Automated Synthesis of Delay-Insensitive Circuits

Roger Anthony Sayle

Doctor of Philosophy
Department of Computer Science
University of Edinburgh
1996

Abstract

The technological trend towards VLSI circuits built from increasing numbers of transistors continues to challenge the ingenuity of both designers and engineers. The use of asynchronous design techniques presents a method for taming the complexity of large concurrent VLSI system design and offers a number of attractive advantages over conventional design styles. In this thesis, we concentrate on the useful class of *delay insensitive* asynchronous circuits. These have the property that their correct operation is independent of the speed of the individual elements and the delays in the connecting wires.

Traditionally, asynchronous circuits are considered much harder to design than their synchronous equivalents due to their inherent nondeterminism. The use of automated formal methods for generating such circuits shields the designer from this complexity. This allows the designer to abstract away from implementation issues and reason about the system behaviour in terms of concurrent processes or high level programs. Because each step of the compilation process can be shown to be sound, the resulting circuits are correct-by-construction.

This thesis presents a compilation methodology for designing delay insensitive VLSI systems from behavioural specifications. The synthesis method makes use of a graph-based representation of the circuit's behaviour. Optimization of the global behaviour, by graph transformation, enables the generation of more efficient circuits than those produced by previous asynchronous circuit compilers based on syntax-directed translation. The resulting circuits are further improved by semantics-preserving circuit transformations. A compiler has been constructed that automatically performs the translation and transformation.

Acknowledgements

Although this page is considered unimportant and is often skipped by the reader, to the author it is perhaps the most important page of all. The work in this thesis would not have taken its final form without the help and friendship of the people mentioned below.

Foremost I would like to thank my thesis supervisor Mike Fourman, for his guidance and for the exceptional creative freedom I have been allowed. I would also like to thank Jo Ebergen, Mark Josephs and Jan Tijmen Udding for many constructive conversations. But most of all I would like to thank my examiners, Julian Bradfield and Graham Birtwistle for their many recommendations that have resulted in a much improved thesis.

Thanks also go to Delia Johnson for her continual support and encouragement.

This work has been financially supported by funding from the Science and Engineering Research Council.

Table of Contents

Chapter 1

Introduction

The complexity of VLSI circuits makes it difficult for designers to ensure their correct operation. This problem is aggravated by the continual reduction of transistor size and growth of circuit size. The use of asynchronous circuits offers a possible solution to the first issue and the use of automated formal methods as a solution to the second. Automated asynchronous circuit synthesis can reduce both the time and expense of producing correct and reliable circuits. In this thesis, we present such a design method for a useful class of asynchronous circuits.

1.1 Thesis Overview

This thesis is organized such that the reader who is not interested in the details of the theorems can skip them and still understand the important points. Each chapter includes a set of examples to illustrate the concepts described in that chapter. A glossary containing definitions of the more commonly used technical terms is presented at the end of the thesis. The rest of this first chapter describes what are asynchronous circuits, and particularly delay insensitive circuits are. This is followed by a section motivating their use and finally by a review of previous work on asynchronous circuit design.

Chapter 2 begins by introducing an abstract model of asynchronous circuit. This model describes the observable operational model of an asynchronous circuit in terms of labelled transition systems (LTSs). This model is then related to other formalisms for describing asynchronous circuit behaviour, including process algebra, trace theory and Petri nets. An improved implementation (or satisfaction)

relation is then defined on these transition systems that holds between a specification and a valid implementation of that specification. Automated techniques are then discussed for translating specifications into LTSs, checking (and correcting) their delay insensitivity and finally converting it into a minimized normal form.

Chapter 3 describes the basic target components used by the synthesis method. The chapter starts by introducing a number of 'primitives' that form a component basis for the synthesis method. Using these primitives, more useful components (commonly used by other researchers) are defined. The chapter goes on to define larger parameterizable 'macro' components that are used by the circuit compilation method. The correctness requirements for transistor level implementations of these components are then discussed. The constraints insure that composition of the physical transistor level implementations obey the abstract model. The chapter ends with a section on initialization conditions and implementation issues for transistor (or gate) level implementation of DI modules.

Chapter 4 presents an introduction and overview of the automated delay insensitive circuit synthesis methodology. This chapter introduces the 'stable state' model of delay insensitive circuit behaviour. This model then provides the formal basis of an useful abstraction of delay insensitive transition systems, called stable state graphs (SSGs). This representation takes advantage of the properties of delay insensitive behaviours to reduce the complexity of representing circuit behaviour, and acts as an underlying formalism for the circuit synthesis method. A classification of stable states and several other properties of SSGs are then defined that are used during circuit synthesis. Next, an improved generic implementation strategy is described that forms the basis for the proposed compilation method. Finally, the chapter ends with a short overview of the steps involved in the proposed compilation (decomposition) method and the target implementation model for this approach.

Chapter 5 contains the detailed description of the proposed synthesis methodology. The chapter is arranged sequentially, presenting decomposition strategies for larger and larger classes of circuit behaviours. This organization allows the exposition of simple concepts and examples first, before proceeding to more complex cases and problems. Chapter 6 describes a number of improvements to the synthesis method given in the previous section. These optimizations and transformations are discussed separately from the main method for clarity. The various improvements presented in this chapter are divided into two groups; circuit trans-

formations and behavioural transformations. Circuit transformations are 'peephole' optimizations that may be applied to the generated circuit to improve the resultant design. Behavioural transformations are semantics preserving transformations that modify the circuit specification as a preprocessing step (or first pass) to the automated compilation method.

Chapter 7 describes the application of the proposed circuit synthesis method to several example circuit specifications. The case studies have been defined by other researchers, and provide a set of common benchmarks in the field of delay insensitive circuit design. Finally, Chapter 8 contains a summary of the thesis. It reviews the main achievements of this research, and proposes some suggestions for future work in this area. This chapter also compares the developed delay insensitive circuit synthesis methodology to the results of similar related work by other researchers.

1.2 Asynchronous Circuits

In classical logic design, digital circuits are classified as being either *combinational* or *sequential*, distinguished by the existence of feedback signals within the circuit. Combinational circuits are those without feedback and are therefore stateless. Each of the circuit's outputs is a (delayed) boolean function of the circuit's inputs. To do more complex computations, state holding sequential circuits must be used. Timing is a central problem in such circuits, especially those using feedback signals, as the design must ensure that a computation does not begin until earlier computations have been completed.

In conventional *synchronous* design, storage elements are used to decouple the feedback signals and break the circuit into combinational logic separated by latches. These latches capture their inputs on the tick of a global synchronization 'clock' signal. The correct operation of the circuit is ensured by spacing the clock ticks by more than the delay of each combinational block. Synchronous design typically also assumes that the inputs to the circuit change on or shortly after each clock tick, and then remain constant until the next tick.

The alternative to this almost universal approach is to use unclocked *asynchronous* design styles. Asynchronous circuits are divided into those that avoid timing problems by carefully co-ordinating the various delays in the circuit and *self-timed*

circuits that use synchronization signals between circuits. A classification of these various timing paradigms is given in figure 1–1.

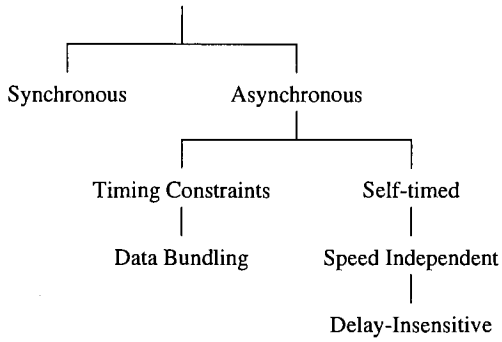


Figure 1–1: Classification of circuit timing paradigms.

The most common form of timing constraint used in asynchronous design is the *data bundle.* A 'bundle' of data wires, or bus, is associated with a single set of control wires that show the validity of the value on the data bundle. The bundling constraint requires that the data value be observed at the receiver before a signal appears on a control wire. This may be achieved by constructing delays in the control wires. Bundling allows data values to be transmitted over buses without the wiring and circuitry overhead of using complex encoding techniques. Bundled data paths are commonly used with self-timed control circuitry. These circuits can make use of standard combinational logic components, such as adders and multipliers. Other forms of timing constraint used in asynchronous design include assumptions about the upper and lower bounds on component delays and assumptions about the maximum ratios of these bounds.

Interest in self-timed circuits has been centered on *speed independent* circuits, those that do not depend on the delays of the individual components within the circuit. Speed independent circuits form pure concurrent systems and are therefore particularly amenable to mathematical methods. These circuits consist of a fixed set of processes (modules) communicating over a fixed set of unbuffered channels (wires). All synchronization between modules is by explicit handshaking protocols over these channels. The complexity caused by the nondeterministic behaviour of such concurrent systems is the principal reason for the popularity of synchronous design styles.

In this thesis we concentrate on *delay insensitive* circuits, a subclass of speed independent circuits. A digital circuit is delay insensitive if its correct operation is

independent of the delays in both the individual components and the interconnecting wires of the circuit, except that those delays be finite. This tolerance to wire propagation delays eliminates several of the technical problems currently facing circuit designers.

Digital circuits cannot always be classified as either totally synchronous or asynchronous. A large system is often designed at several different levels of abstraction. At each level, the use of a different timing paradigm may be appropriate. For example, a single global clock signal is better suited to gate level abstractions than to large computing systems, whereas delay insensitive techniques are inapplicable to transistor level circuit abstractions. An example of a multi-paradigm design style is the use of several independent clocks within a single design, called *self-clocked* design. Conventional synchronous methods are used within each clock domain, and these domains communicate between one another asynchronously. Similarly, an asynchronous circuit may be used within a conventional synchronous (clocked) design by either estimating or measuring the upper bound on the computation delay.

1.3 Motivation

As the technological trend to scale down the feature size of transistors and increase the size of each chip continues, conventional synchronous circuit design encounters several serious, fundamental limitations. Some of these problems relate to the difficulty in synchronization and transfer of information between subcircuits within a single clock period, while others are caused by the complexity of managing large designs [84].

Seitz [106] argues that as the physical size of the circuit is reduced, the parameters determining its behaviour do not scale uniformly. The relationship between wire delay and switching delay changes such that the delays in the connecting wires increasingly dominate the transistor switching times. As the cost of communications becomes ever more important, a change of design discipline is required. The serious problem of *clock skew* which is facing current designers is caused by the difficulty of distributing global clock signals. Transmission delays cause clock ticks that are supposed to be simultaneous, to occur at significantly different times.

The ability to isolate and abstract the implementation timing from the logical design of a circuit is a useful property of a hardware design method. It allows both the behaviour and the correctness of a design to be reasoned about, without reference to the physical properties of a specific implementation. Delay insensitive circuits allow this separation between design and engineering issues and are therefore attractive for analysis by formal methods and particularly automated synthesis.

Delay insensitive circuits also have several advantages over conventional synchronous design that are attractive to engineers and designers.

Performance Asynchronous circuits are potentially faster than their equivalent clocked designs. A synchronous system is composed of several parts that carry out specific computations. The correct behaviour of the system requires the clock rate to be slowed to the speed of the slowest subcomputation. This clock period must also take into account the worst case execution time and propagation delays under worst case operating conditions. In, comparison, Self-timed circuits operate at a rate determined locally by component and wire delays, and this tends to reflect average case rather than worst case delay for the circuit.

Power Consumption Asynchronous circuits typically have a much lower power consumption for two principal reasons. Firstly, no power is dissipated by driving a global clock signal at high frequencies. Secondly, transistors in an asynchronous circuit only fire when they contribute to a computation. A related engineering benefit is that power dissipation is distributed over time rather than concentrated at rising and falling clock edges. This property leads to a reduction in electromagnetic interference.

Modularity Self-timed circuits have useful composition properties that simplify the process of system design. Large complex systems can be split into independent modules without interface timing constraints. Self-timing also allows efficient circuits to be collected to form libraries and re-used within designs. This modularity also allows any component within a design to be replaced safely by a functionally equivalent one, with improved properties such as performance or cost, without compromising the circuit's correct behaviour.

Robustness Delay insensitive circuit behaviour is less sensitive to variations in the environment. Physical factors such as temperature, process spread and power supply variations tend to affect the relative delays in the circuit rather than its logical correctness. Reducing the temperature of a delay insensitive circuit increases the observable speed of the circuit. Asynchronous circuits also suffer less from other effects, such as switching noise. Tolerance of these parameters is typically difficult to assess before fabrication and to detect during testing.

Implementation The low level physical design of delay insensitive circuits is also much simplified. Routing constraints on the physical layout of the circuit and geometry of the circuit are relaxed. Transistor sizing and placement optimization can be performed, provided the functionality of the primitive components is maintained. Reimplementation of the design at a different feature size or in a different fabrication process requires only redesign of the primitive components. This allows designs to be reused and to take advantage of improvements in implementation technology with minimum effort.

Metastability Another benefit is the reliability of circuits containing components with metastable behaviour. Metastability is an unstable equilibrium in the state of an electrical circuit. A metastable circuit can remain in a metastable equilibrium for an indefinite period of time before resolving into a stable state. This can cause functional errors in synchronous circuits when the duration of the metastability lasts longer than a single clock period. This 'glitch' phenomenon, first discovered by Chaney and Molnar [25], is fundamental to arbiter and synchronizer circuits. Indeed, Mendler and Stroup [85] show that any such device built on Newtonian principles (that voltage changes are continuous) will behave incorrectly with the appropriate inputs. Self-timed circuits avoid this problem as the circuit waits until the element has settled into a stable state before the computation proceeds.

Testability Delay insensitive circuits are completely self-checking for single and multiple *stuck-at-faults*. The occurrence of a break in any of the circuit's wires will eventually halt the operation of the circuit. This property means that deterministic delay insensitive circuits are fully testable for stuck-at-faults. Any test computation that fails to complete within a maximum permitted time limit indicates a fault within the circuit. It is also possible to build fault tolerant systems by using a

'watchdog' time-out signal, where the absence of a result from a subcircuit within a time limit causes the system to detect that the subcircuit is malfunctioning.

1.4 Background and Related Work

Previous work on asynchronous circuit design methodologies can be categorized by the 'delay-model' used in the circuit abstraction. The organization is chronological from the date of the model's introduction. The first work on asynchronous circuit design assumed an upper limit on the delays in both the components and the wires of the circuit. This was followed by speed independent circuit design, using the assumption that component delays were unbounded and that wire delays are negligible. This is equivalent to stating that all wire delays are concentrated at the outputs of the components. The most recent delay model is that used for delay insensitive circuit design, which allows unbounded delays in both components and wires. As with most classifications, some works may be placed in more than a single class or group.

Similar reviews of the field of asynchronous circuit design have been published by several authors [4,50,51].

1.4.1 Bounded Delay Model

Asynchronous State Machines

Early work on asynchronous circuit design concentrated on the design of asynchronous state machines, referred to as *Huffman circuits* [56,57]. These state machines are implemented as combinational logic with delayed feedback signals to maintain the current state. The circuit is assumed to operate in *fundamental mode* [83] where the interval between successive input changes is long enough for the circuit to complete its transition to the next stable state.

The principal difficulty in asynchronous state machine synthesis is the state assignment problem. For a synchronous finite state machine (FSM), $\lceil \log_2 n \rceil$ state variables are both necessary and sufficient for representing n states. As long as each state code is unique, the assignment of codes to states may be made arbitrarily without affecting the correct operation of the FSM. However, this is not the case

for asynchronous state machines where a 'critical-race-free' state assignment is required, which may need more than $\lceil \log_2 n \rceil$ variables. A *race* is said to occur when two or more state variables change during a state transition. This race is termed *critical* if the final state reached depends on the ordering of the changes.

Algorithms for these critical-race-free state assignments have been suggested by several authors [43,73,111] and techniques permitting non-fundamental mode operation have also been suggested [116]. Detailed reviews of early asynchronous state machine synthesis may be found in the text books by Unger [115] and Friedman and Menon [44]. Enhancements to the asynchronous state machine model have been developed by Hayes [52], Hollaar [55] and Molnar [91]. More recently, asynchronous state machine synthesis based on *burst mode* operation has been suggested [31]. This alternative to fundamental mode operation allows construction of fast and efficient control circuits. The principal disadvantage of the state machine model is that their behaviour is strictly sequential and concurrent operation cannot be described.

Self-clocked Design

Seitz, in his chapter 'System Timing' in Mead and Conway's text book [106], proposes *self-clocked* design as one approach to asynchronous circuit design. Small self-timed elements are designed as synchronous systems with internal clocks, in *equipotential* regions inside which wire delays are considered negligible. The design may be made free of synchronization failure by using 'universal clocks' that may be stopped and restarted asynchronously. This retains the advantages of synchronous design under those conditions in which it is workable. Rosenberger [101] describes a similar design style called *Q-Modules*, where local clock ticks are suspended until all storage elements have settled into the next state.

Micropipelines

In his 1988 Turing Award Lecture, Sutherland [110] described another asynchronous design style based on the bounded gate-delay model. This style, termed *micropipelines*, uses fine grain pipelines where each stage communicates asynchronously with its neighbours. In addition to a single completion signal, these stages have another signal to acknowledge the receipt of the start request and permit another request to be sent. In conventional self-timed design, acknowledgement

takes place only after the processing has been completed. The micropipelined paradigm can achieve high speed throughput by overlapping operations, with different stages operating concurrently. Sutherland's approach traditionally uses a four phase bundled datapath with two phase handshaking. Furber and his colleagues at the University of Manchester have used this approach to implement a self-timed RISC processor [45,47].

