\ \delta s, \\ mod\_val, \\ pc\_val \end{array}\right) = t\left(\begin{array}{l} wire(t_1,\beta s,\delta s,mod\_val,1,pc\_val), \\ wire(t_2,\beta s,\delta s,mod\_val,2,pc\_val), \\ wire(t_3,\beta s,\delta s,mod\_val,3,pc\_val) \end{array}\right)$$

with the supplementary operation *wire* being given as:

$$wire : Term \times \beta dSCAEqList \times \delta dSCAEqList \times N^3 \rightarrow Term$$

and defined for the three cases that may make up an atomic term within M_A:

$$\begin{array}{rcl} wire\left(\begin{array}{l} const, \\ \beta s,\delta s,i,j,pc \end{array}\right) & = & const \\ wire\left(\begin{array}{l} a_p(t), \\ \beta s,\delta s,i,j,pc \end{array}\right) & = & a_{new\_index}(t) \\ wire\left(\begin{array}{l} V_p(t-1,a,x), \\ \beta s,\delta s,i,j,pc \end{array}\right) & = & V_{new\_index}(new\_time,a,x) \end{array}$$

where:

$$new\_index = RetTerm(GetEl(\beta s, mod\_val, j, pc\_val), 2)$$

and:

$$new\_time = RetTerm(GetEl(\delta s, mod\_val, j, pc\_val), 2) + 1$$

To complete the generation of a State Transition equations for module m_{mod_val} in the target dSCA the list of rewired VFOpDef terms must be turned into the component specifications. This is done using the *NewST* operation, given as:

$$NewST : VFOpDefList \times N \rightarrow VFOpDef$$

which takes the list of VFOpDef terms (which has the VFOpDef term corresponding to $pc = Max_N - 1$ at the head and the VFOpDef term corresponding to $pc = 0$ at the end) and recurses down the list producing the appropriate target dSCA VFOpDef term. For the recursive case it is defined as:

$$NewST\left(\begin{array}{l} (e,es), \\ pc\_val, \\ neqs \end{array}\right) = NewST\left(\begin{array}{l} es, \\ pc\_val - 1, \\ cond(V_{pc}(t,a,x) = pc\_val, e, neqs) \end{array}\right)$$

and the base case is defined:

$$NewST\left(\begin{array}{l} e, \\ pc\_val, \\ neqs \end{array}\right) = cond(V_{pc}(t,a,x) = pc\_val, e, neqs)$$

The function call to create the new VFOpDef term for the target dSCA is therefore:

$$NewST\left(\begin{array}{l} rewire\left(\begin{array}{l} NSTS\left(\begin{array}{l} Max_N, \\ mod\_val, \\ [\,],\ oldeqs, \\ \Xi^{-1} \end{array}\right), \\ mod\_val, \\ pc\_val, \\ \beta eqs, \\ \delta eqs \end{array}\right), \\ Max_N, \\ null \end{array}\right)$$

which can be seen in the definition of the BST operation, wrapped by the value function building operation. This functionality is walked through in the section where we manually transform dSCAs.

The second case of $BSTs$, where the module is 0 is where the list of already generated state transition value functions is appended to the State Transition equations for the program counter, and is defined as:

$$BSTs\left(\begin{array}{l} 0, \\ Max_N, \\ neqs, \\ STVFs, \\ \Xi^{-1}, \\ \beta s, \\ \delta s \end{array}\right) = \left(\begin{array}{l} V_0(t+1,a,x) = \left\{\begin{array}{l} mod(add(V_0(t,a,x),1),Max_N) \text{ if } c_1 \\ \vdots \\ mod(add(V_0(t,a,x),1),Max_N) \text{ if } c_2 \end{array}\right., \\ neqs \end{array}\right)$$

where:

$$c_1 = V_{pc}(t,a,x) = 0$$

$$c_2 = V_{pc}(t,a,x) = Max_N - 1$$

G.1.7 Transformation Process

Each of the operations above need to be coordinated together so that a new abstract dSCA can be created by transforming the source abstract dSCA. The $Create\_adSCA$ operation is provided to do this, it is given as:

$$Transform : adSCAAlgebra \times N^2 \times MapEqList^2 \rightarrow adSCAAlgebra$$

The operation takes the source abstract dSCA and the defining shape of the target abstract dSCA together with the mapping and invers mapping functions.. It is defined:

$$Transform\begin{pmatrix} SCA_{src}, \\ k, \\ Max_N, \\ \Xi, \\ \Xi^{-1} \end{pmatrix} = CreateadSCA\begin{pmatrix} GetName(SCA_{src}), \\ adSCAAlgebra, \\ [], \\ [], \\ VFOp, \\ \gamma_0 : N^2 \to \{M, S, U\}, \\ \beta_0 : N^2 \to N, \\ \delta Op, \\ Create\gamma s\begin{pmatrix} SCA_{src}, \\ k, \\ Max_N, \\ \Xi^{-1} \end{pmatrix}, \\ Create\beta s\begin{pmatrix} SCA_{src}, \\ k, \\ Max_N, \\ \Xi^{-1} \end{pmatrix}, \\ Create\delta s\begin{pmatrix} SCA_{src}, \\ k, \\ Max_N, \\ \Xi, \\ \Xi^{-1}, \end{pmatrix}, \\ CreateIVs\begin{pmatrix} SCA_{src}, \\ k, \\ Max_N, \\ \Xi^{-1} \end{pmatrix}, \\ CreateSTs\begin{pmatrix} SCA_{src}, \\ k, \\ Max_N, \\ \Xi^{-1}, \\ \Xi \end{pmatrix} \end{pmatrix}$$

where:

$$VFOp = \begin{pmatrix} V_0 : T \times M_A^n \times M_A^k \to M_A, \\ \vdots, \\ V_k : T \times M_A^n \times M_A^k \to M_A \end{pmatrix}$$

$$\delta Op = \begin{pmatrix} \delta_{0,0,0} : T \times M_A^n \times M_A^k \to T, \\ \vdots, \\ \delta_{i,j,0} : T \times M_A^n \times M_A^k \to T \end{pmatrix}$$

and:

$$j = Get\_MaxA(Src\_SCA)$$
$$n = num\_inp(Src\_SCA)$$

Appendix H

Abstract dSCA to Concrete dSCA Transformation Details

H.1 Process

This appendix defines the processes for the transformation of an abstract dSCA with defining shape $\nabla = (n, m)$ to a concrete dSCA with a defining shape of $\nabla = (n, m)$. The following equation lists, within a supplied abstract dSCA specification, are considered for transformation:

1. Wiring Functions;
2. Delay Functions;
3. Initial State Equations; and
4. State Transition Equations.

Recall that this abstract dSCA has a defining shape of $\nabla = (1, 36)$. The transformation will be to a cycle consistent concrete dSCA. It should be noted that if transformation to a cycle inconsistent concrete dSCA was required then alteration of the tuple lengths and the use of appropriate tuple mapping functions (examples of which are given in Chapter 7) would have to be used.