1.4.2 Unbounded Gate-Delay Model

Speed Independence

Although most early interest was in this state machine model of asynchronous circuits, the roots of the theory of self-timed circuits go back as far as the early 1950s. Muller and his colleagues at the University of Illinois developed a fundamental algebraic theory of circuits whose observable behaviour does not depend on the relative speeds of their elements [92]. It was in this seminal paper that the term *speed independent circuit* was first coined. However the relevance of this work, and a similar later paper by Armstrong, Friedman and Menon [2], was not recognized until much later. It was in this second publication that the self-checking properties of self-timed circuits were first mentioned.

Independent research in the Soviet Union during the 1950s and 1960s tackled the problem of the variable delays in standard SSI and MSI IC packages. The best case gate delays of identical commercially available circuits were four or five times better then their worst case delay. A theory of speed independent circuits, termed *aperiodic* circuits in translation, was developed by Varshavsky and Rosenblum [126] based on earlier work on asynchronous circuits by Gavrilov [48]. Varshavsky *et al.* [125] showed that any four-phase asynchronous state machine can be implemented speed independently from AND-OR-NOT components. Originally, implementations required these 'antitone' components of arbitrary complexity. However subsequent work describes implementations in a finite basis of primitive components [124]. Synthesis from event-based specifications, rather than state transition diagrams, was first suggested by Starodubtsev [107]. His doctoral thesis [108] contains efficient synthesis and analysis methods for arbitrary autonomous circuits described by *taxograms*, where events are labelled by signal transitions.

Petri Net Modules

A module based approach to the implementation of speed independent circuits was investigated during the 1970s by Dennis' group of project MAC at the Massachusetts Institute of Technology [32]. They used Petri nets to model the behaviour of speed independent circuits. The *Petri net* is a widely used model of concurrent systems developed by Carl Petri. A Petri net is a bipartite directed graph with two types of nodes: places and transitions. Places within the net are capable of containing some number of 'tokens' or 'markers', such that a condition (place) is said to hold or be true if a token is present in it. Places are connected by directed arcs, via transitions, along which tokens may pass. The movement of tokens along the arcs is controlled by the occurrence of a 'firing' of a transition, known as an *event*. A transition may only fire when there is a token in each input state, when a single token is removed from each of its input states and placed in each of its output states.

Patil [96] proposed implementing speed independent circuits that resembled the Petri net itself. The Petri net was structurally realized by modules that performed the roles of the places, transitions and arcs of the Petri net. The main drawback with this approach is that it resulted in very large implementations for relatively simple circuit behaviours. Misunas [90] improved upon this idea by using modules of common Petri net functions. These basic functions are then composed to implement the behaviour of the specification Petri net. Using this relatively simple approach Misunas was able to design a speed independent processor based on the design of a CDC 6600. More recently, Lister and Alhelwani have proposed a similar method for implementing speed independent circuits from data-flow Petri nets [72].

Signal Transition Graphs

The *signal transition graph* (STG), originally called a 'signal graph', was first introduced as a formal model of asynchronous circuit behaviour by Rosenblum and Yakovlev [102,103,128]. Similar work on STGs has been independently introduced by Chu [26] and Molnar *et al.* [91] under the name 'I-Nets'. An STG is an 'interpreted Petri net' where the transitions are interpreted as signal transitions. STGs formally model the causal relationships between the circuit and the environment in which it operates.

The synthesis of asynchronous circuits from STGs involves finding both a state assignment and a hazard-free implementation of the circuit using the appropriate delay-model. The problem of state assignment, assigning a binary coding to each state, is similar to that in asynchronous state machines. An STG has a *unique state coding* (USC) property if the binary code assigned to each state is distinct from the codes assigned to every other state. An STG has the weaker *complete state coding* (CSC) property, if any two markings that enable different sets of output signals have distinct state codes. A CSC state assignment often permits a more efficient implementation that a USC state coding, as distinct states may have the same coding allowing states to be represented by shorter binary codes. Different synthesis techniques require the STG to satisfy either the USC or CSC property.

Algorithms for modifying the STG by adding state variables to obtain both the above properties have been given by several authors [71,86,123,129]. The chosen state assignment must also have a boolean implementation that is free from both critical races and hazards. A *hazard* in an asynchronous circuit is an unexpected 'glitch' of a signal value that is a transition that is not allowed by the specification. Hazards are typically caused either by the delays in the circuit's feedback signals or the differences in the gate delays of the circuit. The use of STGs with bounded-wire delay models has been investigated by Lavagno *et al.* [68,69].

CalTech Design Method

Alain Martin [76,78,80] has developed a design method for speed independent circuits that produces very efficient implementations. His method is based on a sequence of transformations from a specification of communicating processes to a transistor level implementation. The final implementations contain isochronic forks but are independent of transistor switching delays. A fork or branch of a wire is considered *isochronic* if the difference in propagation delays between the two branches is negligible. This assumes all wire delays are concentrated at gate outputs, and hence circuits are speed independent but not delay insensitive.

Martin specifies the desired behaviour of the circuit using a language based on Hoare's CSP [53] and Dijkstra's guarded command language [33], called Communicating Hardware Processes (CHP). In addition, CHP contains assignments, arrays, functions, procedures and a 'probe' construct that allows a process to determine if there are any incoming communications pending [75]. The synthesis

method begins by decomposing the processes into small, sequential processes with explicit handshaking between them. All communication is assumed to be four-phase using dual-rail encoding. After a state assignment similar to those described above, each process is translated into a set of *production rules*, which form a compact representation for CMOS transistor networks. Production rules are selected that ensure the correct sequential behaviour of the circuit and minimize the number of state holding operators required. Burns and Martin have also developed an algorithm for automatic transistor sizing for the generated circuits [22]. This technique has been used on a number of complex designs including a fast asynchronous microprocessor [77]. Methods based upon production rules have also been investigated by several other authors [109,117].

1.4.3 Unbounded Wire-Delay Model

Macromodules

The first attempt to construct circuits that were independent of wiring delays was made by the Macromodules project at Washington University around 1970 [27]. It was this research that first coined the term 'delay insensitive circuit'. The aims of the project were to investigate computer architecture using modular building blocks that performed specific operations. Physical implementation issues demanded that regular interfaces be used to plug modules together and their operation be independent of wiring delays between the racks that held the modules. Around 16 types of module were implemented including modules for ALUs, registers, adders and ferrite core memory. Communication between modules used transition signalling for the control paths and bundled data paths. This design approach led to ease of reconfiguration and no natural limit to the size of the design.

Following the work of the Macromodules project, Keller [64] attempted to determine a minimum set of primitive delay insensitive modules that were 'universal' to the class of speed independent or delay insensitive circuits. That is, all such circuits can be implemented entirely from a restricted set of basic elements. This is similar to the property that any synchronous design may be implemented using just NAND or NOR gates. Keller showed that only three components (the merge, the select and the arbitrating test and set) are required for all four-phase speed independent circuits. However, no attempt was made to formalize either the composition semantics or the class of circuits covered by the method.

Recent results by Seger [105], Brzozowski and Ebergen [16,17], Martin [79] and Leung and Li [70] have shown that the class of purely delay insensitive circuits is extremely restricted. This proof, called the Unique Successor Set (USS) property, states that circuits of components with only single outputs are either sensitive to wire delays or perform trivial computations. This result requires primitive component bases for general delay insensitive circuits to contain elements with multiple outputs. Implementations of these elements using transistors (or conventional logic gates) must make use of timing constraints, such as isochronic forks.

Trace Theory and Process Algebra

The first formal model of delay insensitive circuits was developed at the Technical University of Eindhoven [99,121]. *Trace theory*, first proposed by Hoare [54], represents the behaviour of an asynchronous circuit as the sequence of voltage transitions at its interface. Directed trace structures classify each observable wire as either an input or an output, and specify all permissible orderings of events on these wires. These orderings, or *traces*, specify the correct operation of a circuit and also constrain the circuit's environment. A circuit is delay insensitive if its wires are free from *computation interference* and *transmission interference.* Computation interference occurs when a signal arrives before a receiver is ready for it and transmission interference occurs when two signals on the same wire interfere.

Snepscheut [122] defines a composition operator for directed trace structures, called *agglutination*, that places an explicit delay in wires connecting outputs to inputs. Because this operator allows an unbounded number of messages per wire, explicit handshaking signals must be added to avoid transmission interference. Snepscheut implements trace structures as a network of communicating asynchronous state machines whose interconnections are independent of wiring delays. Each state machine is constructed from a programmable logic array (PLA), flip-flops and c-elements and is assumed small enough to be embedded in an isochronic region.

Udding [112,113] presented a formal definition of the delay insensitivity of a component as a set of constraints on the component's trace structure. These rules also lead to system of classification of delay insensitive circuits. Udding also states necessary conditions for a composition of trace structures to be free from computation and transmission interference and proves that the class of delay insensitive circuits is closed under composition.

In his doctoral thesis Ebergen [36,38] presents an alternative formalization of delay insensitivity which he proves equivalent to Udding's rules. Ebergen also introduces a more general (de)composition operator for delay insensitive networks that reflects the permissible behaviour of the circuit's environment. Using this definition, an automatic method for synthesis for delay insensitive circuits is presented for a restricted class of circuit specifications. This syntax directed approach produces a delay insensitive network of primitive components from a predefined (but infinite) basis.

Dill [34] suggests the use of trace theory for the automatic hierarchical verification of speed independent circuits. His LISP model checker uses a state based approach to verify 'safety' properties, such as the delay insensitivity of a circuit. He also defines an implementation equivalence between two trace structures that holds when one may be safely replaced by the other.

The prefix-closed trace structures described fail to capture properties such as the 'liveness' and 'fairness' of a circuit [46]. Black [8] extends trace theory to include infinite traces which may be used to describe the behaviour of fair arbiters. A similar approach using non-prefix-closed sets of finite and infinite traces, termed *complete trace structures*, has been described by Dill to cover both liveness and fairness [34].

An alternative to the use of trace theory is to describe the behaviour of an asynchronous circuit as a set of communicating sequential processes using a process algebra such as CSP [53] or CCS [89]. This model of computation considers a static set of concurrent processes that interact via input and output commands on shared communication channels. Josephs and Udding [58,59] have developed a delay insensitive process algebra, based on CSP, that expresses the properties of delay insensitive circuits. The formalism contains algebraic laws for modelling interference that can be shown equivalent to Udding's original rules. The laws of the DI algebra allow expressions to be transformed into a normal form [49] and semantically equivalent forms. This system of manual transformations provides a basis for algebraically specifying and verifying component behaviours but is unsuitable for an automatic synthesis methodology.

1.4.4 Module Based Synthesis

Several systems have been developed to automate the synthesis of delay insensitive circuits from behavioural specifications. These circuit compilers translate a process based description of the desired behaviour into a network of primitive elements. A process based specification is either the composition of simpler processes or a primitive process. Typically a source language will contain constructs for sequential, parallel and conditional composition, alternation and repetition. All of the compilers described below use syntax directed translation techniques [1], where each syntactic construct of the source language generates a fixed set of components and wires. Using this method, an implementation is given for each of the primitive processes of the language and a circuit composition rule is given for each construct of the specification language. For a compound specification of a process and valid implementations of its subprocesses, these rules should generate a valid implementation of the whole construct.

A restrictive communications protocol is imposed on each process to ensure the semantics of circuit composition is equivalent that of the specification language. Typically each process is viewed as a module that begins execution upon the receipt of a request signal and indicates completion by an acknowledge signal. These processes may then be connected/composed to create larger circuits with similar request and acknowledge signals. Synchronization and communication between concurrent processes is by implicit handshaking circuitry. In this way, the source specification may be used to describe the hierarchical structure of the implementation. This simplistic translation may produce relatively large and inefficient circuits.

Burns

The first automated asynchronous circuit compiler was developed by Burns and Martin [21]. This program automated the CalTech design methodology described earlier. The translation is from a variant of CSP that includes the *probe* construct [75]. The addition of the probe allows CSP to specify 'fair' arbiters. Several possible rules are given for each construct allowing trade-offs between cost and performance. For example the guards of the ALT construct may be tested either sequentially or concurrently (possibly using arbiters to ensure mutual exclusion). The compiler also performs a significant amount of optimization at both the source and circuit level. Source level optimizations involve testing invariants that guide

the selection of transformation rules. Circuit level, or *peep-hole*, optimizations remove locally redundant components from the final circuit. The resulting circuits use four phase handshaking and consist of six basic components. These are the standard logic gates, AND, OR and NOT, and three state holding components; the Muller c-element, the synchronizer and the set/reset flip-flop.

Because this design style requires isochronic forks, the generated circuits are speed independent but not delay insensitive. A full description of the translation rules and optimizations is given in Burns' masters thesis [20]. The complete system consists of about 800 PROLOG clauses. The compiled circuits are typically no more than three times the size of those derived by hand. This inefficiency is caused by the compiler not detecting all cases where explicit sequencing can be removed.

Brunvand

A similar approach has been used by Brunvand and Sproull [14]. Their compilation uses similar techniques to generate circuits that consist of delay insensitive control units that direct the flow of data of bundled data paths. All control signals within the target design use transition signalling. The source language to the compilation system is a large subset of the Occam programming language [81] given in a LISP-like syntax. The synthesis system also includes an Occam interpreter that allows behavioural simulation and debugging of specifications prior to their compilation [11]. The details of the nine translation rules are given in Brunvand's doctoral thesis [13].

Except for the data path operators, the synthesis method requires only 11 primitive modules. The complete list of primitive components includes the merge element, the c-element, the toggle element, the arbiter and several kinds of storage element including the select and call modules. All of the required components have been implemented as standard cells requiring only a small number of transistors [12]. The state holding components respond to a global 'clear' signal to reset them to an initial known state. The major advantage of data bundling is that standard library components are used to implement the data path, such as adders and multipliers.

Brunvand's compiler, like Burns', improves the generated designs using semantics preserving circuit-to-circuit transformations, similar to the peep-hole optimizations used in conventional compilers. This pass substitutes inefficient subcircuits by more efficient implementations by identifying common module topologies.

These local improvements overcome some of the disadvantages of the relatively simplistic translation method. Brunvand proves the validity of the circuit transformations by showing their conformance equivalence using Dill's trace tools [34].

Brown

The first attempt at produce a truly delay insensitive circuit compiler was made by Brown [10]. His system used a variable free subset of the Occam programming language as the specification language. The absence of variables removes the assignment, conditional and data communication commands from the language. The design of circuits without data paths avoids the need for either data bundling constraints or isochronic forks. The compilation process requires only ten rules to generate delay insensitive circuits from their specifications. The target circuits use transition signalling and are composed of six types of primitive component. These components are the merge, the c-element, the select, two types of call module and an arbitrating test-and-set module.

van Berkel

A far more advanced delay insensitive circuit compiler has been developed by van Berkel and the group at Philips Research Labs [119,117]. The source language to their synthesis system, called *Tangram*, is based heavily upon Hoare's CSP. In addition to the constructs of Occam, Tangram supports finite iteration, guarded commands, arrays, tuples, arithmetic operators, broadcast communication, functions and procedures. Recently, the syntax of the Tangram programming language has developed a Pascal-like syntax to make the notation more accessible to designers.

The first phase of the Tangram compilation process is the syntax directed translation of the source description into a graphical intermediate representation, called *handshake circuits*. A handshake circuit is a graph representing both the structure and behaviour of the circuit. Each vertex of the graph is a handshake component and control and data flow is explicitly represented by arcs between nodes. These delay insensitive channels form the only interaction between handshake components. This representation abstracts the communication actions away from specific handshaking protocols and data encodings. The full implementation of Tangram requires about two dozen types of handshake component, many of which are parameterized on word width. The Tangram compiler includes a peephole optimization

pass that identifies subcircuits of the handshake circuit and replaces them with cheaper equivalents.

The final stage of the compilation process implements the handshake circuit as a CMOS VLSI circuit. Currently, the Tangram compiler synthesizes circuits with four phase handshaking and dual rail encoding, and the corresponding speed independent gate-level decompositions of the handshake components. The realizations of handshake components allow all components in the circuit to be initialized by forcing the electrical inputs low. This property called *weakly initializable* avoids the need for reset circuitry. More recently a testability option has been added to the compiler, which generates test circuitry for a restricted class of circuits [100]. This test strategy allows circuits to be tested effectively in polynomial time.

The Tangram compiler is part of an integrated tool set. The Tangram source program can be compiled into a C-code simulator that will generate a coarse timed trace of the circuit's behaviour or the intermediate handshake circuit may be translated to VHDL to generate more detailed timing information. The 'backend' of the compiler produces standard cell net lists that may be input to Philips' VLSI layout tools. At a more abstract level, the VOC project at Eindhoven University of Technology have produced a front end to the system that generates regular language acceptors [7].

Chapter 2

Behavioural Specification

The design process can be seen as a translation of notation, starting with an abstract descriptive specification and ending with a more concrete realization or implementation of that specification. A synthesis method is a theory for constructing realizations that are guaranteed to be correct with respect to their specifications and therefore do not require debugging or verification. The foundations of any synthesis method must contain a model for reasoning about a system's behaviour. Several different models have been suggested for describing the concurrent behaviour of asynchronous circuits, including process algebra [23], Petri nets [26], temporal logic [74] and trace theory [122]. In this thesis, a mathematical model based on labelled transition systems is proposed to describe the behaviour of delay insensitive circuits.

2.1 Abstract Circuit Model

2.1.1 Labelled Transition Systems

Labelled transition systems use an *interleaving* model of concurrency where simultaneous actions can occur in any temporal ordering, thus reducing concurrency to non-deterministic interleaving. Models based upon *true concurrency*, such as Petri nets, describe concurrent operation by explicit causal independence. Although truly concurrent formalisms have greater discriminating power than interleaving models, the 'observable' behaviour of an asynchronous system is a chronological sequence of events. Hence, an interleaving model is a suitable formalism for behavioural specification, which need not distinguish between underlying causal models.

It is the design methodology's task to find an implementation with a suitable causal mechanism.

Definition 1 *A directed labelled transition system is a structure* (S, s_0, I, O, T) *where* S *is a finite non-empty set of* states, $s_0 \in S$ *is an* initial state, I *and* O *are disjoint sets of input and output* labels *respectively and* $T \subseteq S \times (I \cup O) \times S$ *is the* transition relation.

Usually we write $s \xrightarrow{a} s'$ instead of $(s, a, s') \in T$, and $s \rightarrow s'$ when $\exists a \quad s \xrightarrow{a} s'$. The set of labels $I \cup O$ is sometimes called the alphabet of the directed labelled transition system (DLTS). Labels from the set of input labels, I, are distinguished by suffixing with a question mark and output labels, O, by suffixing with an exclamation mark. For a transition $t = (s, a, s')$, s is called the prestate of t, and s' is called the poststate.

At any moment in time, the asynchronous process described by a directed labelled transition system can be in one, and only one, state or situation. Time is introduced into our descriptive model by the transition relation T which describes the sequential evolution of the asynchronous process. The definition of T as a relation rather than a function expresses the non-determinism inherent in a process' specification. Typically the transition relation is non-reflexive.

This definition of a directed labelled transition system leads to the usual graphical representation as a *directed transition graph.* The vertices of the transition graph are used to denote the states of the transition system, and labelled arcs represent the transition relations between them. Usually, the vertex of the transition graph that denotes the initial state, s_0, is emphasized to distinguish it from the remaining states. An example of a transition graph is given in figure 2–1.

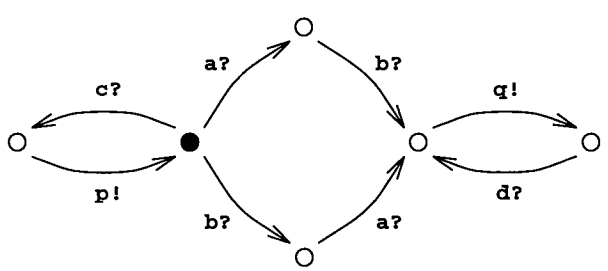


Figure 2–1: Graphical representation of a directed labelled transition system.

The common notion of *bisimulation* [88,95] can be used to define an equality relation on labelled transition systems. Two labelled transition systems are equivalent if their initial states are bisimilar.

Definition 2 *The two states P and Q, of labelled transition systems $\mathcal{P}$ and $\mathcal{Q}$ respectively, are bisimilar, written $P \sim Q$, iff, for all a,*

i) Whenever $P \xrightarrow{a} P'$ then, for some Q', $Q \xrightarrow{a} Q'$ and $P' \sim Q'$

ii) Whenever $Q \xrightarrow{a} Q'$ then, for some P', $P \xrightarrow{a} P'$ and $P' \sim Q'$

This notion of equivalence is not the same as transition graph isomorphism. Although graph isomorphism implies transition system equivalence, equal transition systems may not have isomorphic transition graphs. This notion of equality can be used to define a normal or canonical form for directed labelled transition systems. The normal form of a DLTS is the coarsest member of the bisimulation equivalence class, i.e. the equivalent transition system which has the minimum number of states.

Of special interest in behavioural specification is the class of strongly determinate directed labelled transition systems. This is the class of transition systems that do not make non-deterministic choices between transitions with the same label. This restricted class of behaviours may be completely specified by trace structures.

Definition 3 *The state s of the directed labelled transition system (S, s_0, I, O, T) is* strongly determinate *iff for all states $s_1, s_2 \in S$, if $s \xrightarrow{a} s_1$ and $s \xrightarrow{a} s_2$ for any label $a \in I \cup O$, then s_1 and s_2 are bisimilar, $s_1 \sim s_2$. A DLTS is called strongly determinate if all its states are strongly determinate.*

2.1.2 Operational Behaviour

An digital circuit has associated with it a finite set of terminals (or ports) used to communicate with its environment. These terminals are partitioned into a set of input ports I and a set of output ports O. The observable behaviour of such a circuit can be represented by a directed labelled transition system $(S, s_0, I, O, Tran)$ where each terminal of the circuit's interface is denoted by a unique label.

Each state of the transition system represents an instant in the behaviour of the component, with initial state representing the starting state of the circuit. Each state transition represents a possible communication action, or *signal*, between

the circuit and environment on the terminal denoted by the transition's label. The transitions from a given state specify the set of *permissible* actions that may be produced at that point in the circuit's behaviour, and by whom. Transitions labelled by elements of I mean that the environment may produce that communication signal next, and those labelled by O allow the circuit to generate the corresponding output action next. Once the communication has taken place, the behaviour of the circuit is represented by the poststate of the performed transition. In addition to above *safety* constraint, the behavioural specification of a component also contains a *liveness* constraint. A safety property asserts that "nothing bad will happen", while a liveness property asserts that "something good will happen". If in a given state, there exists a transition from that state labelled by O, then the behavioural state of the circuit must eventually advance. This progress requirement guarantees that the environment can never wait indefinitely for a valid output to be produced.

Operationally, the occurrence of a communication action represents a change of voltage, a *transition*, at the corresponding terminal. No distinction is made between rising and falling voltage transitions. Hence the same state transition may correspond to both low-to-high and high-to-low voltage transitions. Although it is possible to 'unfold' a transition system such that each label denotes either a rising or falling transition, specifications that do not make this distinction tend to be much smaller. By convention, all terminals of a circuit are considered to initially low, unless stated otherwise.

The complete labelled transition system contains all communication behaviours that may take place between the component and its environment. The specification of the boundary between component and environment acts as a contract between them. The correct behaviour of the component should be guaranteed, provided the environment behaves only as prescribed. Hence the behavioural specification both defines the correct behaviour of a circuit and restricts the behaviour of the environment.

As behavioural specifications detail the permitted observational behaviour of a circuit's environment, they may be used to specify the class of correct 'environment components'. The specification for a *live* environment may be found by *reflecting* the directed labelled transition system. The reflection of a strongly determinate DLTS L, denoted by $\overline{L}$, is obtained by exchanging the elements of the sets I and O. In this way, every output action of the circuit becomes an input action of the environment and vice versa.

This interpretation of the permissible interface behaviour is similar to the use of the *must* and *may* modalities of modal transition systems [67]. In this framework, an implementation must be able to accept all the inputs and may generate any of the outputs that are prescribed in the specification. In this case, the transitions $\xrightarrow{a?}$ and $\xrightarrow{a!}$ can be seen as variants of the $\rightarrow_{\Box}$ and $\rightarrow_{\Diamond}$ relations.

Asynchronous circuit behaviours that are not 'strongly determinate' pose interesting problems. Consider the three states s_1, s_2 and s_3 of a DLTS such that $s_1 \xrightarrow{a} s_2$ and $s_1 \xrightarrow{a} s_3$ for any symbol a where the states s_2 and s_3 are not bisimilar $s_2 \not\sim s_3$. After the communication action a, the circuit is in one of two states, indistinguishable by the environment. To ensure correct operation in such an event, the environment may only send input signals that may be received in all such indistinguishable states and must be able to accept output communication actions from any such state. Hence, for any non-strongly determinate DLTS it is possible to determine an 'equivalent' strongly determinate behaviour that ensures safe operation. The proposed design methodology assumes (ensures) that a DLTS specifies the 'safe' strongly determinate behaviour of an asynchronous circuit, and issues an error (warning) otherwise.

2.2 Circuit Specification

Directed labelled transition systems serve as suitable abstract model of asynchronous circuit behaviour, but lack a convenient notation for a specification formalism. For this reason, we give a labelled transition system semantics to a number of common circuit specification styles. This allows the use of conventional descriptions to clearly and concisely define required circuit behaviours. Note that some of the formalisms mentioned below are less expressive than DLTSs and hence are used to describe restricted classes of circuit behaviour.

2.2.1 Process Algebra

Process algebra [53,89] provides a framework for describing the modular structure of concurrent systems (or *processes*) and also details their operational behaviour. Syntactically, processes form a term algebra, where terms are built by a given set of operators that typically include a parallel composition operator.

In this thesis, circuit specifications are given in a small subset of Milner's Calculus of Communicating Systems (CCS) [89]. This subset has sufficient expressive power to allow the description of any directed labelled transition system. The *agent*, or process, `Nil` can perform no communication action. The *prefix* operator, $a.E$, where a is an action label and E represents an agent expression, performs the communication action a before evolving into the process E. By convention, communication labels begin with a lower case letter. The *choice*, or sum, of two agents E_1 and E_2, written $E_1 + E_2$, non-deterministically behaves like either E_1 or E_2. Action prefixing binds tighter than the choice operator. We shall assume a set of *agent identifiers*, which may be bound to an agent expression that may contain a reference to itself. This enables the definition of recursive processes. By convention, agent identifiers are labels beginning with an upper case letter (unique from `Nil`). An agent definition has the form `bi` A E, where the identifier A is associated with the agent expression E. This syntax is the same as that used by the original version Concurrency Workbench process algebra tool [30].

The directed labelled transition system describing the observable behaviour of an agent can be determined using the structural operational semantics (SOS) rules of that process algebra [98]. The initial agent, by convention the first agent identifier defined in a circuit specification, is associated with the initial state s_0 of the DLTS. If the agent P, associated with a state s_1 of a labelled transition system, can perform a communication action α, then the DLTS has a state s_2 that denotes the α-derivative of P, and $s_1 \xrightarrow{\alpha} s_2$. The process $\alpha.E$ has the α-derivative E. If P has an α-derivative P', then so do the agents $P + E$ and $E + P$, for any agent E and communication action (label) α. This definition allows the derivation of a directed labelled transition system by syntactic search for reachable states.

For example, the directed labelled transition system described by the transition graph in figure 2–1 is given by the CCS specification in figure 2–2 below.

```
bi S0  a?.b?.S1 + b?.a?.S1 + c?.p!.S0
bi S1  q!.d?.S1
```

Figure 2–2: Example CCS circuit specification

It is trivial to extend this technique to allow asynchronous circuit specifications to be given in the complete CCS syntax, including restriction, relabelling and various composition operators. Similarly, for other process algebras such as CSP [53,

59], Circal [87], Esterel [6], ACP [3] and LOTOS. The principal advantage of this approach over the formalisms used by other circuit synthesis methodologies is that the semantics of the process algebra's operators need not be that of circuit composition. This allows the specification to describe the circuit's required observational behaviour without determining its structure.

2.2.2 Trace Theory

The most common formalism for describing delay insensitive circuit behaviour is trace theory [36,112,122]. Trace theory specifies circuit behaviour by explicitly stating all permissible sequences of communication actions that may take place between a circuit and its environment.

Definition 4 *A* directed trace structure T *is a pair* $\langle \mathbf{a}T, \mathbf{t}T \rangle$ *in which* $\mathbf{a}T$ *is a finite set of symbols partitioned into two sets* $\mathbf{i}T$ *and* $\mathbf{o}T$*, and* $\mathbf{t}T$ *is a set of finite-length prefix-closed sequences of elements of* $\mathbf{a}T$*, which are called* traces.

The set of symbols $\mathbf{a}T$ is referred to as the *alphabet* of T, which is split into the input alphabet, $\mathbf{i}T$, and the output alphabet, $\mathbf{o}T$. The trace set, $\mathbf{t}T$ specifies all permissible communication sequences at the mechanism's interface with its environment. Since trace sets are often infinite, representation by enumeration of their elements is unreasonable, so traces structures are often defined using a program notation based on regular expressions called *commands.*

Definition 5 *A* trace command *over the alphabet of symbols (or labels)* Σ *is either* ϵ*, any symbol* $a \in \Sigma$*, or any expression of the form* $S;T$*,* $[S]$*,* $S|T$*,* $S\lceil A$*,* $S\|T$ *or* $pref\ S$ *where* S *and* T *are trace commands over* Σ*, and* A *is a set of symbols.*

Each trace command E over an alphabet Σ denotes a directed trace structure $\langle \mathbf{a}E, \mathbf{t}E \rangle$ where $\mathbf{a}E \subseteq \Sigma$. The command ϵ denotes the trace structure $\langle \emptyset, \{\epsilon\} \rangle$ (where $\{\epsilon\}$ is the set containing the empty trace), the *atomic* command $a \in \Sigma$ denotes the trace structure $\langle \{a\}, \{a\} \rangle$ and each of the *concatenation*, *union*, *repetition*, *prefix closure*, *projection* and *weaving* operators is defined as follows.

$$
\begin{aligned}
S;T &= \langle \mathbf{a}S \cup \mathbf{a}T, \{st \mid s \in \mathbf{t}S \wedge t \in \mathbf{t}T\} \rangle \\
S|T &= \langle \mathbf{a}S \cup \mathbf{a}T, \mathbf{t}T \cup \mathbf{t}S \rangle \\
[S] &= \langle \mathbf{a}S, (\mathbf{t}S)^* \rangle
\end{aligned}
$$

$$\begin{aligned}
\text{pref } S &= \langle \mathbf{a}S, \{s \mid \exists t.st \in \mathbf{t}S\}\rangle \\
S\lceil A &= \langle \mathbf{a}S \cap A, \{s\lceil A \mid s \in \mathbf{t}S\}\rangle \\
S\|T &= \langle \mathbf{a}S \cup \mathbf{a}T, \{t \in (\mathbf{a}S \cup \mathbf{a}T)^* \mid t\lceil \mathbf{a}S \in \mathbf{t}S \wedge t\lceil \mathbf{a}T \in \mathbf{t}T\}\rangle
\end{aligned}$$

where $t\lceil A$ is the trace t from which all symbols not in set A have been removed. To save parentheses, unary operators have the highest priority, followed be weaving, concatenation, union and finally projection has the lowest priority. Snepscheut [122] presents proofs for several interesting properties for these trace operators. Ebergen [36] extends this notation to include general tail recursion, similar to the agent identifiers used in process algebras.

We can introduce the notion of states of a trace structure by determining an equivalence relation $\sim_R$ on traces of a prefix closed trace T. The relation $t \sim_R s$ holds iff for any trace $r \in \mathbf{a}T$, $tr \in \mathbf{t}T \Leftrightarrow sr \in \mathbf{t}T$, i.e. the $\sim_R$ relation holds between traces with the same continuations. The equivalence classes of this relation form the states of the trace structure T. The state containing the trace t is denoted by $[\![t]\!]$. The directed labelled transition system representing the directed trace structure T, has input label set $\mathbf{i}T$, output label set $\mathbf{o}T$ and initial state $[\![\epsilon]\!]$. There is a transition $[\![s]\!] \xrightarrow{a} [\![t]\!]$ for any label $a \in \mathbf{a}T$ iff $sa \sim_R t$. One corollary of this definition is that any directed labelled transition system described by a trace structure is strongly determinate. As an example, the trace command $\text{pref}([c?; p!]; (a?||b?); [q!; d!])$ denotes the directed labelled transition system of figure 2–1 on page 21.

A similar notation, based on regular expressions, is used by several asynchronous circuit researchers to describe directed labelled transition system that do not have a branching structure. These notations are called *cyclograms* or *taxograms* by Starodubtsev [107] and *handshaking expansions* by Martin [76]. The syntax of both notations annotate transitions depending on whether the signal is rising or falling. Martin suffixes output signals with '↑' for low-to-high transitions, and the suffix '↓' for high-to-low transitions. The input actions of a circuit are described syntactically as boolean expressions of signal names enclosed square brackets (not to be confused with the repetition operator of trace commands) called *wait conditions*. A handshaking expansion is an alternating sequence of input and output communication actions separated by semi-colons enclosed by a repetition operator $*[\ldots]$. An example handshaking expansion is given in figure 2–3 below.

$$*[[li];\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri];\ lo\uparrow;\ [\neg li];\ lo\downarrow]$$

Figure 2–3: Example handshaking expansion

Translating this syntax into a directed labelled transition system is straightforward; the example given above specifies the same DLTS as the trace command pref$[li?; ro!; ri?; ro!; ri?; lo?; li?; lo!]$. Starodubtsev uses an almost identical syntax using the characters '+' and '-' to denote transition polarity and delimiting the repetition sequence by -> and <-. However, cyclograms may also contain an initial sequence of communication actions that are only performed once.

2.2.3 Petri Nets

Another popular representation of asynchronous circuit behaviour, typically used by speed independent circuit researchers, is the *Petri net*. This formalism, developed by C.A. Petri, is widely used as a model of concurrent systems. Structurally, a Petri net is a bipartite directed graph with two types of nodes: *places* and *transitions*. This graph represents the relationship between conditions and events in a system. The places (represented graphically by circles) correspond to conditions and the transitions (represented graphically by thick lines) correspond to events.