H.1.1 Prerequisites

The following prerequisites are required for the transformation:

- The source and object networks have $k > 1$ modules and $Max_N > 0$ component specifications in their modules definitions;
- The defining shape of the target network equals that of the source network; and
- Condition definitions of each adSCA module, except the programme counter, are of the format:

$$cond(pc = 0, a, cond(pc = 1, b, cond(pc = 2, c, cond(...))))$$

H.1.2 γ-Wiring Functions

The γ-wiring functions in the target concrete dSCA will not differ much from those in the source abstract dSCA since the "look and feel" of the SCA is not being altered. What is different is the introduction of a new input to argument 1 which will require arguments $1, \ldots, n(i)$ of the abstract dSCA becoming arguments $2, \ldots, n(i) + 1$ in the concrete dSCA. The new argument introduced in concrete dSCA is a wiring of the first argument to the output of the module itself.

Informally, to generate the target concrete dSCA γ-wiring functions from a source abstract dSCA the following process is followed:

- For each module m_i where $i \in \mathbb{N}_{k_2}$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$:
    - For each argument where $j \in \{2, \ldots, n(i) + 1\}$ create a new β-wiring function

$$\gamma^2_{pc\_val}(i, j) = \gamma^1_{pc\_val}(i, j - 1)$$

    - For the o^{th} argument of each module create:

$$\gamma'_{pc\_val}(i, 0) = M$$

    - For the 1^{st} argument of each module create:

$$\gamma'_{pc\_val}(i, 1) = M$$

- For module 0 create Max_N β-wiring functions to wire m_0 back to itself.

Formally, the $Create\gamma s$ operation is introduced as:

$$Create\gamma s : adSCAAlgebra \rightarrow \gamma dSCAEqList$$

and is defined:

$$Create\gamma s\ (\ Source\_SCA\ ) = B\gamma s \begin{pmatrix} GetNumModules(Source\_SCA), \\ GetNumModules(Source\_SCA), \\ GetMaxN(Source\_SCA), \\ Get\gamma Eqs(Source\_SCA), \end{pmatrix}$$

The $B\gamma s$ operation achieves two purposes, first it calls the $Reindex\gamma s$ operation to manage the alteration of indexes, as described above, and then it calls the $Rewire\gamma s$ operation which is responsible for adding the new wiring function for argument 0 to all modules, except the program counter module, at all times of the program counter. $B\gamma s$ is given as:

$$B\gamma s : N^2 \times \gamma dSCAEqList \rightarrow \gamma dSCAEqList$$

taking as its first two arguments the defining shape of the concrete dSCA, and the third argument being the source abstract dSCA γ-wiring functions. It is defined as:

$$B\gamma s \begin{pmatrix} num\_mod, \\ Max_N, \\ old\gamma s, \end{pmatrix} = Rewire\gamma s \begin{pmatrix} num\_mod, \\ Max_N, \\ Reindex\gamma s\,(old\gamma s) \end{pmatrix}$$

where the operation $Reindex\gamma s$ is given as:

$$Reindex\gamma s : \gamma dSCAEqList \rightarrow \gamma dSCAEqList$$

and defined as:

$$Reindex\gamma s(e, es) = (Reindex\gamma(e), Reindex\gamma s(es))$$

and finally $Reindex\gamma$ is given as:

$$Reindex\gamma : \gamma dSCAEquation \rightarrow \gamma dSCAEquation$$

and is defined in two cases, the first where the wiring function is for the 0^{th} argument or the module is 0, and the second for where it is not. The first case is defined:

$$Reindex\gamma(e) = e \text{ if } \begin{pmatrix} RetArg(RetTerm(e,1),2) = 0 \vee \\ RetArg(RetTerm(e,1),1) = 0 \end{pmatrix}$$

In the second case a new γ-wiring function is created from the components of the source γ-wiring function, with the argument index incremented by one:

$$Reindex\gamma(e) = Build\gamma \begin{pmatrix} \gamma_{RetFn(RetTerm(e,1))}, \\ RetArg(RetTerm(e,1),1), \\ RetArg(RetTerm(e,1),2)+1, \\ RetTerm(e,2) \end{pmatrix}$$

Having shuffled the existing γ-wiring functions, the $Rewire\gamma s$ operation adds the additional γ-wiring functions for argument 1 for all values of the programme counter for all modules, except the program counter module. It is given as:

$$Rewire\gamma s : N^2 \times \gamma dSCAEqList \rightarrow \gamma dSCAEqList$$

and is defined recursively over the module number, in two cases. The first case is defined as:

$$Rewire\gamma s \begin{pmatrix} mod\_num, \\ pc\_val, \\ new\gamma s, \end{pmatrix} = Rewire\gamma s \begin{pmatrix} mod\_num - 1, \\ pc\_val, \\ \left(ReWire\gamma pc \begin{pmatrix} pc\_val - 1, \\ mod\_num, \\ [] \end{pmatrix}, new\gamma s \right) \end{pmatrix}$$

where the operation $Rewire\gamma pc$ used by the above definition is responsible for recursing over the values of the program counter and producing the actual wiring function. It is given as:

$$Rewire\gamma pc : N^2 \times \gamma dSCAEqList \rightarrow \gamma dSCAEqList$$

and is defined recursively over the program counter values. The first case is where the program counter is not 0, and it is therefore defined as:

$$Rewire\gamma pc \begin{pmatrix} pc\_val, \\ mod\_num, \\ new\gamma s \end{pmatrix} = Rewire\gamma pc \begin{pmatrix} pc\_val - 1, \\ mod\_num, \\ \left(Build\gamma \begin{pmatrix} mod\_num, \\ 1, \\ pc\_val, \\ M, \end{pmatrix}, new\gamma s \right) \end{pmatrix}$$

The second case of the $Rewire\gamma pc$ operation is defined as:

$$Rewire\gamma pc \begin{pmatrix} 0, \\ mod\_num, \\ new\gamma s \end{pmatrix} = \left(Build\gamma \begin{pmatrix} mod\_num, \\ 1, \\ 0, \\ M, \end{pmatrix}, new\gamma s \right)$$

The definition of the second case of the $Rewire\gamma s$, where the module number if 0 is defined to simply return back the list of newly generated equations:

$$Rewire\gamma s \begin{pmatrix} 0, \\ pc\_val, \\ old\gamma s, \\ new\gamma s \end{pmatrix} = new\gamma s$$

H.1.3 β-Wiring Functions

In a similar way to how the target concrete dSCA γ-wiring functions were constructed from source abstract dSCA γ-wiring functions, so are the concrete dSCA β-wiring functions. The β-wiring functions in the target concrete dSCA again differ only in so much that the index of arguments $1, \ldots, n(i)$ shifts to $2, \ldots, n(i)+1$.