Definition 6 *A* Petri net *is a four-tuple* (P, T, F, M_0)*, where* P *is a set of* places*,* T *is a set of* transitions *and* F *is the* flow relation *between the places and the transitions* $F \subseteq (P \times T) \cup (T \times P)$ *and* M_0 *is the* initial marking *(or "state") of the net.*

Places within the net are capable of containing some number of 'tokens' or 'markers', such that a place is said to hold or be true if a token is present in it. Places are connected by directed arcs, via transitions, along which tokens may pass. The movement of tokens along the arcs is controlled by the occurrence of a *firing* of a transition, known as an *event*.

A condition is said to be *incident* on an event if there is a directed arc from the condition to the event. If there exists a directed arc from an event to a condition, the condition is a *successor* of that event. If all members of the input set of an event

hold, the event is "enabled" and sometime later will "fire", removing the tokens from its input set and placing tokens in all members of its output set. Multiple arcs directed away from a place indicate that a token may travel over either arc, but not both. Multiple arcs directed to a condition indicate that a token may enter the condition through one of several paths. The *marking* M (or state) of a Petri net is the set of conditions which hold at an instant of time (the assignment of tokens to places).

A net is *live* if, for any event, it is impossible for the net to reach a state from which that event cannot be enabled. A net is *safe* if there can never be more than one token in a condition at one time. Two events which share a common input place can be in *conflict* if both events are enabled at the same time. If there exist 'conflicting' events, it is indeterminate which will occur; however the first will disable the other. A net which is not safe or has conflicting events can often have these situations resolved by properly constraining the inputs to the net. If a Petri net has an upper bound on the number of tokens that can appear in a marking (in which case it is said to be *bounded*), it can be regarded as a regular sequence of transition firings.

The directed labelled transition system represented by a labelled bounded Petri net, or *STG*, is obtained by "executing" the net. A Petri net is executed by examining all the markings reachable from M_0. There exists a transition $s_1 \xrightarrow{a} s_2$ in the DLTS, if there exists a marking M_2 (corresponding to state s_2) which is reached from marking M_1 (corresponding to state s_1) by firing a single transition t, where t is labelled by the communication action a. The state s_0 of the DLTS represents the net's initial marking M_0.

The example Petri net given in figure 2–4 below, denotes the same directed labelled transition system as given by the transition graph in figure 2–1.

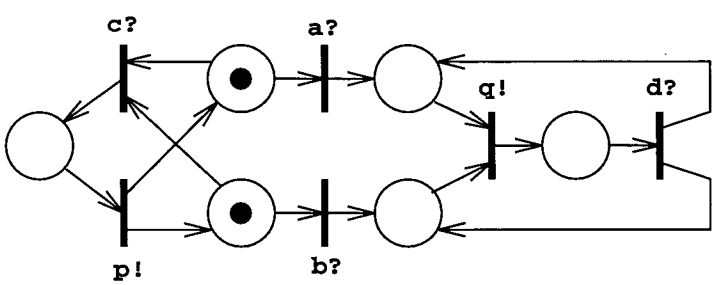


Figure 2–4: Example Petri net circuit specification

Speed independent circuit researchers conventionally annotate Petri net transitions with both the signal name and whether the signal changes from high-to-low or low-to-high. This leads to larger Petri nets than those that do not differentiate polarity, and requires the specification to be checked that the rising and falling transitions of each signal strictly alternate (called *switch-over correct* [124]).

2.3 Circuit Composition

In this section, we develop a composition operator on directed labelled transition systems that models interconnection of asynchronous components. This allows us to define an algebra of asynchronous circuit behaviour, and provides a basis for our design methodology. The definition of circuit composition states that all communication is point-to-point and hence all internal behaviour is hidden from the environment. Although this is reasonable for delay insensitive circuit synthesis, a more general operator is required for speed independent design.

We consider the composition of two directed labelled transition systems $\mathcal{P}$ and $\mathcal{Q}$, written $\mathcal{P}\|\mathcal{Q}$, where the elements of $\mathcal{P}$ and $\mathcal{Q}$ are $(S_p, p_0, I_p, O_p, T_p)$ and $(S_q, q_0, I_q, O_q, T_q)$ respectively. Because the composition operator is both associative and commutative, arbitrary networks may be composed in any order.

For these components to be composed, we require that their input and output ports be disjoint, $O_p \cap O_q = \emptyset$ and $I_p \cap I_q = \emptyset$. This constraint on outputs means that circuits containing bidirectional wires and buses cannot be modelled. However, because such components are not insensitive to wiring delays, we are not restricted by this formalism. The disjoint input constraint requires that inputs common to different components are modelled explicitly by *fork* components. It is possible to define a composition operation on components with shared inputs, as 'syntactic sugar', by renaming the appropriate ports and composing the resulting composite component with the required forks.

The set of ports $O_p \cap I_q$ represent the internal wires used for signalling from $\mathcal{P}$ to $\mathcal{Q}$ and the set $O_q \cap I_p$ the internal wires from $\mathcal{Q}$ to $\mathcal{P}$. Composition of directed labelled transition systems hides all internal terminals used for communication and synchronization between the components. Hence, the observable input and output port sets of $\mathcal{P}\|\mathcal{Q}$ are given by $(I_p - O_q) \cup (I_q - O_p)$ and $(O_p - I_q) \cup (O_q - I_p)$ respectively. In order to define the operational behaviour of $\mathcal{P}\|\mathcal{Q}$, we shall make

use of the notion of a *composite state.* The set of composite states of the DLTS $\mathcal{P}\|\mathcal{Q}$ is the cartesian product of S_p and S_q. The initial composite state of the composition of two directed labelled transition systems is the tuple formed from the initial states of each DLTS, (p_0, q_0).

In order to define the observable behaviour of the composition of two directed labelled transition systems, it is necessary to introduce a number of useful definitions. For example, it is convenient to distinguish between communication actions that are internal to the composed system from those that are externally observable. An very important property of a state of a transition system is its *stability.* This property, based upon a state's ability to emit an output signal, will fundamental to much of the theory described in this thesis.

Definition 7 *The state s of the directed labelled transition system (S, s_0, I, O, T) is stable, written Stable(s), if there does not exist any output transition $s \xrightarrow{o} s'$ for any output label $o \in O$. Similarly, the state s is unstable, written Unstable(s), iff it is not stable, i.e. the state is the prestate of an output transition.*

A composite state is unstable, if either of its members is unstable. An unstable composite state may perform an output communication action and evolve into another composite state, which may also be unstable. We distinguish between internal communication actions and those observable by the composition's environment. The *internal derivatives* of a composite state are the composite states that are reachable by internal communication actions only.

Definition 8 *The set of* internal derivatives *of a composite state (p, q) of directed labelled transition systems $\mathcal{P}$ and $\mathcal{Q}$, written int(p, q), is defined as*

i) $(p, q) \in int(p, q)$

ii) For any $(r, s) \in int(p, q)$, *if there exists a label* $a \in (O_p \cap I_q) \cup (O_q \cap I_p)$ *such that* $r \xrightarrow{a} r'$ *and* $s \xrightarrow{a} s'$ *then* $int(r', s') \subseteq int(p, q)$.

iii) Nothing else is in $int(p, q)$.

The set of internal derivatives of a state are the states of the composite circuit that are indistinguishable to the composition's environment. After a visible transition, all that may be assumed about the state of the composite circuit is that it is one of the internal derivatives of the destination state. Similarly the *observable derivatives* of a composite state are all those composite states that may be reached by only internal or output communication actions.

Definition 9 *The set of* observable derivatives *of a composite state (p, q) of directed labelled transition systems $\mathcal{P}$ and $\mathcal{Q}$, written $obs(p, q)$, is defined as*

i) $int(p, q) \in obs(p, q)$

ii) For any $(r, s) \in obs(p, q)$, *if there exists a label* $a \in (O_p - I_q)$ *such that* $r \xrightarrow{a} r'$ *then* $obs(r', s) \subseteq obs(p, q)$.

iii) For any $(r, s) \in obs(p, q)$, *if there exists a label* $a \in (O_q - I_p)$ *such that* $s \xrightarrow{a} s'$ *then* $obs(r, s') \subseteq obs(p, q)$.

iv) Nothing else is in $obs(p, q)$.

The set of observable derivatives of a state are those states that may be reached by the composition's own actions. Once a given state is reached via interaction with the environment, the environment can do nothing to prevent the circuit entering any of the observable derivatives of that state, they are in some sense 'inevitable'.

The principal condition for correct operation of a connection of components is that it is free from *interference*. A composition of two circuits exhibits interference when an internal signal may be generated by one component before it is ready to be received by the other. In this case, the first component violates the constraints placed on the environment of the second by its behavioural specification.

Definition 10 *The composite state (p, q) of directed labelled transition systems $\mathcal{P}||\mathcal{Q}$ is defined to* interfere, *written $p \leftrightsquigarrow q$, if either there exists a transition $p \xrightarrow{a} p'$ where $a \in O_p \cap I_q$ and there is no state q' such that $q \xrightarrow{a} q'$ or there exists a transition $q \xrightarrow{a} q'$ where $a \in O_q \cap I_p$ and there is no state p' such that $p \xrightarrow{a} p'$.*

We state that the communication actions permissible at the interface of $\mathcal{P}||\mathcal{Q}$ guarantee that the composition can never reach a state where the components could interfere and that internal communication actions are synchronized. The behaviour of $\mathcal{P}||\mathcal{Q}$ is defined using a mapping from composite states to states of the directed labelled transition system, the initial state of which is (p_0, q_0). Two DLTSs may be composed only if there is no interfering state in the observable derivatives of the initial state.

Definition 11 *The composition of two directed labelled transition systems $\mathcal{P}$ and $\mathcal{Q}$, written $\mathcal{P}||\mathcal{Q}$, has the set of input labels $(I_p - O_q) \cup (I_q - O_p)$, the set of output labels $(O_p - I_q) \cup (O_q - I_p)$ and initial state (p_0, q_0). The transition relation of $\mathcal{P}||\mathcal{Q}$ contains the transition $(p, q) \xrightarrow{a} (p', q')$ when*

i) For $a \in O_p - I_q$, iff $r \xrightarrow{a} p'$ for some composite state $(r, q') \in int(p, q)$.

ii) For $a \in O_q - I_p$, *iff* $s \xrightarrow{a} q'$ *for some composite state* $(p', s) \in int(p, q)$.

iii) For $a \in I_p - O_q$, *iff* $r \xrightarrow{a} p'$ *for some composite state* $(r, q') \in int(p, q)$ *and this input action cannot result in the composition entering a state that could interfere, i.e. for all states* $(r, s) \in int(p, q)$ *and transitions* $r \xrightarrow{a} r'$ *for some state* r', *there is no state* $(u, v) \in obs(r', s)$ *that interferes,* $u \leftrightsquigarrow v$.

iv) For $a \in I_q - O_p$, *iff* $s \xrightarrow{a} q'$ *for some composite state* $(p', s) \in int(p, q)$ *and this input action cannot result in the composition entering a state that could interfere, i.e. for all states* $(r, s) \in int(p, q)$ *and transitions* $s \xrightarrow{a} s'$ *for some state* s', *there is no state* $(u, v) \in obs(r, s')$ *that interferes,* $u \leftrightsquigarrow v$.

Our notion of circuit behaviour places very strong constraints on input and output communications. An input communication may only be sent to the composition in an observable state, if all internal derivatives of that state allow the receipt of that signal. Otherwise there is a possibility of the circuit being in a non-receptive state when the signal is sent causing interference between circuit and environment. Similarly, an input may only be sent to a composition, if it is impossible for any of the observable states of any of the destinations to interfere. Note that the composition is only strongly determinate if the effects of receiving an input signal are the same for every internal derivative of the observable state.

With the above definition of composition, it is possible for a composite circuit to become *live-locked.* This is a condition that can arise if there exists a cycle or loop in the internal derivatives of a state, i.e. if there are two states (r, s) and (u, v) in the derivative set $\mathbf{int}(p, q)$ such that $(r, s) \in \mathbf{int}(u, v)$ and $(u, v) \in \mathbf{int}(r, s)$. In such a state, the circuit may always decide to perform an internal communication action rather than generate an output signal. This possibility of 'indefinite postponement' means that the outputs from that state can no longer guaranteed to be live. An environment may wait indefinitely for such an output signal that may never be generated. If the outputs generated by a circuit are required to be live, the circuit's behaviour must be shown to avoid the possibility of entering a live-locked state. This can be done by using a method similar to that for avoiding interference in the definition above.

2.4 Implementation Relation

In the previous sections, we defined a model for reasoning about the behaviours of asynchronous circuits. Directed labelled transition systems may be used both to describe the actual and required behaviours of a circuit. This notion of circuit specification prescribes formally a number of desired properties of a correct implementation's operational behaviour. In this section, we define what it means for one transition system to *implement* or *refine* another.

A specification can be regarded as an idealized component, which may be used conceptually in an abstract design. A valid implementation of this specification is a circuit behaviour that can be 'substituted' for an idealized component while preserving the correct behaviour of the design in which it is instanced. The advantage of using the same formalism for both specification and implementation is that it allows both hierarchical design and verification of circuits. The description of a component at one level of abstraction may be used as a circuit specification at lower level of abstraction. Among the many advantages this approach offers is the use of libraries of common subcircuits as abstract components at higher levels of design.

The relationship between a specification and an implementation should allow the implementation to exceed the minimum requirements stated in the specification. This is similar to other engineering disciplines where a component with a 1% tolerance may be used safely in a design that specifies a 5% component. Similarly, we also allow the specification to describe a number of design alternatives. Hence, we allow a non-deterministic circuit specification to be replaced with an indistinguishable deterministic one. This appeals to the engineering maxim, "a component is correct if it can not be shown to be faulty".

It is possible to use bisimulation equivalence as potential implementation relation, but this would be overly restrictive, requiring the actual implementation to be 'identical' to the specification. An alternative implementation relation, between prefix closed directed trace structures, has been proposed by Dill [34]. This relation, called *conformation*, is used by his model checking software to automatically verify asynchronous circuit designs.

Definition 12 *Dill's conformation states that the behaviour of an implementation* P conforms *to a specification* Q*, written* $P \preceq Q$ *when*

$$\begin{aligned} P \preceq Q \quad & \textit{iff} \;\; Q \xrightarrow{i \in I} Q' \Rightarrow P \xrightarrow{i \in I} P' \;\textit{and}\; P' \preceq Q' \\ & \textit{and} \;\; P \xrightarrow{o \in O} P' \Rightarrow Q \xrightarrow{o \in O} Q' \;\textit{and}\; P' \preceq Q' \end{aligned}$$

Dill's notion of a conformation requires that an implementation must be able receive any input that may be sent by the specification's environment and may only generate those outputs permitted by the specification.

This potential implementation relation has the advantage over bisimulation that the implementation may have more behaviours than the minimum requirements of a specification. As an example of this property, consider the two circuit behaviours specified by CCS agents below. The first agent `Merge` describes a general merge component that accepts an input transition from either of two inputs and generates a transition at its single output. The `AltMerge` combines strictly alternating input signals. By Dill's definition the general merge element 'conforms' to the specification of the alternating merge.

```
bi Merge   a?.c!.Merge + b?.c!.Merge
bi AltMerge  a?.c!.b?.c!.AltMerge
```

The principal drawback with this method is that it places no 'liveness' constraints on an implementation. Consider a circuit that can accept any input and that never produces an output, termed a 'Universal Do-Nothing' module by Charles Molnar. Such a component conforms to any specification. Dill uses this example to justify the development of complete trace structures to express liveness properties of circuits.

An improved notion of implementation is given by Ebergen [36]. Although Ebergen only defines the decomposition specifications, it is possible to determine the relation used to state when one prefix closed directed trace structure 'satisfies' a specification. This satisfaction relation places a stronger constraint on an implementation than Dill's conformation. A correct implementation must be able to generate exactly those outputs performed by the specification.

Definition 13 *Ebergen's satisfaction relation states that the behaviour of an implementation* P satisfies *a specification* Q*, written* $P \sqsupseteq Q$ *when*

$$\begin{aligned} P \sqsupseteq Q \quad & \textit{iff} \;\; Q \xrightarrow{i \in I} Q' \Rightarrow P \xrightarrow{i \in I} P' \;\textit{and}\; P' \sqsupseteq Q' \\ & \textit{and} \;\; P \xrightarrow{o \in O} P' \Leftrightarrow Q \xrightarrow{o \in O} Q' \;\textit{and}\; P' \sqsupseteq Q' \end{aligned}$$

Both Dill's and Ebergen's implementation relations permit the general merge element as a valid implementation of an alternating merge element. However, a Universal Do-Nothing module will not satisfy every Ebergen specification.

One further property of an implementation relation that would be beneficial in an asynchronous circuit synthesis is the ability to treat "static" non-determinism as describing design alternatives. In this way, specifications that state an arbitrary decision may be made at some point in a circuit's behaviour can be implemented by circuits that make a fixed decision. One example of this is shown by the pair of CCS agents below. A `Choice` component receives an input signal then makes a completely arbitrary choice between which of its two outputs to generate. A valid implementation of this circuit is the `Toggle` component that strictly alternates between its outputs after each input transition.

```
bi Choice  a?.(b!.Choice + c!.Choice)
bi Toggle  a?.b!.a?.c!.Toggle
```

An improved implementation relation with all of these properties is given in the following definition.

Definition 14 *A directed labelled transition system* $\mathcal{P}$ implements *a directed labelled transition system* $\mathcal{Q}$, *iff the initial state* p_0 *of* $\mathcal{P}$ *implements the initial state* q_0 *of* $\mathcal{Q}$, *written* $p_0 \sqsupseteq_1 q_0$, *where*

$$\begin{aligned} P \sqsupseteq_1 Q \quad & \textit{iff} \quad Q \xrightarrow{i \in I} Q' \Rightarrow P \xrightarrow{i \in I} P' \textit{ and } P' \sqsupseteq_1 Q' \\ & \textit{and} \quad P \xrightarrow{o \in Q} P' \Rightarrow Q \xrightarrow{o \in Q} Q' \textit{ and } P' \sqsupseteq_1 Q' \\ & \textit{and} \quad \textit{Unstable}(Q) \Rightarrow \textit{Unstable}(P) \end{aligned}$$

In this defintion the addition of the **Unstable** constraints is used to impose a liveness constraint on the implementation that is weaker than Ebergen's equivalence. The statement $\textbf{Unstable}(Q) \Rightarrow \textit{Unstable}(P)$ should be interpreted that if Q can make an output transition then P muct be able to make an output transition. This forces P to implement atleast one, but not all, of Q's possible output actions. This is enough to ensure that if the environment is waiting for Q to generate a signal, then it will not deadlock if Q is implemented by P.

This definition of implementation is similar to Larsen's definition of refinement in model process logic [67]. This is due to the similarity between the $\xrightarrow{a?}$ and $\xrightarrow{a!}$ transitions of directed labelled transition systems and the $\rightarrow_{\Box}$ and $\rightarrow_{\Diamond}$ modalities

of modal process logic. The principal difference in interpretation resulting from the lack of an analogue for $\rightarrow_{\Box} \subseteq \rightarrow_{\Diamond}$.

2.5 Delay Insensitivity

In this section we formalize the notion of delay insensitivity based previous trace theoretic results [34,36,112]. Informally, a network of components is 'delay insensitive' (DI) if its correct operation is independent of any delays in the response times of its components and the wires connecting those components, provided such delays are finite. An individual component is called delay insensitive if its correct operation is independent of any finite delays in the wires attached to its interface.

For networks of components, delay insensitivity requires two conditions to be complied with under composition [121]. These conditions are absence of *computation interference* and absence of *transmission interference.* Computation interference occurs when a signal arrives at a receiver before it is ready for it. Transmission interference occurs when there is more than one transition propagating along a wire at a time. Both of these types of interference result in ill defined voltage transitions and malfunctioning components. In this thesis, we shall consider transmission interference to be a special case of computation interference involving explicit 'wire' components.

The definition of circuit behaviour composition given previously assumes communications between components to be instantaneous and explicitly prohibits interference between components. It is possible to use this composition operator to define a composition operation in which all the connection wires between the two subprocesses are taken into account. This is the method used by Snepscheut in his 'agglutination' composition operator. An alternative approach is based upon the so-called *Foam Rubber Wrapper* (FRW) principle, proposed by Molnar [91]. This requires that the component behaviours being composed be delay insensitive, i.e. invariant under extension by 'wire' components. The composition of two such delay insensitive component behaviours is independent of any finite delays in its observable or internal connecting wires. A formalization of the FRW principle is given in Ebergen's thesis [36].

As mentioned above, an individual component behaviour is delay insensitive if its behaviour is invariant of inserting wire delays at the interface between the com-

ponent and its environment. To ensure a network is delay insensitive, we require that it be formed solely from the composition of delay insensitive behaviours. One method of determining if a circuit behaviour, directed labelled transition system, is delay insensitive is to explicitly check whether the component may be composed with a wire component at each terminal without affecting its observable behaviour. This is the technique used to test delay insensitivity by Dill's state based model checker [34] and Ebergen and Gingras' network verifier [39].

An alternative method of checking an asynchronous circuit behaviour for delay insensitivity is to test Udding's rules [113]. These are a set of four restrictions (and their variants) derived by Udding for prefix closed directed trace structures that are necessary and sufficient conditions for a behaviour to be delay insensitive. Proofs that these rules are equivalent to the FRW postulate are presented in Udding's thesis [112]. Udding's principal rules are presented below for any state $s_1 \in S_p$ of the directed labelled transition system $(S_p, p_0, I_p, O_p, T_p)$.

$\mathbf{R}_0$ For any symbol $a \in I_p \cup O_p$, if there exists a state s_2 such that $s_1 \xrightarrow{a} s_2$ then there exists no state s_3 such that $s_2 \xrightarrow{a} s_3$.

$\mathbf{R}_1$ For any pair of symbols a and b of the same type (either $a, b \in I_p$ or $a, b \in O_p$), if there exists two states s_2 and s_3 such that $s_1 \xrightarrow{a} s_2$ and $s_2 \xrightarrow{b} s_3$ then there exist a pair of states s_2' and s_3' such that $s_1 \xrightarrow{b} s_2'$ and $s_2' \xrightarrow{a} s_3'$ and $s_3 \sim s_3'$.

$\mathbf{R}_2$ For any pair of symbols a and b of different types (either $a \in I_p$ and $b \in O_p$ or $a \in O_p$ and $b \in I_p$), if there exist three states s_2, s_3 and s_2' such that $s_1 \xrightarrow{a} s_2$, $s_2 \xrightarrow{b} s_3$ and $s_1 \xrightarrow{b} s_2'$ then there exists a state s_3' such that $s_2' \xrightarrow{a} s_3'$ and $s_3 \sim s_3'$.

$\mathbf{R}_3'$ For any pair of distinct symbols a and b, $a, b \in I_p \cup O_p$ if there exist two states s_2 and s_2' such that $s_1 \xrightarrow{a} s_2$ and $s_1 \xrightarrow{b} s_2'$ then there exists a state s_3 such that $s_2 \xrightarrow{b} s_3$.

$\mathbf{R}_3''$ For any pair of distinct symbols a and b, $a, b \in I_p \cup O_p$, not both input symbols (either $a \in O_p$ or $b \in O_p$), if there exist two states s_2 and s_2' such that $s_1 \xrightarrow{a} s_2$ and $s_1 \xrightarrow{b} s_2'$ then there exists a state s_3 such that $s_2 \xrightarrow{b} s_3$.

$\mathbf{R}_3'''$ For any pair of symbols a and b of different types (either $a \in I_p$ and $b \in O_p$ or $a \in O_p$ and $b \in I_p$), if there exist two states s_2 and s_2' such that $s_1 \xrightarrow{a} s_2$ and $s_1 \xrightarrow{b} s_2'$ then there exists a state s_3 such that $s_2 \xrightarrow{b} s_3$.

Rule $\mathbf{R}_0$ guarantees the absence of transmission interference by limiting the number of consecutive transmissions on a wire to at most one. Rule $\mathbf{R}_1$ states that two signals being sent one after another in the same direction via different wires need not be received in the order in which they are sent. All delay insensitive trace structures satisfy rules $\mathbf{R}_0$ and $\mathbf{R}_1$. Note that rule $\mathbf{R}_0$ forces the DLTS transition relation to be *antireflexive*, i.e. the prestate and poststate of every transition of a delay insensitive DLTS must be different. Rule $\mathbf{R}_2$ states that if at some phase of a computation signals are not ordered, then the order should be 'intuitively' immaterial.

The remaining rules may be used to form a classification of delay insensitive behaviours. The class satisfying rules $\mathbf{R}_2$ and $\mathbf{R}'_3$ is called the *synchronization class*, denoted by $\mathbf{C}_1$. A specification of this class allows synchronization only, due to the absence of internal decisions. The class allowing decisions to be made as due to input, satisfying rules $\mathbf{R}_2$ and $\mathbf{R}''_3$, is called the *data communication class*, denoted $\mathbf{C}_2$. Class $\mathbf{C}_3$, called the *arbitration class*, allows a circuit to chose between two output symbols (rules $\mathbf{R}_2$ and $\mathbf{R}'''_3$). Obviously, $\mathbf{C}_1 \subseteq \mathbf{C}_2 \subseteq \mathbf{C}_3$ since $\mathbf{R}'_3 \Rightarrow \mathbf{R}''_3$ and $\mathbf{R}''_3 \Rightarrow \mathbf{R}'''_3$.

Udding [113] then states that the intuitive reasoning behind rule $\mathbf{R}_2$ is overly restrictive to ensure that a trace structure is delay insensitive. To relax or weaken this condition, he derives the following rule:

$\mathbf{R}'_2$ for traces s and t and for symbol $b \in \mathbf{a}T$ of another type than symbols $a \in \mathbf{a}T$ and $c \in \mathbf{a}T$ $\quad sabtc \in \mathbf{t}T \wedge sbat \in \mathbf{t}T \Rightarrow sbatc \in \mathbf{t}T$

The class of trace structures that satisfy rules 0, 1, $2'$ and $3'''$ is called $\mathbf{C}_4$ or the *class of delay insensitive trace structures.* Notice that $\mathbf{R}_2 \Rightarrow \mathbf{R}'_2$ and that therefore $\mathbf{C}_3 \subseteq \mathbf{C}_4$. Udding [112] proved that the rules for this class are both necessary and sufficient to guarantee the absence of both communication and transmission interference, i.e. formalize to the FRW principle.

One of the principal advantages in using Udding's rules to check the delay insensitivity of an asynchronous circuit behaviour is that this approach also provides a useful classification of delay insensitive circuits. Each of these rules is easily translated into a formula in a suitable temporal logic such as Computation Tree Logic (CTL) [28] or the modal μ-calculus [66]. By considering the directed labelled transition system a model for the logic, it is possible to check whether the model satisfies a given formula. This approach is termed 'model checking' and a

number of efficient algorithms have been developed to perform such checking. An example of such a model checker for CTL is described by Clarke *et al.* [29,35]. By identifying which of Udding's rules of delay insensitivity are violated, it is possible to determine whether the circuit is delay sensitive or to which class a component behaviour the circuit belongs.

Typically in a delay insensitive circuit synthesis system, once the model checker determines a circuit specification violates delay insensitivity, compilation is terminated with an error message. However, it is possible to modify some delay sensitive specifications to enforce their delay insensitivity. This ability to 'correct' partially delay insensitive specifications allows a shorthand representation of DI circuit behaviour. For example, the interleaving of communication actions of the same type requires the user to specify all permutations of a given set of signals. By automatically determining all such permutations, the designer need only describe a single representative permutation of the circuit's required behaviour.

Of the four rules, only rules $\mathbf{R}_1$ and $\mathbf{R}_2$ may be 'corrected'. Rule $\mathbf{R}_0$ and the variants of rule $\mathbf{R}_3$ determine the required behaviour of the circuit, and breaking these rules reflects a fundamental flaw in the specification. Violations of rules $\mathbf{R}_1$ and $\mathbf{R}_2$ may be considered an abbreviated form of the required behaviour, and the complete behaviour may be determined by adding states and transitions to ensure their satisfaction. The first form of contravention to be handled, occurs where an interleaving of signals is not given. By adding either a single transition, or an extra state and two transitions (depending on whether a partial interleaving exists) the interleaving requirement is met. The other form of exception occurs when interleaving of communication actions lead to non bisimilar states. In this case, a construction similar to that for determining the 'safe' strongly determinate behaviours of a non strongly determinate specification can be employed. Also notice that by adding transitions, the modified DLTS may no longer satisfy rule $\mathbf{R}_0$ in which case the original specification is reported as being erroneous.

Stack Example

A demonstration of 'correcting' a delay sensitive circuit specification is given in the following stack element example due to Josephs and Udding [60]. A delay insensitive stack of finite capacity may be decomposed into a number of identical stack elements. The example below describes the control section of a stack element,

which when composed with a stack of capacity N yields a stack of capacity $N+2$. One of the interesting properties of this circuit is that it has constant response time; the time taken to respond to a 'push' or 'pop' request is independent of the stack's size and its current contents/depth. Extending the specification to handle data values is straightforward.

A stack element has two inputs, `push?` and `pop?` used to add and remove items from the stack respectively. In response to a `push?` request, the stack element may respond with an `ack-push!` if the request was successful or a `full!` if the stack is already full to capacity. In response to `pop?` request, the stack element may acknowledge success with an `ack-pop!` acknowledgement or output an `empty!` signal if the stack is currently empty. The specification insists that push and pop operations are mutually exclusive, and that further requests may only be sent once the previous request has been acknowledged. A stack of zero capacity may be implemented by two wires, one connecting `push?` to `full!` and the other connecting `pop?` to `empty!`. The suggested decomposition of an arbitrary finite stack is given in figure 2–5 below.

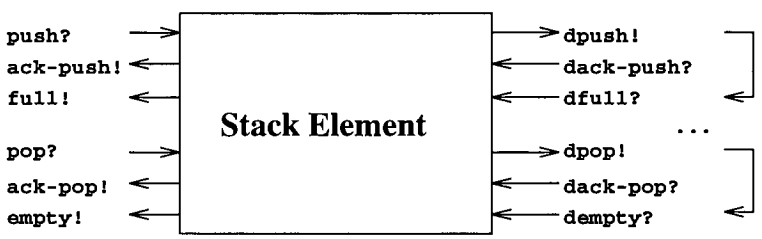


Figure 2–5: Delay insensitive stack decomposition

In this decomposition, a stack element has "upward" terminals `push?`, `pop?`, `ack-push!`, `full!`, `ack-pop!` and `empty!` that are used to communicate with the stack's environment, and the "downward" terminals `dpush!`, `dpop!`, `dack-push?`, `dfull?`, `dack-pop?` and `dempty?` that are connected to the rest of the stack. In Josephs and Udding's decomposition, a stack element is conceptually in one of three states, either empty (`E`), partially full (`P`) or full (`F`). In the empty state a push is immediately acknowledged and the state changes to `P`. In a partially full state, a push is immediately acknowledged and the current value is pushed downwards. If this downward push is successful, acknowledged by `dack-push?`, the element remains partially full otherwise on receipt of `dfull?` the stack element enters the full state, `F`. When full, a stack immediately declines push requests with

`full!`. The actions performed when popping a stack are similar due to symmetry. The CCS specification of the behaviour of an (empty) stack element is given in figure 2–6.

```
bi E pop?.empty!.E + push?.ack-push!.P
bi P pop?.ack-pop!.dpop!.W1 + push?.ack-push!.dpush!.W2
bi F pop?.ack-pop!.P + push?.full!.F

bi W1 dack-pop?.P + dempty?.E
bi W2 dack-push?.P + dfull?.F
```

Figure 2–6: Original stack element specification

The DLTS states denoted by the CCS agents `W1` and `W2` in the above specification correspond to the points ('states') in the stack element's behaviour in which the circuit is quiescent waiting for the stack below to respond to either a `dpop!` or `dpush!` signal This behavioural specification denotes the directed labelled transition system represented by the transition graph displayed in figure 2–7.

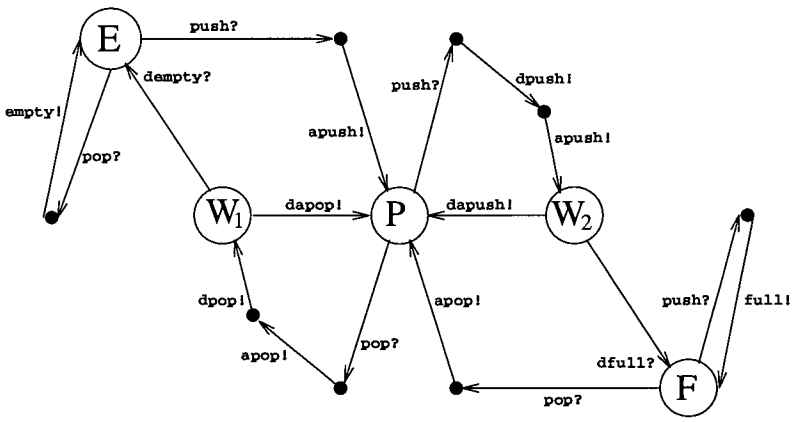


Figure 2–7: Original stack transition graph

Applying the model checking procedures described earlier, the above strongly determinate directed labelled transition fails to satisfy Udding's rules of delay insensitivity for certain states. For example, the specification states that after a partially full stack element receives a `push?` input it must first generate an `ack-push!` output and then generate a `dpush!`. This strict sequential ordering on communication actions of the same type is prohibited by rule $\mathbf{R}_1$ which requires such signals to be reordered. Intuitively, by attaching wires with unknown delays

to such a circuit, even if `dpush!` is generated after (or even because of) `ack-push!` it may reach the stack element's environment first due to adverse delays in the connecting wires. This situation may be remedied adding the requisite transitions to also allow the circuit to generate these two signals in the reverse order. Applying this 'correction' approach to the whole directed labelled transition system results in the DLTS represented by the transition graph in figure 2–8 below.