Informally, to generate the target concrete dSCA β-wiring functions from a source abstract dSCA the following process is used:

- For each module m_i where $i \in \mathbb{N}_{k_2}$:
  - For each $pc\_val \in \{0, \ldots, Max_{N_2} - 1\}$:
    - For each argument where $j \in \{2, \ldots, n(i)+1\}$ create a new β-wiring function

$$\beta^2_{pc\_val}(i,j) = \beta^1_{pc\_val}(i, j-1)$$

    - For the o^{th} argument of each module create:

$$\beta'_{pc\_val}(i,0) = M$$

    - For the 1^{st} argument of each module create:

$$\beta'_{pc\_val}(i,1) = M$$

- For module 0 create Max_N β-wiring functions to wire m_0 back to itself.

Formally the $Create\beta s$ operation is introduced as:

$$Create\beta s : adSCAAlgebra \rightarrow \beta dSCAEqList$$

and is defined:

$$Create\beta s\left(\ Source\_SCA\ \right) = B\beta s\left(\begin{array}{l} GetNumModules(Source\_SCA), \\ GetMaxN(Source\_SCA), \\ Get\beta Eqs(Source\_SCA), \end{array}\right)$$

The $B\beta s$ operation achieves two purposes, first it calls the $Reindex\beta s$ operation to manage the alteration of indexes, as described above, and then it calls the $Rewire\beta s$ operation which is responsible for adding the new wiring function for argument 1 to all modules, except the program counter module, at all times of the program counter. $B\beta s$ is given as:

$$B\beta s : N^2 \times \beta dSCAEqList \rightarrow \beta dSCAEqList$$

taking as its first two arguments the defining shape of the concrete dSCA, and the third argument being the source abstract dSCA β-wiring functions. The final argument is the transformed β-wiring functions. It is defined as:

$$B\beta s\left(\begin{array}{l} num\_mod, \\ Max_N, \\ old\beta s, \end{array}\right) = Rewire\beta s\left(\begin{array}{l} num\_mod, \\ Max_N, \\ Reindex\beta s\,(old\beta s) \end{array}\right)$$

where the operation $Reindex\beta s$ is given as:

$$Reindex\beta s : \beta dSCAEqList \rightarrow \beta dSCAEqList$$

and defined as:

$$Reindex\beta s(e, es) = (Reindex\beta(e), Reindex\beta s(es))$$

and finally $Reindex\beta$ is given as:

$$Reindex\beta : \beta dSCAEquation \rightarrow \beta dSCAEquation$$

and is defined by two cases, the first where the wiring function is for the 0^{th} argument or the module is 0, and the second for where it is not. The first case is defined:

$$Reindex\beta(e) = e \text{ if } \left(\begin{array}{l} RetArg(RetTerm(e,1),2) = 0 \vee \\ RetArg(RetTerm(e,1),1) = 0 \end{array}\right)$$

In the second case, a new β-wiring function is created from the components of the source β-wiring function, with the argument index incremented by one:

$$Reindex\beta(e) = Build\beta\left(\begin{array}{l} \beta_{RetFn(RetTerm(e,1))}, \\ RetArg(RetTerm(e,1),1), \\ RetArg(RetTerm(e,1),2)+1, \\ RetTerm(e,2) \end{array}\right)$$

Having altered the indices of the existing β-wiring functions, the *Rewireβs* operation adds the additional β-wiring functions for argument 1 for all values of the programme counter for all modules, except the program counter module. It is given as:

$$Rewire\beta s : N^2 \times \beta dSCAEqList \rightarrow \beta dSCAEqList$$

and is defined recursively over the module number. Where the module number is not 0, then *Rewireβs* is defined as:

$$Rewire\beta s \begin{pmatrix} mod\_num, \\ pc\_val, \\ new\beta s, \end{pmatrix} = Rewire\beta s \begin{pmatrix} mod\_num - 1, \\ pc\_val, \\ \begin{pmatrix} ReWire\beta pc \begin{pmatrix} pc\_val - 1, \\ mod\_num, \\ [] \end{pmatrix}, l \\ new\beta s \end{pmatrix} \end{pmatrix}$$

the operation *Rewireβpc* used by the above definition is responsible for recursing over the values of the program counter and producing the actual wiring function. It is given as:

$$Rewire\beta pc : N^2 \times \beta dSCAEqList \rightarrow \beta dSCAEqList$$

and is defined recursively over the program counter values. The first case is where the program counter is not 0:

$$Rewire\beta pc \begin{pmatrix} pc\_val, \\ mod\_num, \\ new\beta s \end{pmatrix} = Rewire\beta pc \begin{pmatrix} pc\_val - 1, \\ mod\_num, \\ \begin{pmatrix} Build\beta \begin{pmatrix} mod\_num, \\ 1, \\ pc\_val, \\ mod\_num, \end{pmatrix}, new\beta s \end{pmatrix} \end{pmatrix}$$

The second case of the *Rewireβpc* operation is defined as:

$$Rewire\beta pc \begin{pmatrix} 0, \\ mod\_num, \\ new\beta s \end{pmatrix} = \begin{pmatrix} Build\beta \begin{pmatrix} mod\_num, \\ 1, \\ 0, \\ mod\_num, \end{pmatrix}, new\beta s \end{pmatrix}$$

The definition of the second case of the *Rewireβs*, where the module number is 0 is defined to return back the newly generated $\beta-$ wiring functions:

$$Rewire\beta s \begin{pmatrix} 0, \\ pc\_val, \\ old\beta s, \\ new\beta s \end{pmatrix} = new\beta s$$

H.1.4 Delay Functions

Delay functions for the concrete dSCA are all of unit delay, and there are a number equal to the wiring functions. Thus, a unit delay function will be created for every element in the newly generated γ-wiring equation list.

Formally, the $Create\delta s$ operation is introduced as:

$$Create\delta s : aSCAAlgebra \rightarrow \delta SCAEqList$$

Note that delay functions in the concrete dSCA are of the type $\delta SCAEqList$ and not $\delta dSCAEqList$. The $Create\delta s$ operation is defined as:

$$Create\delta s\left(\ Source\_SCA \ \right) = B\delta s \left(\begin{array}{l} Create\gamma s(Source\_SCA), \\ [] \end{array} \right)$$

The $B\delta s$ operation is defined recursively over the elements in the γ-wiring function list:

$$B\delta s : \gamma dSCAEqList \times \delta dSCAEqList \rightarrow \delta dSCAEqList$$

the case where the list is not a single element is defined as:

$$B\delta s \left(\begin{array}{l} (e, es), \\ neqs, \end{array} \right) = B\delta s \left(\begin{array}{l} es, \\ \left(Build\delta \left(\begin{array}{l} GetIndex(RetTerm(e,1),1), \\ GetArg(RetTerm(e,1),1), \\ GetArg(RetTerm(e,1),2), \\ t-1 \end{array} \right), neqs \right) \end{array} \right)$$

and the definition of $B\delta s$ where there is only one element in the list of γ-wiring functions is defined :

$$B\delta s \left(\begin{array}{l} e, \\ neqs, \end{array} \right) = Build\delta \left(\begin{array}{l} GetIndex(RetTerm(e,1),1), \\ GetArg(RetTerm(e,1),1), \\ GetArg(RetTerm(e,1),2), \\ t-1 \end{array} \right)$$

H.1.5 Initial State Equations

The initial states for each module m_i, where $1 \leq i \leq k$ are Max_N tuples of length Max_N (recall that the mapping is being defined for a cycle consistent abstract dSCA). We will make use of the fact that calculations will only care about the initial state given for $t = Max_N - 1$ and $t = 0$, by

defining the tuple at time $t = 0$ and use that value for all other initial values until $t = Max - N - 1$ where the final Initial State equation will be generated.

The operation $CreateIVs$ is introduced that takes the source abstract dSCA specification and produces the Initial State equations. It is given as:

$$CreateIVs : adSCAAlgebra \rightarrow dSCAISVEqList$$

and is defined as:

$$CreateIVs\left(\ Source\_SCA\ \right) = BIVs\left(\begin{array}{l} num\_modules(Source\_SCA), \\ GetMaxN(Source\_SCA), \\ GetIV(Source\_SCA), \\ [] \end{array}\right)$$

The call to the $BIVs$ operation is where the work of the transformation takes place. $BIVs$ is given as:

$$BIVs : N^2 \times dSCAISVEqList^2 \rightarrow dSCAISVEqList$$

and it is defined recursively over module numbers in two cases, the first is where the module number is greater than zero, and in such a case $BIVs$ is defined as:

$$BIVs\left(\begin{array}{l} mod\_num, \\ Max_N, \\ oeqs, \\ neqs \end{array}\right) = BIVs\left(\begin{array}{l} mod\_num - 1, \\ Max_N, \\ oeqs, \\ \left(\begin{array}{l} BIV\left(\begin{array}{l} 0, \\ Max_N - 1, \\ mod\_num, \\ oeqs, \\ [] \end{array}\right), neqs \end{array}\right) \end{array}\right)$$

The operation BIV:

$$BIV : N^3 \times dSCAISVEqList^2 \rightarrow dSCAISVEqList$$

is defined recursively in two cases over the first argument. Firstly for when the first argument does not equal Max_N, then the operation is dealing with an initial state from a time prior to $t = Max_N - 1$, and as such an initial state will be created containing u elements in all positions, except for 0^{th} element. Note that the positioning of the first element is dependant upon the tuple

management schemes used, however for both schemes identified as of interest the first generated value is placed at position 0 in the tuple. BIV is defined as:

$$BIV\begin{pmatrix} pc\_val, \\ Max_N, \\ mod\_num, \\ oeqs, \\ neqs \end{pmatrix} = BIV\begin{pmatrix} pc\_val + 1, \\ Max_N, \\ mod\_num, \\ oeqs, \\ \begin{pmatrix} GenIVs\begin{pmatrix} mod\_num, \\ pc\_val, \\ Max_N, \\ oeqs \end{pmatrix}, neqs \end{pmatrix}, \end{pmatrix}$$

The $GenIVs$ operation used in BIV is given as:

$$GenIVs : N^3 \times dSCAISVEqList \rightarrow dSCAISVEquation$$

and is defined to create a Max_N length tuple with the first element being the initial value produced at time $t = 0$ in the source abstract dSCA initial values:

$$GenIVs\begin{pmatrix} mod\_num, \\ pc\_val, \\ Max_N, \\ oeqs \end{pmatrix} = BuildIV\begin{pmatrix} mod\_num, \\ pc\_val, \\ \begin{pmatrix} RetTerm(VF, 2), \\ u_0, \ldots, u_{Max_N - 2} \end{pmatrix} \end{pmatrix}$$

where:

$$VF = GetEl(oeqs, mod\_num, pc\_val)$$

With the second case of BIV, where $t = Max_N - 1$, then the complete initial state needs to be generated (from previous values):

$$BIV\begin{pmatrix} Max_N, \\ Max_N, \\ mod\_num, \\ oeqs, \\ neqs \end{pmatrix} = \begin{pmatrix} BuildIV\begin{pmatrix} mod\_num, \\ pc\_val, \\ InitState\begin{pmatrix} Max_N, \\ mod\_num, \\ oeqs, \\ [] \end{pmatrix} \end{pmatrix}, neqs \end{pmatrix}$$