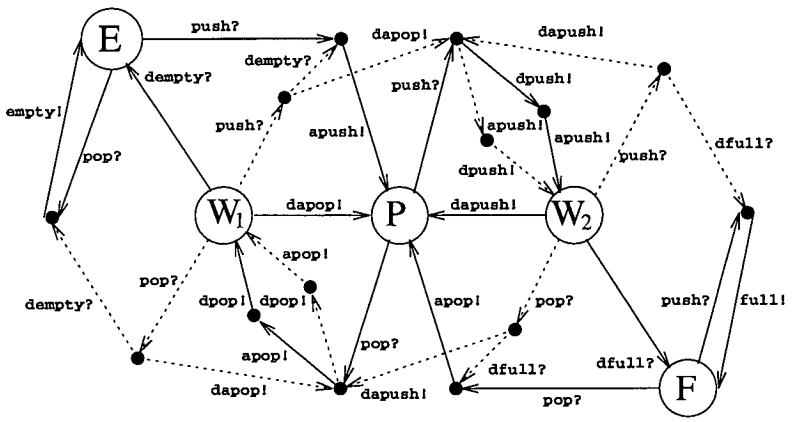


Figure 2–8: Complete stack transition graph

This modified directed labelled transition system now satisfies the constraints imposed by Udding's rules of delay insensitivity. It satisfies rules $\mathbf{R}_0$, $\mathbf{R}_1$, $\mathbf{R}_2$ and $\mathbf{R}_3''$ but breaks rule $\mathbf{R}_3'$. This identifies the stack element as a valid data communication (Udding $\mathbf{C}_2$) class component. The complete transition system contains 19 states and 34 transitions whilst the original consisted of only 13 states and 18 transitions. The advantage in 'correcting' delay sensitive behaviours is demonstrated by considering the relative sizes of the specifications for the two DLTSs. The original specification given in figure 2–6 is almost half the size of its delay insensitive equivalent in figure 2–9 below. These savings are even more dramatic for circuit behaviours that exhibit a significant amount of concurrency.

```
bi E push?.ack-push!.P + pop?.empty!.E
bi P push?.Mpush + pop?.Mpop
bi F push?.full!.F + pop?.ack-pop!.P

bi Mpush ack-push!.dpush!.Wpush + dpush!.ack-push!.Wpush
bi Mpop  ack-pop!.dpop!.Wpop + dpop!.ack-pop!.Wpop
bi Wpush dack-push?.P + dfull?.F + push?.WFpush + pop?.WFpop
bi Wpop  dack-pop?.P + dempty?.E + push?.WEpush + pop?.WEpop
bi WFpush dack-push?.Mpush + dfull?.full!.F
bi WFpop  dack-push?.Mpop + dfull?.ack-pop!.P
bi WEpush dack-pop?.Mpush + dempty?.ack-push!.P
bi WEpop  dack-pop?.Mpop + dempty?.empty!.E
```

Figure 2–9: Complete stack element specification

2.6 Interface Partitioning

Delay insensitive circuits operate in *input-output* mode. Input signals may be sent to a component or circuit as soon as the preceding outputs have been generated. Hence an input may only be sent to a delay insensitive component if it may be sent initially or in response to an 'enabling' output communication. Typically, when a delay insensitive circuit is specified, its environment is assumed to consist of a single component or mechanism that is able to observe all of the circuit's outputs and hence regulate when it sends signals to the component to the allowed points in the behaviour. The principal assumption is that for each input signal, the signal's enabling output action can be observed by the same environment that generates that input. Commonly however, a circuit's interface may be divided among a number of autonomously operating mechanisms or *subenvironments*. The specifications for these circuits must ensure that each interface may independently operate correctly. The hazard is that a subenvironment has to assume that an input signal is enabled by the previous output signal at its interface, rather than its 'true' enabling output. This requires that each input signal and its enabling output must be in the same partition of the interface.

Definition 15 *A directed labelled transition system* (S, s_0, I, O, T) *may have an* independent environment *consisting of a set of input signals,* $I_E \subseteq I$*, and output*

signals $O_E \subseteq O$, *iff for all states* $s_1 \in S$ *both the following conditions hold*

i) If there exists a pair of transitions $s_1 \xrightarrow{a!} s_2$ *and* $s_2 \xrightarrow{b?} s_3$ *where* $a \in O - O_E$ *and* $b \in I_E$ *then there exists a transition* $s_1 \xrightarrow{b?} s_4$

ii) If there exists a pair of transitions $s_1 \xrightarrow{a!} s_2$ *and* $s_2 \xrightarrow{b?} s_3$ *where* $a \in O_E$ *and* $b \in I - I_E$ *then there exists a transition* $s_1 \xrightarrow{b?} s_4$

A corollary of this definition is that if the subenvironment formed by I_E and O_E is a valid independent environment then so is the subenvironment formed by the remaining terminals $I - I_E$ and $O - O_E$.

A useful operation on directed labelled transition systems is the automatic partitioning of its environment. Much like 'correcting' delay (in)sensitivity above, this operation adds the necessary states and transitions to ensure that a given set of inputs and outputs is a valid independent environment of the circuit's behaviour. Using the names introduced in the above definition, every violation of the above rule may be 'corrected' by introducing a new state s_4 such that it has transitions $s_1 \xrightarrow{b?} s_4$ and $s_4 \xrightarrow{a!} s_3$. Once this construction has been applied to all states in the circuit's behaviour, the DLTS may need to be rechecked for delay insensitivity.

This construction allows a subenvironment to send a transition to the circuit as soon as it can determine the input may be accepted, and this input is ignored by the circuit until such time as the original specification was ready for it. A circuit level construction that performs a similar operation using decision wait elements will discussed in a later chapter.

An example of partitioning a circuit's environment is given in the one-place buffer circuit specification given below. This example is due to Martin [79] and is translated directly from the handshaking expansions used in his original paper to the CCS process algebra used in this thesis. The one place buffer has two inputs, `xi?` and `yi?`, and two outputs, `xo!` and `yo!`, that are used to implement two four-phase handshaking channels to independent environments {`xi?`,`xo!`} and {`yi?`,`yo!`}. The behaviour of this component is to receive a single 'token' on its passive X channel and then output it on its active Y channel (These terms will be described in more detail in the later section on communications protocols). The circuit initially receives an `xi?` input indicating that the environment wishes to place a token in the buffer. This request is then acknowledged by the output `xo!`. Once the environment receives this acknowledgement it withdraws the request `xi?`, to which the component withdraws `xo!`, to complete the four-phase handshake

on channel X placing a single token in the buffer. The circuit then attempts to output the value via the Y channel, by signalling the request `yo!`. When the environment is ready to accept the token, it acknowledges with `yi?`. And the cycle is completed by withdrawing the request `yo!` and waiting for the acknowledgement to be withdrawn. Martin's specification for this component is given by the CCS agent `Buffer` in figure 2–10 below.

```
bi Buffer  xi?.xo!.xi?.xo!.yo!.yi?.yo!.yi?.Buffer
```

Figure 2–10: Original one place buffer specification

Once this specification is corrected for being delay sensitive (caused by no interleaving of the signals `xo!` and `yo!` or `yi?` and `xi?`), this specification describes the directed labelled transition graph given in figure 2–11 below. Note the additional states and transitions added to ensure the circuit is a delay insensitive synchronization (Udding $\mathbf{C}_1$) class component.

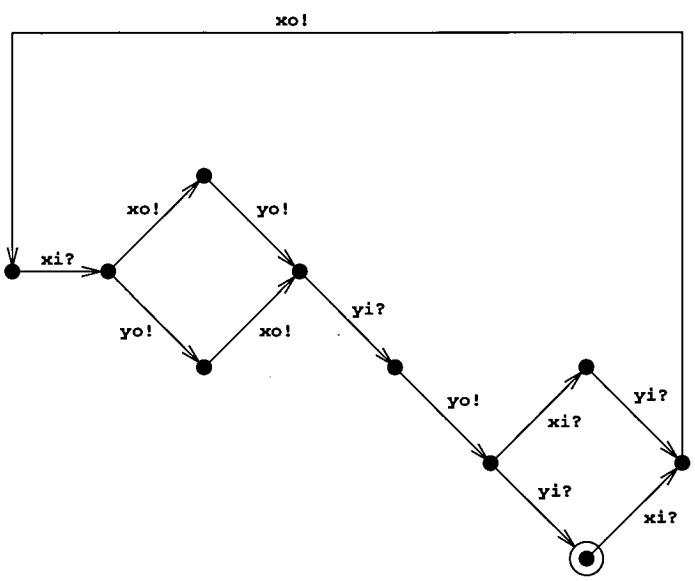


Figure 2–11: Original one place buffer transition graph

Implicit in the high level description of the one place buffer is the fact that the circuit's environment consists of two independent interfaces, {`xi?`,`xo!`} and {`yi?`,`yo!`}. In his paper, Martin even gives potential implementations of the environment as two distinct components, a wire and an invertor. The behavioural

specifications of these components (appropriately labelled) are given by the two CCS agents below.

```
bi Invertor  xi!.xo?.Invertor
bi Wire  yo?.yi!.Wire
```

However, careful inspection of the transition graph given in figure 2–11 reveals that composition of these components with the original buffer specification leads to interference. This is because {xi?,xo!} is not a valid independent environment of the original buffer specification. Consider the point in the circuit's behaviour once a token has just been placed in the buffer and the acknowledgement xo! has been withdrawn. To the 'sending' X environment, the buffer is now observationally indistinguishable from the point when it may receive another xi? request. This is because one of xi?'s enabling outputs, yo!, is not part of xi?'s partition of the circuit's interface. However, if the above construction to partition an environment is applied to the specification this problem is resolved. The resulting directed labelled transition system has the transition graph given in figure 2–12 below, which has the intended behaviour, and hence may safely be composed with the components described above. This construction provides another convenient 'short-hand' method for specifying complex circuit behaviours.

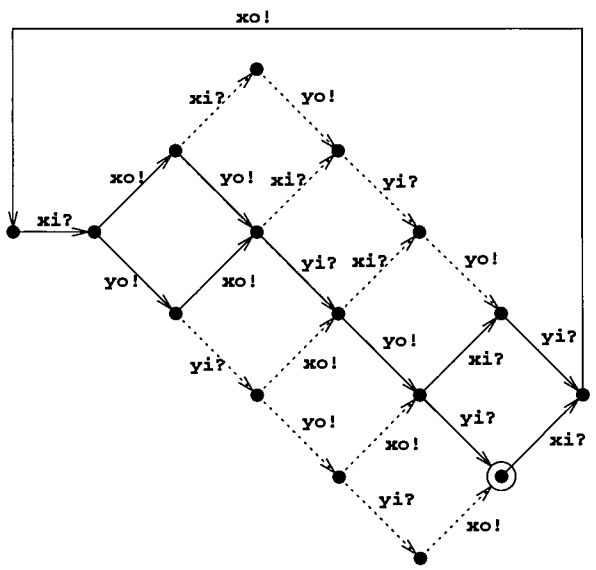


Figure 2–12: Complete one place buffer transition graph

2.7 Compiler Representation

A delay insensitive circuit synthesis system must perform some or all of the transformations and checks described in this chapter. This requires the abstract model of a circuit behaviour to be efficiently represented on a computer and suitable algorithms to apply the above techniques to this representation. The principal aim is to avoid the problem of *state explosion*, namely that a system of N components with n states each may have n^N states. Luckily, a number of approaches have been developed that permit the efficient checking and manipulation of circuit specifications.

Binary Decision Diagrams

A suitable data structure for representing directed labelled transition systems is the binary-decision diagram (BDD). A BDD is a normal form representation of a boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$, that is often much smaller than other normal forms [15]. The BDD is used to encode a binary representation of the characteristic function of the transition relation. This representation allows very large transition systems to be compactly represented within a computer. The use of additional BDD techniques, such as partitioned transition relations [18], permit current computer workstations to manipulate transition systems with several orders of magnitude more states than would be possible/practical with explicit transition graph based data structures.

An efficient implementation method for a BDDs, using reduced ordered BDDs has been described by Brace, Ruddel and Bryant [9]. Burch *at al.* [19] describe how to efficiently implement BDD model checking for a powerful modal μ-calculus (that includes bisimulation testing), and Enders, Filkorn and Taubner [42] describe a technique for construction of BDD transition graphs directly from process algebras such as CCS, using the parallel composition, restriction and relabelling operators. The resulting BDDs from such a technique grow only linearly in the number of parallel components, and hence avoid the problem of state explosion.

State Minimization

In order to reduce the time complexity of many of the operations performed on directed labelled transition systems, the specification is processed at several stages to reduce the number of states in the transition system. This amounts to determining the minimal normal form of a DLTS as described in section 2.1.1. Calculating the normal form, called *state minimization*, removes redundant and unreachable states. This procedure is one of the major advantages of using a behaviour based synthesis method over existing syntax directed translation techniques. It enables global optimization of circuit specifications, reducing a specification to a minimal behaviourally equivalent form by examining all of the circuit's states. This step typically results in significant improvements in the quality of circuits generated over existing delay insensitive circuit synthesis systems [119,14]. The algorithm used to minimize the number of states in the specification is based on 'partition refinement'. Such algorithms are commonly used to minimize the number of states in the finite automata used by lexical analyzers. An example of such an application is presented by Aho, Sethi and Ullman [1].

The minimization algorithm maintains a *partitioning* of the nodes of the graph G, which is a set of *blocks*, where each block is a set of nodes such that each node of G is contained in exactly one block. Such a partitioning naturally induces an equivalence relation on the nodes of the graph: two nodes are related if they are in the same block. The algorithm starts with the initial partition containing only a single block and then successively refines this partition. It terminates when the induced equivalence relation becomes an observational equivalence on the behaviour of the circuit. Finally, all the nodes within a single block are collapsed to a single node, to produce an equivalent graph with the minimum number of states. The current implementation is based upon the algorithm described by Kanellakis and Smolka [63], as used in the 'Concurrency Workbench' process algebra tool [30]. An improved algorithm, described by Paige and Tarjan [94], would shorten the execution time even further from $\mathcal{O}(mn)$ to $\mathcal{O}(m \log n)$ for determining the normal form of an n-state DLTS having m transitions.

In addition to the advantage of reducing the size of the specification, state minimization has the added benefit that no two distinct states in a circuit behaviour's normal form are bisimilar. Two states in the original specification are bisimilar iff they correspond to the same state in its normal form. This greatly reduces the time complexity of many of the tests performed during the synthesis process

since testing for bisimulation equivalence reduces to testing for state equality. Having performed state minimization all of the test described in this chapter may be performed by simple local model checking. These tests may even be optimized in the compiler, allowing a number of properties to be tested simultaneously and hence reducing the number of times a given state is examined. For example, all of Udding's rules may be checked concurrently.

In the current delay insensitive circuit compiler implementation, each state of the state graph is initially tagged with the location in the source specification file at which it is defined. This allows diagnostic error messages, such as those caused by delay sensitivity, to be presented to the user indicating the erroneous point in the specification.

Chapter 3

Basic Components

One of the original difficulties in asynchronous design until recently has been the unavailability of appropriate components. The advent of semiconductor technology has made it easier to design asynchronous circuits. VLSI design tools have expanded the menu of components that a designer can use. If necessary, new primitives can be custom-designed.

3.1 Handshaking

Since there are time and energy costs associated with driving a transition onto a wire, it pays to use as few transitions as possible in asynchronous signalling conventions, commonly referred to as *handshaking.* It is clear that there must be at least one transition, or change of voltage, on a wire to signify the occurrence of an event. Hence consecutive signals or events may be indicated by alternating high-to-low and low-to-high transitions. This signalling scheme is variously called transition, two-phase or non-return-to-zero (NRZ) signalling. The major advantages of two phase handshaking are that it is as fast and as energy efficient as possible. However, additional logic and state information may be required in each element since logic devices tend to be sensitive to voltage levels or transitions in a particular direction.

The only real alternative to two-phase handshaking is a protocol first used by Muller and used in many of his examples of speed independent circuits. It is sometimes referred to as either Muller, four-phase or return-to-zero (RZ) signalling. The return to zero character of four-phase handshaking tends to result in very

simple and natural circuit implementations but uses twice as many transitions as transition signalling. Whenever wire delay is a substantial fraction of the operation time, the extra trip required by a single communication is a serious performance penalty.

In four-phase handshaking all wires are initially low, and after each message is sent are returned back to their initial low state. This is given the name 'four phase handshaking' owing to the fact that both transitions are accompanied by an additional acknowledgement, resulting in four phases for a complete message transfer. One principal advantage of this approach is it limits the number of wire state patterns that may occur with each wire pattern indicating a unique datum.

3.2 Primitive Components

One of the advantages of the described methodology is that it decomposes the delay insensitive specification into a finite basis of standard primitive components. By using standard primitive components instead of custom designing, the investment in finding an especially good design may be amortized over all its uses. This usually reduces design time and results in faster implementations.

In much the same way as conventional clocked circuits are composed out of standard primitive components such as AND and OR gates, speed independent and delay insensitive circuits are frequently composed purely of common elements. Owing to the constraints placed on this class of digital circuit it is not surprising that these components are not those normally used in conventional design. The operation of a few of the more common primitive building blocks is given in the sections below. A more complete list of such devices may be found in the literature [10,14,20,60,64,110].

Keller [64] determined a set of primitive modules that were "universal" with respect to a class of speed independent circuits. This is a similar result to the fact that the NAND gate and the NOR gate are both universal to synchronous designs, i.e. all such designs may be implemented entirely of such primitives. Keller showed that only three elements were required for four-phase speed independent circuits. These three elements were the Merge, the Select and the Arbitrating Test-and-Set elements.

3.2.1 Merge

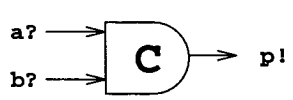
A merge produces an output transition for each transition on either input. This component is commonly realized with an XOR gate. The merge element in transition signalling is analogous to an OR gate in conventional level signalling. Merges are commonly used to connect several components to a shared bus in data path based systems. The use of an XOR gate places several constraints on the environment, since there is a possibility of transmission interference if both inputs change simultaneously.


```
bi Merge  a?.p!.Merge + b?.p!.Merge
```

Figure 3–1: Merge element

3.2.2 Muller C-element

The Muller C-element is a very useful device for the design of self-timed circuits. Its output goes low when all of its inputs are low and becomes high when all of its inputs are high, and otherwise the output stays in whatever condition it was. In transition signalling terms, a c-element produces an output transition after every pair of transitions on its inputs. Russian terminology refers to a c-element as a *hysteresis flip-flop* (Γ-flip-flop). Delay insensitive use of the c-element places the constraint on the environment that it may not send two transitions down an input without sending a transition to the other input and waiting to receive the output transition between them. This is a fundamental difference in the use of the c-element between delay insensitive and speed independent circuits.