The operation $InitState$ is where the Initial State for module mod_num at time $t = Max_N - 1$ is created. Since we are using the array tuple management then the Initial State under these conditions will consist of a list of values with the first being the element calculated at $t = 0$ and the last being the one calculated at $t = Max_N$ in the source abstract dSCA. It is given as:

$$InitState : N^2 \times dSCAISVEqList \times TermList \rightarrow TermList$$

and is defined recursively, with the recursive case:

$$InitState\left(\begin{array}{l} pc\_val, \\ mod\_num, \\ oeqs, \\ nlist \end{array}\right) = InitState\left(\begin{array}{l} pc\_val - 1, \\ mod\_num, \\ oeqs, \\ (RetTerm\,(VF, 2)\,, nlist) \end{array}\right)$$

and the recursion being stopped by the 1^{st} argument reaching 0:

$$InitState\left(\begin{array}{l} 0, \\ mod\_num, \\ oeqs, \\ nlist, \\ \Xi^{-1} \end{array}\right) = (RetTerm\,(VF, 2)\,, nlist)$$

where in both cases:

$$VF = GetEl(oeqs, mod\_num, pc\_val)$$

The base call to the recursive $BIVs$ operation is defined as:

$$BIVs\left(\begin{array}{l} 0, \\ Max_N, \\ oeqs, \\ neqs \end{array}\right) = \left(BIVpc\left(\begin{array}{l} Max_N - 1, \\ [], \\ Max_N \end{array}\right), neqs\right)$$

where $BIVpc$ is given as:

$$BIVpc : N \times dSCAISVEqList \times N \rightarrow dSCAISVEqList$$

and is defined recursively over the values in Max_N, such that:

$$BIVpc\left(\begin{array}{l} pc\_val, \\ neqs, \\ Max_N \end{array}\right) = BIVpc\left(\begin{array}{l} pc\_val - 1, \\ \left(BuildIV\left(\begin{array}{l} 0, \\ Max_N, \\ pc\_val + 1 \bmod Max_N \end{array}\right), neqs\right), \\ Max_N \end{array}\right),$$

and:

$$BIVpc\left(\begin{array}{l} 0, \\ neqs, \\ Max_N \end{array}\right) = \left(BuildIV\left(\begin{array}{l} 0, \\ 0, \\ 1 \end{array}\right), neqs\right)$$

H.1.6 State Transition Equations

Consider the format of the State Transition equation in the source abstract dSCA, it will be similar to:

$$V_i(t, a, x) = \begin{cases} \vdots & \\ or(V_1(t-32, a, x), V_1(t-31, a, x)) & \text{if } \mathrm{V_{pc}(t-1, a, x) = 23} \\ gt(V_1(t-31, a, x), V_1(t-30, a, x)) & \text{if } \mathrm{V_{pc}(t-1, a, x) = 24} \\ \vdots & \end{cases}$$

the corresponding component specification in the concrete dSCA would be of the form:

$$V_i(t+1,a,x) = \begin{cases} \vdots & \\ \Upsilon \left(\begin{array}{l} V_{pc}(t,a,x), \\ V_1(t,a,x), \\ or \left(\begin{array}{l} \Pi^{tl}_{d_{1,1,23}}(V_1(t,a,x)), \\ \Pi^{tl}_{d_{1,2,23}}(V_1(t,a,x)) \end{array} \right) \end{array} \right) & \text{if } \mathrm{V_{pc}(t,a,x) = 23} \\ \Upsilon \left(\begin{array}{l} V_{pc}(t,a,x), \\ V_1(t,a,x), \\ gt \left(\begin{array}{l} \Pi^{tl}_{1,1,24}(V_1(t,a,x)), \\ \Pi^{tl}_{1,2,24}(V_1(t,a,x)) \end{array} \right) \end{array} \right) & \text{if } \mathrm{V_{pc}(t,a,x) = 24} \\ \vdots & \end{cases}$$

The differences are attributable to the introduction of the tuple management functions, Υ and Π (as well as the need to identify the value in the tuple that results are to be extracted from).

Informally, the process for creating the new State Transition equations is a two step process

- Generate the d functions - those that are used in the projection part of the tuple management functions
- Create the new State Transition equations.

Generation of the d functions

For an indexed array tuple management approach the results are stored relative to the value of the program counter when that result was calculated. The values of the d functions for each argument, given a cycle consistent dSCA, can be determined by using the following formula:

$$d_{mod\_num,arg\_num,pc\_val} = (Max_N + pc\_val - \delta_{mod\_num,arg\_num,pc\_val})$$

As an example, if a module has a definition:

$$V_i(t,a,x) = \begin{cases} \vdots & \\ cond \left(\begin{array}{l} V_1(t-34,a,x), \\ V_1(t-33,a,x), \\ V_1(t-32,a,x) \end{array} \right) & \text{if } \mathrm{V_{pc}(t-1,a,x) = 0} \\ \vdots & \end{cases}$$

Then its arguments would be stored at positions 1,2 and 3 in the array. Assuming $Max_N = 36$, then if the first argument is considered, $d_{1,2,0}$ can be determined as:

$$d_{1,2,0} = (36 + 0 - \delta_{1,2,0}) - 1$$

From the definition of the value function it can be seen that $\delta_{1,2,0}(t, a, x) = t - 34$, therefore:

$$\begin{aligned} d_{1,2,0} &= (36 + 0 - 34) - 1 \\ &= (2) - 1 \\ &= 1 \end{aligned}$$

To generate the d functions the *Createds* operation is introduced that recurses over the structure of the concrete dSCA (since the source abstract dSCA and concrete dSCA are the same "shape" means there is no requirement to use the mapping function). *Createds* is given as:

$$Createds : adSCAAlgebra \rightarrow ProjEqList$$

which is defined to take the abstract dSCA, defining shape of the target concrete dSCA and the number of arguments per module, and calls the *Bds* operation whilst extracting the δs equations from the source abstract dSCA:

$$createds\left(\ Source\_SCA\ \right) = Bds \begin{pmatrix} GetNumModules(Source\_SCA), \\ GetMaxN(Source\_SCA), \\ GetMaxA(Source\_SCA) + 1, \\ Get\delta Eqs(Source_SCA), \\ [] \end{pmatrix}$$

The *Bds* operation is given as:

$$Bds : N^3 \times \delta dSCAEqList \times ProjEqList \rightarrow ProjEqList$$

which is defined, in the recursive case, as:

$$Bds \begin{pmatrix} mod\_num, \\ Max_N, \\ Max_A, \\ oldeqs, \\ neqs \end{pmatrix} = Bds \begin{pmatrix} k - 1, \\ Max_N, \\ Max_A, \\ oldeqs, \\ \begin{pmatrix} Bdspc \begin{pmatrix} Max_N - 1, \\ mod\_num, \\ Max_A, \\ Max_N, \\ oldeqs, \\ [] \end{pmatrix}, neqs \end{pmatrix} \end{pmatrix}$$

In keeping with most of the definitions in the transformations so far, the *Bdspc* operation will recurse over the program counter values, and is defined as:

$$Bdspc : N^4 \times \delta dSCAEqList \times ProjEqList \rightarrow ProjEqList$$

this is also recursively defined, and the recursive case is as follows:

$$Bdspc \begin{pmatrix} pc\_val, \\ mod\_num, \\ Max_A, \\ Max_N, \\ oldeqs, \\ neqs \end{pmatrix} = Bdspc \begin{pmatrix} pc\_val - 1, \\ mod\_num, \\ Max_A, \\ Max_N, \\ oldeqs, \\ \begin{pmatrix} Bdsarg \begin{pmatrix} Max_A - 1, \\ pc\_val, \\ mod\_num, \\ Max_N, \\ oldeqs, \\ [] \end{pmatrix}, neqs \end{pmatrix} \end{pmatrix}$$