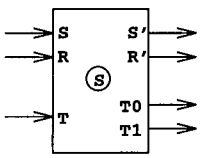



```
bi CElem  a?.b?.p!.CElem + b?.a?.p!.CElem
```

Figure 3–2: Muller c-element

3.2.3 Keller Select

The primary state holding element is the select element, or SR. Its state is set by a signal on its S input, reset by a signal on its R input and tested by a transition on its T input. The receipt of an S input is acknowledged by a S´ output, and a R request produces a R´ acknowledgement. When the state of the select element is tested, the two output signals T0 and T1 indicate the result of the test and acknowledge the T. For correct operation each input transition must be acknowledged by an output transition, before any further requests are sent.




```
bi Sel0  t?.t0!.Sel0 + r?.r´!.Sel0 + s?.s´!.Sel1
bi Sel1  t?.t1!.Sel1 + r?.r´!.Sel0 + s?.s´!.Sel1
```

Figure 3–3: Keller Select element

An instance of a select element may be used within a circuit in one of two ways. In the first, the environment always guarantees to send either a set or reset signal before the state of the select is tested. The other mode of operation relies on the initialization of the select element to a default state. Because the signals R, R´ and T0 are completely symmetric to S, S´ and T1 respectively, the default state is conventionally chosen to be reset. The implementation of the initialized selects is not covered by the design methodology, but may be achieved by the use of a global reset signal. Such reset signals are not usually displayed in schematic representation of delay insensitive circuits.

Occasionally, a Keller select element is used within a circuit where it is never reset. In such a case the select element is used as a '**switch**' component. All tests on the state of a switch element result in the acknowledgement T0 until the circuit receives its first set signal S, whereafter all further tests return T1. Hence the select element is only used as a simple switch, and never has a state transition from set to reset. This restriction on its behaviour results in more efficient implementations of switch elements than of generic select elements.

Select elements may be implemented using decision wait elements (as described in the next section). The ability to use decision waits to create select elements and

vice versa, allows either component to be used as a primitive basis component. In the delay insensitive circuit synthesis procedure described later, the select element is preferred as it is considered the principal state holding component and the Muller c-element the principal synchronization component. The decision wait element, performing both these roles, is considered as a composition of the two. However, in line with current delay insensitive research, many circuit synthesis steps are explained in terms of both select elements and decision waits.

The select element may be fabricated from a 2×3 decision-wait and four 2-input merge gates, as shown in figure 3–4 above. The test, set and reset signals form one dimension and the state of the select forms the other dimension. Two merges are used to combine the set and reset outputs into a pair of single set and reset acknowledge. The remaining merges are used to generate the feedback inputs for the next state.

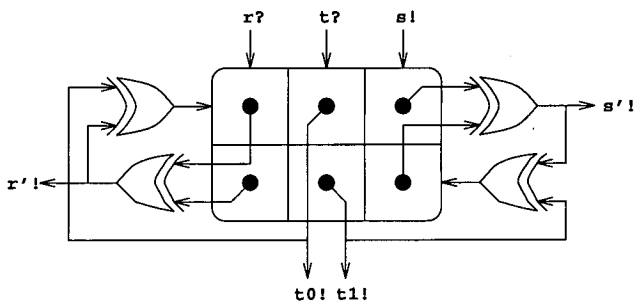
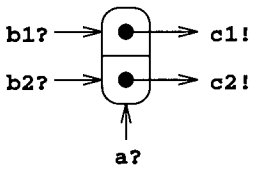


Figure 3–4: Select element implementation

3.2.4 Decision-wait elements

The circuit synthesis method makes use of either of two circuit bases at the lowest level of abstraction. The first circuit basis consists of the merge, c-element and Keller's select element. Although this is the smallest basis for the class of deterministic delay insensitive circuits considered, other researchers use the class of $M \times N$ decision-wait elements in place of the select element. In general any decision-wait element may be delay insensitively decomposed into a network of merges, c-elements (1×1 decision-waits), 2×1 decision-waits and 2×2 decision-waits. This collection of primitive components forms the second circuit basis of the synthesis method.

The schematic representation of a decision wait element is shown in Figure 3–5. Conventionally, two dimensional decision wait elements are depicted as a matrix of cells, with inputs along two perpendicular edges and the corresponding outputs being generated in the centre of each cell.




```
bi DW21  b1?.a?.c1!.DW21 + b2?.a?.c2!.DW21 +
         a?.(b1?.c1!.DW21 + b2?.c2!.DW21)
```

Figure 3–5: 2×1 decision wait

A 2×1 decision-wait is implemented by merging the set and reset acknowledges of a select element and combining the output and the column input with a c-element which is fed back into the test input of the select element. The select element and merge gate are often combined into a single call element (see section 3.3.3), if such a component exists as a standard cell.

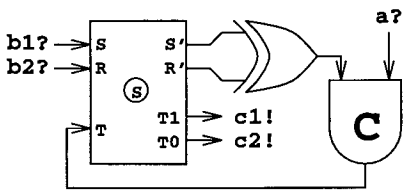
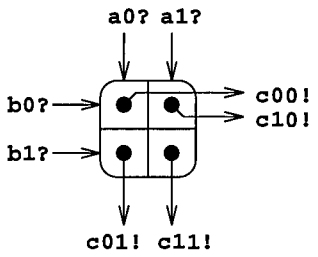


Figure 3–6: Decision wait implementation

In addition to the 2×1 decision wait element, the 2×2 decision wait element is required in order to implement arbitrary two (and higher) dimensional decision wait elements.

Unlike one dimensional decision waits, instances of higher dimension decision wait elements may have outputs that never transition. This is caused when a particular row/column combination is guaranteed never to occur. These outputs are termed '*unused*' or '*non-live*', while outputs that do transition are termed '*live*'. The non-live outputs of a decision wait are denoted in schematic diagrams by not marking the row/column position with a 'dot' and also omitting the output wire.




```
bi DW22  a0?.b0?.c00!.DW22 + a0?.b1?.c01!.DW22 +
         a1?.b0?.c10!.DW22 + a1?.b1?.c11!.DW22
```

Figure 3–7: 2×2 decision wait

In the one dimensional case, an $N \times 1$ decision wait with a non-live output can be replaced with an $(N-1) \times 1$ decision wait. This is because a non-live output can only be caused by a non-live input.

Transistor level implementations of small decision-wait circuits tend to be more efficient than the corresponding select element implementations. However, in the general case, the choice of circuit basis depends upon the specification being implemented.

3.2.5 RGDA Arbiter

The RGDA Arbiter is another primitive arbitrating component with a slightly more complicated communication protocol communication protocol. Each subenvironment of the arbitration has four signals, two inputs `r?` and `d?` and two outputs `g!` and `a!`. The protocol as observed by each subenvironment is a strict sequencing of input and output actions. The subenvironment initially requests to enter its critical region by sending a *request* `r?` which is eventually acknowledged by a *grant* output. To leave the critical region, the subenvironment sends a *done* signal `d?` which is confirmed by a final *acknowledge* output `a!`. This sequence of operations leads to the RGDA name of the arbiter.

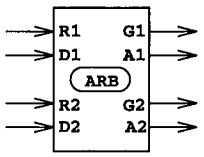


Figure 3–8: RGDA Arbiter

3.3 Standard Components

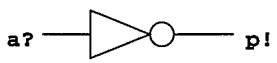
Although the design methodology decomposes all circuit specifications into the three 'atomic' primitive components described above, it also refers to common connection topologies of these components. These common components, or subcircuits, do not increase the generality of the existing primitives, but are frequently used by other asynchronous circuit researchers, and are used effectively as a convenient representation for the subcircuit they denote.

Although the primitive components are sufficient to fabricate any data communication class component, the actual implementations derived from them are not necessarily optimal in terms of speed, area or power consumption. The advantage of using standard macro components is that if efficient implementations of these circuits exist in a library of parts, they may be instantiated directly into the final design. This enables circuits to be fabricated from a minimal cell library of only three parts, and still take advantage of the benefits of larger cell libraries.

3.3.1 IWire

The IWire component is commonly used in delay insensitive circuits to set up initial conditions. An IWire (Initialized Wire) has a single input `a?` and a single output `b?`. It behaves by initially generating an output transition and subsequently generating an output in response to each input. By considering voltage levels, an IWire component can be seen as equivalent to an invertor or NOT gate in conventional digital design. As such it may be implemented using an XOR gate (primitive merge component) with one input connected to logic high. This then behaves as an invertor.

However, as described in section 3.6, the initialization of delay insensitive circuits poses a number of problems. Hence an IWire is often implemented as a merge component with one input connected to a 'hidden' initialization wire.

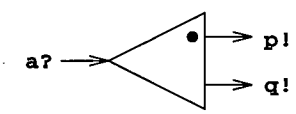



```
bi IWire  p!.a?.IWire
```

Figure 3–9: IWire component (Invertor)

3.3.2 Toggle

The toggle element is a commonly used component in delay insensitive circuit design. A toggle element has a single input `a?` and two outputs `p!` and `q!`. A toggle element is a state holding component that routes an input transition alternately to its two outputs. The first input transition generates the output `p!` and subsequent input transitions will generate alternating output signals. The schematic representation of a toggle element marks its *initial* output with a dot, as shown in figure 3–10 below. VLSI implementations of the toggle element typically require 'hidden' reset circuitry to initialize its internal state.




```
bi Toggle  a?.p!.a?.q!.Toggle
```

Figure 3–10: Toggle element

The toggle element is easily constructed from an initialized select, where the results of the test input are used to set or reset the select element into the opposite state. The set and reset acknowledge outputs of the select element then form the toggle outputs. Similarly, the toggle element may also be implemented by an initialized 2×1 decision wait element, where the two outputs are used to define the next (row) state and the column input forms the toggle's input.

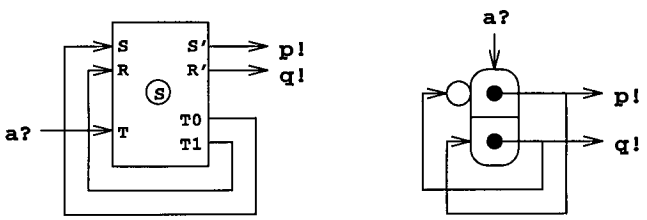
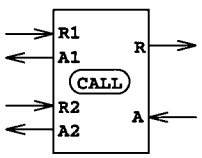


Figure 3–11: Toggle implementation

3.3.3 Call Element

A *call module* implements the hardware equivalent of a subroutine call. It combines two pairs of request/acknowledge signals, `R1?/A1!` and `R2?/A2!`, into a single pair `R!/A?`. This allows two mutually exclusive 'processes' to share a common communication channel. After a request from either `R1?` or `R2?`, the call component generates the output `R!`. When this request is acknowledged by the input `A?`, the call acknowledges the appropriate source; generating `A1!` if the original request was `R1?` or `A2!` if the original request was `R2?`. Much like a software subroutine call, the call module is responsible for remembering which of the two 'clients' made the request in order to acknowledge the correct 'client'.




```
bi Call  r1?.r!.a?.a1!.Call + r2?.r!.a?.a2!.Call
```

Figure 3–12: Call element

There are two common ways to implement a call module depending upon the choice of primitive basis. The first method requires a Keller select element and a merge gate, and the second method uses a 2×1 decision wait and a merge gate. The select implementation uses `R1?` and `R2?` to set and reset the select element respectively. The set and reset acknowledge signals are then merged to create the `R!` output. The returning `A?` input is then used to test the state of the select element, the `T0!` output is used to generate `A2!` and the `T1!` output generates `A1!`.

The alternative, and potentially faster, implementation is to use a 2×1 decision wait element. The `R1?` and `R2?` inputs are forked and merged to generate the

R! output. The other branches of the fork are fed into the rows on the decision wait. The acknowledge input A? is then used as the decision wait's column input, and the two decision wait outputs for the acknowledge signals A1! and A2!. This implementation is potentially faster than the select method described above, since the state setting/resetting is performed in parallel with the communication action on the shared channel.

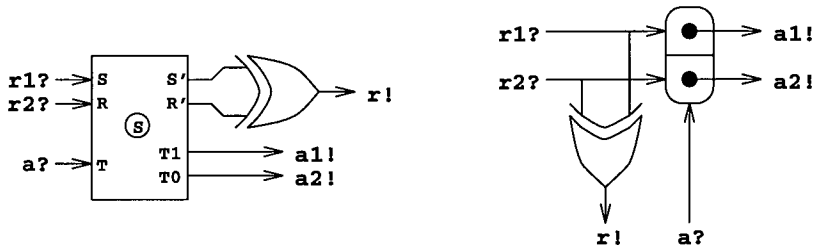


Figure 3–13: Call element implementation

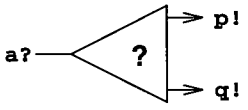
3.3.4 Conventional Logic Gates

The normal logic gates used in conventional logic synthesis (AND, OR, NAND and NOR) may also be used in delay insensitive circuits. Typically they are used as efficient implementations of merge elements (XOR gates) that have appropriate restrictions on the environment's behaviour. In addition to the invertors mentioned above, a subset of delay insensitive specifications will generate conventional logic implementations, which are guaranteed both race and hazard free. This feature of the synthesis process allows the use of multi-level logic optimization and technology mapping to existing technology libraries. However, extreme care must be taken to ensure that the delay insensitivity of the resulting circuit is not compromised by the logic minimization. Suitable minimization algorithms are described by several of the researchers mentioned in section 1.4.2.

Although the proposed circuit synthesis methodology does not use conventional logic gates during the synthesis steps, the peephole optimization and technology mapping phases (section 6.1.9) replace primitive delay insensitive components with conventional gates that have efficient transistor level implementations.

3.3.5 Choice Element

Another common component in delay insensitive circuit synthesis is the choice element that non-deterministically chooses between one of two possible output events. A choice element has a single input `a?` and two symmetric outputs `p!` and `q!`. After each transition on the input, the circuit generates a single transition on either of the two output terminals. This component is the canonical example of a 'statically non-deterministic' component. The schematic representation and CCS behaviour of the choice are given in figure 3–14.




```
bi Choice  a?.(p!.Choice + q!.Choice)
```

Figure 3–14: Choice element

Depending upon the notion of implementation, it may be permissible to implement a choice element with a deterministic component such as a toggle element or even a single wire connecting the input to one of the outputs. One possible implementation is to fork a signal and feed the two branches into an arbiter to determine which arrived first, relying on our non-deterministic model of delays. However, once fabricated such an implementation would typically make the same decision under identical operating conditions. The problem is that there no notion of fairness or probability attached to the generation of either output. It may sometimes desirable to make a truly random 'fair' choice between outputs, but this cannot be expressed in the current formalism.

General N-output choice components can be implemented by simply generating a binary tree of $N-1$ two-output choice elements. If the probabilities of each output on the primitive choice are known, the choice tree may be skewed/balanced to improve performance. It is also possible to generate an N-output choice component with equal probability outputs from individual choice elements with equal probability outputs. The technique is to use a statistical 'test and reject' method. A balanced 2^k-output choice tree is constructed, where $k = \lceil \log_2 N \rceil$, which generates outputs with equally probability. N of the outputs are then nominated as the choice's outputs and the other $2^k - N$ outputs are then combined in a merge tree and merged with the input to produce the root signal of the choice tree. Hence,

if one of these outputs is ever produced, it is 'rejected' and the signal used to generate another possible output. Note, this construction only makes sense if the 'fairness' properties of the individual components are known.

3.4 Generalized Components

Throughout the description of the circuit synthesis process, the design methodology assumes the existence of several classes of circuits that are generalizations of the standard primitive components. Typically these generalized delay insensitive components are abstractions of primitive components with an arbitrary number of inputs or outputs. These macrocomponents are parameterized on the number of inputs or outputs and instantiated during the synthesis process. These circuits, and methods of decomposing them into delay insensitive networks of primitive components, are introduced in this section.

One advantage of this approach is that efficient implementations can often be developed for a given size macrocomponent which may then be added to a designer's parts library. When an asynchronous circuit compiler requires this particular size of component, the optimized version may be taken from the cell library. However, if no such custom design exists, the generalized component may be implemented using standard primitive components.

3.4.1 N-Input C-Elements

The simplest form of generalized component are the N-input merge and the N-input c-element. The semantics of the N-input c-element is to generate an output only after all of its inputs have arrived, and those of the N-input merge element are to generate an output after any of its inputs arrive. The usual restrictions on input event occurrences on these parts follow from their primitive counterparts. Both of these macro components may be synthesized by constructing a binary tree of $N-1$ primitive 2-input components.

A typical standard cell library may contain efficient implementations for some of these components for $N > 2$, that may be used in the decomposition of other macro-cells. For example, a very efficient CMOS transistor implementation of a 3-input c-element exists. In order to minimize the total number of standard cells

in such trees, the maximum number of input signals should be combined to form a single intermediate signal and the process applied recursively. This technology mapping decomposes a 6-input c-element (using the cells described above) into two 3-input elements feeding their results into a single 2-input element. This requires fewer cells than the three 2-input elements into a 3-input element decomposition.