The *Bdsarg* is the operation that recurses over the arguments in a module:

$$Bdsarg : N^4 \times \delta dSCAEqList \times ProjEqList \rightarrow ProjEqList$$

Again, this is defined recursively, and the recursive case is as follows:

$$Bdsarg \begin{pmatrix} arg\_num, \\ mod\_num, \\ pc\_val, \\ Max_N, \\ oldeqs, \\ neqs \end{pmatrix} = Bdsarg \begin{pmatrix} arg\_num - 1, \\ mod\_num, \\ pc\_val, \\ Max_N, \\ oldeqs, \\ \begin{pmatrix} Buildd \begin{pmatrix} mod\_num, \\ Max_A, \\ pc\_val, \\ Max_N - 1, \\ d\_val, \end{pmatrix}, neqs \end{pmatrix} \end{pmatrix}$$

where:

$$d\_val = (Max_N + pc\_val) - \left(t - RetTerm \left(GetEl \begin{pmatrix} oldeqs, \\ mod\_val, \\ Max_A, \\ pc\_val, \end{pmatrix}, 2 \right) \right)$$

the base case of the *Bdsarg* operation, where the argument number is equal to 1 (since argument 0 and 1 are wired to the program counter and the module itself and therefore require no projection of

results) is defined as:

$$Bdsarg\begin{pmatrix} 1, \\ mod\_num, \\ pc\_val, \\ Max_N, \\ oldeqs, \\ neqs \end{pmatrix} = neqs$$

Note that the value of 1 is subtracted from the Max_A argument in the calculation of to reflect the fact that argument indexes in the source abstract dSCA are one behind those in the target concrete dSCA.

The base case of *Bdspc*, where the program counter value is 0 is defined:

$$Bdspc\begin{pmatrix} 0, \\ mod\_num, \\ Max_A, \\ Max_N, \\ oldeqs, \\ neqs \end{pmatrix} = \begin{pmatrix} Bdsarg\begin{pmatrix} Max_A, \\ pc\_val, \\ mod\_num, \\ Max_N, \\ oldeqs, \\ [] \end{pmatrix}, neqs \end{pmatrix}$$

and the base case of the *Bds* operation - where the module number is 0, simply returns the d functions already generated, since module 0 is the program counter and requires no such functions to be defined:

$$Bds\begin{pmatrix} 0, \\ Max_N, \\ Max_A, \\ oldeqs, \\ neqs \end{pmatrix} = neqs$$

Having produced the d functions for the new network attention can be returned to the generation of the State Transition equations. Consider again the format of the State Transition equation in the source abstract dSCA, it will be similar to:

$$V_i(t,a,x) = \begin{cases} \vdots & \\ or(V_1(t-32,a,x), V_1(t-31,a,x)) & \text{if cond1} \\ gt(V_1(t-31,a,x), V_1(t-30,a,x)) & \text{if cond2} \\ \vdots & \end{cases}$$

and the corresponding component specification in the concrete dSCA would be of the form:

$$V_i(t+1,a,x) = \begin{cases} \vdots & \\ \Upsilon\left(\begin{array}{l} V_{pc}(\delta_{1,0,23}(t,a,x),a,x), \\ V_1(\delta_{1,1,23}(t,a,x),a,x), \\ or\left(\begin{array}{l} \Pi^{tl}_{d_{1,2,23}}(V_1(\delta_{1,2,23}(t,a,x),a,x)), \\ \Pi^{tl}_{d_{1,3,23}}(V_1(\delta_{1,3,23}(t,a,x),a,x)) \end{array}\right) \end{array}\right) & \text{if cond1} \\ \Upsilon\left(\begin{array}{l} V_{pc}(\delta_{1,0,24}(t,a,x),a,x), \\ V_1(\delta_{1,1,24}(t,a,x),a,x), \\ gt\left(\begin{array}{l} \Pi^{tl}_{d_{1,2,24}}(V_1(\delta_{1,2,24}(t,a,x),a,x)), \\ \Pi^{tl}_{d_{1,3,24}}(V_1(\delta_{1,3,24}(t,a,x),a,x)) \end{array}\right) \end{array}\right) & \text{if cond2} \\ \vdots & \end{cases}$$

The structure of the function does not change, except the introduction of the tuple management operations Υ and Π, so the operation can create the new State Transition equations by recursing over the list of source State Transition equations. This is done using the *CreateSTs* operation:

$$CreateSTs : adSCAAlgebra \times Function^2 \rightarrow dSCASTVEqList$$

which is defined as:

$$CreateSTs\left(\begin{array}{l} Source\_SCA, \\ \Upsilon, \\ \Pi \end{array}\right) = BSTs\left(\begin{array}{l} GetEqSTVF(Source\_SCA), \\ [], \\ Createds(Source\_SCA), \\ Create\beta s(Source\_SCA), \\ Create\delta s(Source\_SCA), \\ GetMaxN(Source\_SCA), \\ \Upsilon, \\ \Pi \end{array}\right)$$

The *BSTs* operation is where the structure of the equation list is recursed:

$$\begin{array}{rl} BSTs: & dSCASTVEqList^2 \times ProjEqList \times \beta dSCAEqList \times \\ & \delta dSCAEqList \times N \times Function^2 \rightarrow dSCASTVEqList \end{array}$$

and it is defined recursively in two cases. The first case is where there exists a list of equations, and a recursive call is made to this operation with the list of new equations (neqs) being appended by

the result of a call to the BST operation:

$$BSTs\begin{pmatrix}(e,eqs),\\ neqs,\\ newds,\\ new\beta s,\\ new\delta s,\\ Max_N,\\ \Upsilon,\\ \Pi\end{pmatrix} = BSTs\begin{pmatrix} eqs,\\ \begin{pmatrix} BSTck\begin{pmatrix} e,\\ GetIndex(RetTerm(e,1)),\\ 0,\\ Max_N,\\ newds,\\ new\beta s,\\ new\delta s,\\ \Upsilon,\\ \Pi\end{pmatrix}, neqs\end{pmatrix},\\ newds,\\ new\beta s,\\ new\delta s,\\ Max_N,\\ \Upsilon,\\ \Pi\end{pmatrix}$$

The operation $BSTck$ is a simple checking operation to see if the module index is non zero. If this is true then a call to BST is made to construct a new dSCA State Transition equation. Alternatively, if this index is zero, then the module under consideration is the program counter module and a new definition should be created to reflect this (as the program counter definition will not change between modules, the shortcut of using the abstract dSCA definition in the concrete dSCA rather than creating a brand new definition is taken). $BSTck$ is given as:

$$\begin{aligned} BSTck: \quad & dSCASTVEquation \times N^3 \times ProjEqList \times \\ & \beta dSCAEqList \times \delta dSCAEqList \times Function^2 \rightarrow dSCASTVEquation \end{aligned}$$

with the following definition:

$$BSTck\begin{pmatrix} e,\\ mod\_num,\\ pc\_val,\\ Max_N,\\ newds,\\ new\beta s,\\ new\delta s,\\ \Upsilon,\\ \Pi\end{pmatrix} = \begin{cases} CreateVF\begin{pmatrix} mod\_num,\\ t+1, a, x,\\ BST\begin{pmatrix} RetTerm(e,2),\\ mod\_num,\\ pc\_val,\\ Max_N,\\ newds,\\ new\beta s,\\ new\delta s,\\ \Upsilon,\\ \Pi\end{pmatrix}\end{pmatrix} & \text{if } \text{cond}_1 \\ e & \text{o'wise}\end{cases}$$

where:

$$cond_1 = mod\_num \neq 0$$

BST will be a recursive definition over the structure of a State Transition equation's OpDef Term - recall that this will be of the form:

$$cond(V_{pc}(t,a,x) = 0, a, cond(V_{pc}(t,a,x) = 1, b, cond(V_{pc}(t,a,x) = 2, c, cond(\ldots))))$$

The three components (the conditional test, true path and false path) of each VFOpDef term will be separately "rewired" . The conditional tests component is always of the form:

$$V_{pc}(t,a,x) = pc\_val$$

and in the concrete dSCA definition it will be:

$$V_{new\_pc}\left(RetTerm\left(GetEl\left(\begin{array}{l}\delta s,\\ mod\_num,\\ 0,\\ RetTerm(e,2),\end{array}\right),2\right),a,x\right) = RetTerm(e,2)$$

where:

$$new\_pc = RetTerm\left(GetEl\left(\begin{array}{l}\beta s,\\ mod\_num,\\ 0,\\ RetTerm(e,2),\end{array}\right),2\right)$$

and:

$$e = (V_{pc}(t,a,x) = pc\_val)$$

The *pc_rewire* operation is introduced, which will create the new conditional component. For a complete definition a new equation with references to the extractions from correct wiring and delay functions should be produced, but in practice, the structure of the concrete dSCA does not differ from the abstract dSCA and the definition of *pc_rewire* can be simplified to just return the input. We give *pc_rewire* as:

$$pc\_rewire : STVEquation \rightarrow STVEquation$$

and provide the following definition for it:

$$pc\_rewire(e) = e$$

The true path component, i.e. the functionality that is used if the conditional component is true, needs to be manipulated to incorporate the tuple management functions, i.e given a component

specification:

$$cond(a, b, c)$$

then b would be transformed into:

$$\Upsilon \left(\begin{array}{l} V_{pc}(t, a, x), \\ V_{mod\_num}(t, a, x), \\ rewire(b) \end{array} \right)$$

To achieve this the *cs_rewire* operation is introduced:

$$\begin{aligned} cs\_rewire: \quad & Term \times N^3 \times ProjEqList \times \gamma dSCAEqList \times \\ & \beta dSCAEqList \times \delta dSCAEqList \times Function \rightarrow Term \end{aligned}$$

and it is defined as:

$$cs\_rewire \left(\begin{array}{l} trm, \\ mod\_num, \\ pc\_val, \\ Max_N, \\ ds, \\ \beta s, \\ \delta s, \\ \Upsilon, \\ \Pi \end{array} \right) = \Upsilon \left(\begin{array}{l} V_{pc}(t, a, x), \\ V_{mod\_num}(t, a, x), \\ rw \left(\begin{array}{l} trm, \\ mod\_num, \\ pc\_val, \\ Max_N, \\ ds, \\ \beta s, \\ \delta s, \\ \Pi \end{array} \right) \end{array} \right)$$

A generic rewire operation is not introduced, rather the definition for the number of arguments used in the machine algebra is given (there are zero to 3 arguments):

$$\begin{aligned} rw: \quad & Term \times N^3 \times ProjEqList \times \beta dSCAEqList \times Function \\ & \times \delta dSCAEqList \rightarrow Term \end{aligned}$$

and rw is defined as:

$$rw \left(\begin{array}{l} trm, \\ mod\_num, \\ pc\_val, \\ Max_N, \\ ds, \\ \beta s, \\ \delta s, \\ \Pi \end{array} \right) = trm$$

$$rw \left(\begin{array}{l} trm(trm_1), \\ mod\_num, \\ pc\_val, \\ Max_N, \\ ds, \\ \beta s, \\ \delta s, \\ \Pi \end{array} \right) = trm \left(wire \left(\begin{array}{l} trm_1, \\ \beta s, \\ \delta s, \\ ds, \\ mod\_val, \\ 2, \\ pc\_val, \\ Max_N, \\ \Pi \end{array} \right) \right)$$

$$rw\begin{pmatrix} trm(trm_1, trm_2), \\ mod\_num, \\ pc\_val, \\ Max_N, \\ ds, \\ \beta s, \\ \delta s, \\ \Pi \end{pmatrix} = trm\left(\begin{pmatrix} trm_1, \\ \beta s, \\ \delta s, \\ ds, \\ mod\_val, \\ 2, \\ pc\_val, \\ Max_N, \\ \Pi \end{pmatrix}, wire\begin{pmatrix} trm_2, \\ \beta s, \\ \delta s, \\ ds, \\ mod\_val, \\ 3, \\ pc\_val, \\ Max_N, \\ \Pi \end{pmatrix}\right)$$

$$rw\begin{pmatrix} trm(trm_1, trm_2, trm_3), \\ mod\_num, \\ pc\_val, \\ Max_N, \\ ds, \\ \beta s, \\ \delta s, \\ \Pi \end{pmatrix} = trm\left(\begin{pmatrix} wire\begin{pmatrix} trm_1, \\ \beta s, \delta s, ds, \\ mod\_val, 2, pc\_val, \\ Max_N, \Pi \end{pmatrix}, \\ wire\begin{pmatrix} trm_2, \\ \beta s, \delta s, ds, \\ mod\_val, 3, pc\_val, \\ Max_N, \Pi \end{pmatrix}, \\ wire\begin{pmatrix} trm_3, \\ \beta s, \delta s, ds, \\ mod\_val, 4, pc\_val, \\ Max_N, \Pi \end{pmatrix}, \end{pmatrix}$$

with the supplementary operation *wire* being given as:

$$\begin{aligned} wire: \quad & Term \times \beta dSCAEqList \times \delta dSCAEqList \times ProjEqList \times \\ & N^4 \times Function \rightarrow Term \end{aligned}$$

$Wire$ is defined for the three cases that may make up an atomic term within M_A:

$$\begin{aligned} wire\begin{pmatrix} const, \\ \beta s, \delta s, ds, \\ mod\_val, j, pc\_val, \\ Max_N, \Pi \end{pmatrix} &= const \\ wire\begin{pmatrix} a_p(t), \\ \beta s, \delta s, ds, \\ mod\_val, j, pc\_val, \\ Max_N, \Pi \end{pmatrix} &= a_{new\_index}(t) \\ wire\begin{pmatrix} V_p(t-1, a, x), \\ \beta s, \delta s, ds, \\ mod\_val, j, pc\_val, \\ Max_N, \Pi \end{pmatrix} &= \Pi_{prj\_val}^{Max_N - 1}(V_{new\_index}(new\_time, a, x)) \end{aligned}$$

where:

$$new\_index = RetTerm\left(GetEl\begin{pmatrix} \beta s, \\ mod\_val, \\ j, \\ pc\_val \end{pmatrix}, 2\right)$$

and:

$$new\_time = RetTerm\left(GetEl\left(\begin{array}{l}\delta s,\\ mod\_val,\\ j,\\ pc\_val\end{array}\right),2\right)+1$$

and finally:

$$prj\_val = RetTerm\left(GetEl\left(\begin{array}{l}ds,\\ mod\_val,\\ j,\\ pc\_val\end{array}\right),2\right)$$

Finally attention is turned to the false path of the term; this needs to be passed as an argument back to the BST operation. The definition of BST can therefore be given over the recursive structure of the State Transition equation, given as:

$$\begin{array}{ll} BST: & Term \times N^3 \times ProjEqList \times \beta dSCAEqList \times \delta dSCAEqList \times \\ & Function^2 \rightarrow Term \end{array}$$

and defined as:

$$BST\left(\begin{array}{l}cond(a,b,c),\\ mod\_num,\\ pc\_val,\\ Max_N,\\ newds,\\ new\beta s,\\ new\delta s,\\ \Upsilon,\\ \Pi\end{array}\right) = cond\left(\begin{array}{l} pc\_rewire\left(\begin{array}{l}a,\\ mod\_num, pc\_val,\\ new\beta s, new\delta s\end{array}\right),\\ cs\_rewire\left(\begin{array}{l}b,\\ mod\_num, pc\_val, Max_N,\\ newds, \beta s, \delta s, \Upsilon, \Pi\end{array}\right),\\ BST\left(\begin{array}{l}c,\\ mod\_num, pc\_val+1, Max_N,\\ newds, new\beta s, new\delta s, \Upsilon, \Pi\end{array}\right)\end{array}\right)$$

The recursive base case is defined as:

$$BST\left(\begin{array}{l}c,\\ mod\_num,\\ pc\_val,\\ Max_N,\\ newds,\\ new\beta s,\\ new\delta s,\\ \Upsilon,\\ \Pi\end{array}\right) = cs\_rewire\left(\begin{array}{l}c,\\ mod\_num,\\ pc\_val,\\ Max_N,\\ newds,\\ \beta s,\\ \delta s,\\ \Upsilon,\\ \Pi\end{array}\right)$$

The base case definition of $BSTs$, where only one State Value equation is in the list, is simply the result of making a call to the $BSTck$ operation and appending the result to the new equations

passed in as an argument:

$$BSTs\begin{pmatrix} e, \\ neqs, \\ deqs, \\ new\gamma s, \\ new\beta s, \\ new\delta s, \\ \Upsilon, \\ \Pi \end{pmatrix} = \begin{pmatrix} BSTck \begin{pmatrix} e, \\ GetIndex(RetTerm(e,1)), \\ 0, \\ Max_N, \\ newds, \\ new\beta s, \\ new\delta s, \\ \Upsilon, \\ \Pi \end{pmatrix}, neqs \end{pmatrix}$$

H.1.7 Transformation Process

Each of the operations above need to be coordinated together so that a new concrete dSCA can be created by transforming the source abstract dSCA. The $Create\_cdSCA$ operation is provided to do this, it is given as:

$$Transform : adSCAAlgebra \times Function^2 \rightarrow cdSCAAlgebra$$

The operation takes the source concrete dSCA and is defined as:

$$Transform\begin{pmatrix} SCA_{src}, \\ \Upsilon, \\ \Pi \end{pmatrix} = CreatecdSCA\begin{pmatrix} Name, \\ M_{Atup}\ , \\ [], \\ [], \\ VFOp, \\ \gamma_0 : N^2 \rightarrow \{M, S, U\}, \\ \beta_0 : N^2 \rightarrow N, \\ \delta Op\ , \\ Create\gamma s(SCA_{src})\ , \\ Create\beta s(SCA_{src})\ , \\ Create\delta s(SCA_{src})\ , \\ CreateIVs(SCA_{src})\ , \\ CreateSTs\begin{pmatrix} SCA_{src}, \\ \Upsilon, \\ \Pi \end{pmatrix} \end{pmatrix}$$

where:

$$VFOp = \begin{pmatrix} V_0 : T \times M_{Atup}^n \times M_{Atup}^k \rightarrow M_{Atup}, \\ \vdots, \\ V_k : T \times M_{Atup}^n \times M_{Atup}^k \rightarrow M_{Atup} \end{pmatrix}$$

$$\delta Op = \begin{pmatrix} \delta_{0,0,0} : T \times M_{Atup}^n \times M_{Atup}^k \rightarrow T, \\ \vdots, \\ \delta_{i,j,0} : T \times M_{Atup}^n \times M_{Atup}^k \rightarrow T \end{pmatrix}$$

and:

$$k = num\_mod(SCA_{src})$$
$$j = Get\_MaxA(SCA_{src})$$
$$n = num\_inp(SCA_{src})$$

It is not intended to bring together all the operations defined in this chapter into a written down specification in this thesis for reasons of brevity. If this was to be performed, then it would appear similar to the specification provided for the SCA to abstract dSCA transformation in Appendix F.