For the N-input c-element, the synchronization tree may be biased on the arrival times of each input signal. Those inputs that arrive first may be combined lower in the tree, with the intermediate value being generated one c-element transmission delay after the later of the two signals. This method may be used in completion detectors of multiplier circuits where the least significant bits are typically generated before the most significant bits.

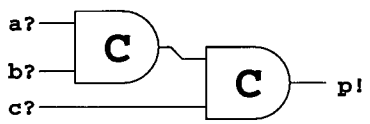


Figure 3–15: 3-input c-element

3.4.2 N-Input Merges

An N-input merge is simply an abstraction of the merge gate to have N inputs. The trivial cases where $N = 1$ and $N = 2$ are implemented by a wire and a standard merge gate respectively. For the remaining cases, an N-input merge gate, $N \geq 3$, is implemented by a balanced binary tree of $N - 1$ two input merge gates.

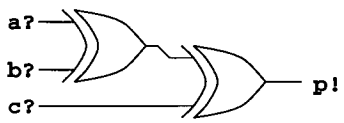


Figure 3–16: 3-input merge component

If the transition time for a single merge is $\tilde{x}$, the minimum propagation delay through a balanced N-input merge is $k\tilde{x}$, where $k = \lfloor \log_2 N \rfloor$, and the maximum delay is $(k+1)\tilde{x}$. The typical case delay (assuming all inputs are equally likely) is given by the expression $((k+2)N - 2^{k+1})/N$.

The average case performance of the N-input merge can occasionally be improved by using *Huffman encoding.* If the relative frequencies of each of the merges inputs are known (or can be approximated) the balance of the merge tree may be biased by the frequency of arrival of each input. Using this technique, frequent inputs pass through less levels of the tree than infrequent signals improving the average case performance. Huffman encoding first combines the two most infrequent input signals to generate an intermediate signal with their cumulative frequency. This step is repeated until only a single output remains. Given exactly equal input frequencies, the process prefers to combine the shallowest tree which results in balanced binary trees of merge gates. For Udding $\mathbf{C}_1$ synchronization class delay insensitive circuits, input signal frequency can be determined at 'compile' time. For the remaining data dependent classes of circuit behaviour, frequency figures can be determined by circuit simulation or profiling using typical environment behaviour. A circuit designer can also provide 'compiler' hints on which signals are typical and which are exceptions that rarely occur.

If Huffman encoding is used, an unbalanced or skewed tree may actually increase the worst case delay path through the network. However, the typical case path is reduced such that the *average* case performance is improved. This is an example of the difference between self-timed and synchronous design; self-timed design may improve a circuit's overall performance at the expense of the worst case delay.

3.4.3 Generalized Call Components

A general call component combines $N \geq 1$ mutually exclusive pairs of request and acknowledge signals into a single pair. The trivial case when $N = 1$ is simply implemented by using the only input request/acknowledge pair as the required multiplexed signals, and when $N = 2$ is implemented by a standard Call element. For $N \geq 3$, a call element is decomposed by continually multiplexing pairs of request/acknowledge signals using standard call elements. Each call element will combine two handshaking pairs, that will need to be combined by an $N - 1$ 'input' call, hence an N 'input' general call component can decomposed into $N - 1$ standard 2 'input' call elements.

In order to reduce the average period/response time of the general call component, the call elements are typically organized as a binary tree. However, if

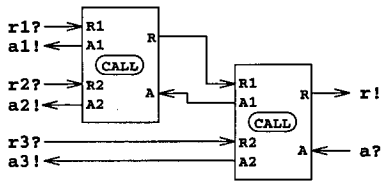


Figure 3–17: general call component

the frequency of each of the inputs is known *Huffman encoding* may be used to improve the typical performance as described previously.

3.4.4 Generalized Select Element

A general select element is a Keller select element that has an arbitrary number of set or reset signals (and their corresponding set and reset acknowledge signals). This allows the compiler to describe a state holding device whose state may be modified independently by any number of external inputs, provided that these requests are mutually exclusive.

For the trivial case, when the select element has no set inputs, a general select element is implemented by a wire connecting the test input `T?` to the `T0!` output (using the convention that a Keller select element is initially in its reset state). If the select element has any reset inputs, each of these is wired directly to their corresponding reset acknowledge outputs. Notice, that because the trivial case is not state holding, it requires no 'hidden' initialization circuitry at the transistor level.

A general select element with a single set input and a single reset input is a standard Keller select element, and one with a single set input and no reset inputs is the 'switch' component described in section 3.2.3 on page 54.

The standard constructions for decomposing a general select element is to use a general call element to combine together all the set (or reset) signals to produce a single set (or reset) request which is fed into the single Keller select (or switch component) that holds the state. The set (or reset) acknowledge signal generated by the component is then fed into the single acknowledge input of the general call element, which in turn produces the appropriate individual set (or reset) signals for each input. If the general select element has no reset inputs, a single general call component is used to combine the set/set acknowledge pairs which feed a single

switch component. Otherwise two general call elements are used to combine the set/set acknowledge pairs and reset/reset acknowledge pairs which are the connected to a Keller select element's set/set acknowledge and reset/reset acknowledge terminals respectively. This single Keller select element needs to be initialized to the reset state, if the general select can be tested before it is set or reset.

The frequency with which each input is used to set or reset a general select element can be used to optimize the decomposition of the general call element (see section 3.4.3).

3.4.5 $N\times1$ Decision Wait Elements

The trivial case for an $N\times1$ decision wait applies when $N = 1$, in which case the 1×1 decision wait element may be implemented by a single Muller c-element. When $N = 2$, the standard implementations of a 2×1 decision wait may be used (see section 3.2.4).

For $N \geq 3$, an $N\times1$ decision wait elements may be implemented using an N-input call module and a single Muller c-element. As described in the previous section, the N-input call module may be decomposed into $N-1$ merge gates and either $N-1$ 2×1 decision waits or $N-1$ Keller select elements. The construction is to multiplex the N row inputs using the call component to produce a single signal which is synchronized with the single column input using the Muller c-element. The output of this c-element is then used to acknowledge the call and hence generate the appropriate output depending upon which of the N row inputs arrived. Initialized decision wait elements may be constructed using an asymmetric c-element and an initialized general call component. The asymmetric c-element allows the first synchronization to require either only a row or a column input, and the initialized call tree allows the correct initial output to be generated if only the column input is required.

3.4.6 $N\times M$ Decision Wait Elements

The decomposition of two dimensional decision wait elements is far more complex than the decomposition of the one dimensional $N\times1$ decision wait. The two sections below describe the decomposition of $N\times M$ decision waits, where $N \geq 2$ and $M \geq 2$, using a decision wait and select element bases respectively.

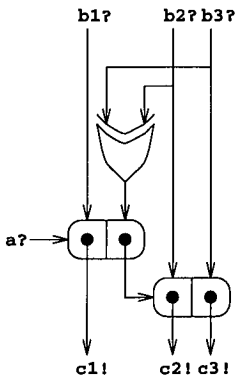


Figure 3–18: 3×1 decision wait component

Decision Wait Method

An efficient decomposition of two-dimensional decision waits into merges and 2×2 decision wait elements is given by Patra and Fussell [97]. Their method recursively subdivides the decision wait into four quadrants. A central 2×2 decision wait element is then used to determine which quadrant contains the output transition and then directs the relevant input transitions to that quadrant. For a complete exposition of this technique, the reader is referred to the original paper [97].

Patra and Fussell's method is an improvement over Mark Joseph's that reduces the number of merge gates by reusing common subexpressions (as described in section 6.1.3). This both improves performance (as detailed in the paper) and reduces hardware. Using this technique an $N \times M$ decision wait may be implemented using $(N-1)(M-1)$ 2×2 decision wait elements and $N+M-4$ 2-input merge gates.

Figure 3–19 below demonstrates the result of this decomposition strategy on a 4×4 decision wait. This implementation requires a total of nine 2×2 decision wait elements and four merge gates.

From this example, it is clear that hardware implementations of large two-dimensional decision waits require a very large number of transistors. Luckily, it is easy to implement small decision waits directly as 'macro components' using speed independent techniques. However, decomposition methods are still required for decision waits larger than equipotential regions.

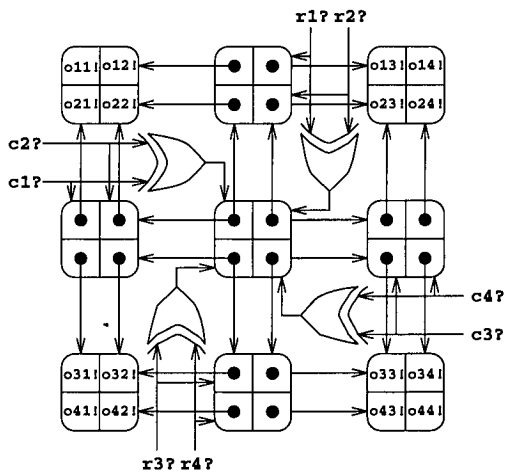


Figure 3–19: 4×4 decision wait decomposition

Select Element Method

An $N \times M$ decision wait can also be implemented using the Keller select element basis. The select element decomposition strategy effectively constructs a two-level binary tree of select elements. The first (or top) level of the tree implements the N rows of the decision wait and the second (or lower) level implements the M columns. The top level of the tree contains $N-1$ select elements and $N-1$ merge gates organized as an $N \times 1$ decision wait element, as described in the previous section. Each of the N outputs of this tree (one-dimensional decision wait) are fed into 'root' of one of N identical select trees of $M-1$ select elements. Each of these trees records which of the M columns of the decision wait transitioned. The organization of these $N(M-1)$ decision waits may be understood by considering the implementation of an M-input call element, but where each select element is repeated N times. These repeated select elements are formed by sequentially connecting the `R´` and `S´` outputs of one select element to `R` and `S` inputs of the next. This 'minimal' hardware arrangement requires an additional $M-1$ merge gates (though faster organizations requiring additional hardware are described in section 6.1.8). The root output of the 'wide call' element is used as the single column input to the top level one-dimensional 'decision wait'.

This implementation strategy requires a total of $NM-1$ select elements, $M+N-2$ merge gates and a single 2-input c-element. This can be considered 'optimal' as the c-element is required for synchronization, and the $NM-1$ select elements are required to generate NM distinct outputs from a single synchronization signal.

The merge gates are necessary to preserve connection parity, i.e. the appropriate number of external inputs and outputs.

3.4.7 N-TOGGLE Circuits

An N-toggle is the generalization of a toggle element to multiple outputs. An N-toggle, sometimes referred to as a 'Johnson counter', has a single input $a?$ and N outputs, labelled $p_0!, p_1!, \ldots p_{N-1}!$. After each transition on the input, the N-toggle cyclically produces a transition on a successive output. After the initial input transition, it produces the output $p_0!$, after the next input $p_1!$ and so on. After N input excitations, the cycle starts again from the beginning with the output $p_0!$. The environment may only send another input upon receipt of the $p_i!$ output caused by a previous input.

The trivial cases for constructing an N-toggle from primitive components are the 1-toggle, which is implemented by a wire, and the 2-toggle which is implemented by a standard toggle component. For $N \geq 3$, there are several possible decomposition strategies for generating N-toggles. The following section present several methods using decision-waits, toggles, select elements and hybrid methods respectively.

Decision Wait Method

Perhaps the easiest way to implement an N-toggle is to use a single $N \times 1$ decision-wait element. The approach uses the decision wait to synchronize the N-toggles input, on the single row input, with one of the N state inputs, one per column. On receipt of both the N-toggle's input and the appropriate state signal, the decision wait produces one of N outputs. This signal is forked, with one branch used to generate the $p_i!$ output of the toggle and the other branch is used to indicate the next state to the decision wait. The output of the last state in the sequence is used to reset the sequence to its start state. The decision wait element is initialized with an input to place it in the N-toggles initial state. This implementation is insensitive to differing delays of the branches in the output forks. The next input will only be sent after output transitions have been received by the circuit's environment. Hence even if there is a large delay in the state feedback wires, the decision wait will wait until both the state signal and input signal have arrived before generating

an output. An example implementation of a 3-toggle using this approach is shown in figure 3–20.

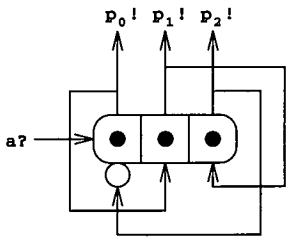


Figure 3–20: Decision wait 3-toggle circuit

Toggle Method

An alternative implementation method is to decompose an N-toggle into several toggle components and merge gates. When $N = 2^k$ for any integer k, $k \geq 2$, an N-toggle may be implemented by a balanced k-level binary tree of toggle components. The N-toggle's input is fed into the input of the toggle at the base of the tree, the outputs of which are used to fed the inputs of two 2^{k-1}-toggles that form the left and right subtrees. This decomposition is applied recursively until $k = 1$, which is implemented using a standard toggle component. The outputs of these $N/2$ toggles at the leaves of the tree, form the outputs of the N-toggle. Using this decomposition an 2^k-toggle may be constructed using $2^k - 1$ toggles. The construction is demonstrated by the 4-toggle example, given in figure 3–21 below.

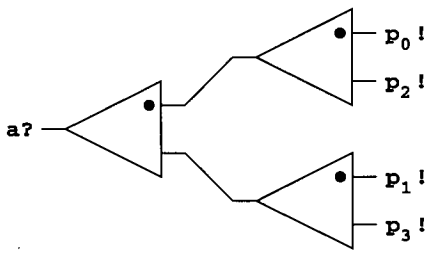


Figure 3–21: 4-Toggle circuit

Some care must be taken to associate each output with its position in the toggle's sequence. The subtree fed by the root toggle's initial output generates all even number outputs ($p_0!, p_2!, p_4! \ldots$) and its other subtree the odd number outputs

$(p_1!, p_3! \ldots)$. One method of assigning the correct outputs is to associate a pair of numbers, representing the 'start' and the 'step' respectively, to each toggle in the N-toggle. The root toggle is assigned the value $(0, 1)$. If an internal toggle has the value (u, v), then the toggle at its initial output is assigned the value $(u, 2v)$ and the toggle at its other output is given the value $(u + v, 2v)$. For a leaf toggle with the value (u, v), its initial output generates the output $p_u!$ and its other output generates the signal $p_{u+v}!$. A delay insensitive circuit compiler can implement this technique efficiently using binary representations and bit vectors.

For the cases where $N \neq 2^k$, the required behaviour is achieved by creating an M-toggle, where $M = 2^{\lceil \log_2 N \rceil}$. The surplus $M - N$ outputs are then merged together and combined with the input signal a?. This has the effect that if one of the 'surplus' output is reached in the M-toggle sequence, that signal is not output but used to advance the toggle to the next state. By combining $M - N$ outputs, the circuit generates a cyclic sequence of N signals, an N-toggle. This decomposition requires $M - 1$ toggles and a single $(M - N) + 1$-input merge (typically decomposed into $M - N$ merge gates). Using the Huffman coding optimization, the a? input appears at the very top of the merge tree because it occurs more frequently than any of the feedback signals. This significantly improves the performance of the decomposition. Figure 3–22 below demonstrates this construction applied to a 3-toggle circuit. Note that the choice of which outputs to merge is arbitrary, but may have an effect on the worst case response time of the toggle. Hence, combining consecutive outputs will cause several iterations of the M-toggle before an output is generated.

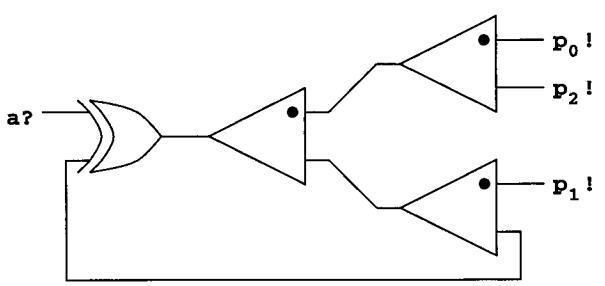


Figure 3–22: 3-Toggle circuit

Performance analysis of the above decompositions is straightforward. Let $k = \lceil \log_2 N \rceil$. For the case $N = 2^k$, the period of the N-toggle is given by $Nk\tilde{t}$ and the response time is $k\tilde{t}$, where $\tilde{t}$ is the transition time for a single toggle. When

$N \neq 2^k$, the minimum response time is $k\tilde{t} + \tilde{x}$ where $\tilde{x}$ is the transition delay of a single merge gate. Provided that consecutive outputs are not used as feedback signals, the worst case response time is given by $2k\tilde{t}+(\lceil \log_2(2^k - N)\rceil + 2)\tilde{x}$, i.e. the worst case delay through the merge tree plus twice the delay through the 2^k-toggle. Determining the period, and therefore the average response time, is slightly more complex. The period of the 2^k-toggle is $2^k k\tilde{t}$, the merge gate at the toggle input has a period of $2^k\tilde{x}$ and the merge tree for the $M = 2^k - N$ feedback signals has a period of $((M-2)\lfloor M \rfloor + 2)\tilde{x}$ (using the result given in section 3.4.2). This merge tree is balanced by Huffman coding because all inputs occur with equal frequency. This gives the total period of a N-toggle by the following equation.

$$2^k k\tilde{t} + ((2^k - N - 2)\lfloor 2^k - N \rfloor + 2^k + 2)\tilde{x}$$

Select Element Method

An alternative approach is to implement an N-toggle using Keller select elements. It is more efficient to implement an N-toggle directly rather than using the toggle decomposition method and then implementing each toggle with a select element. The principal reason is the use of feedback signals, in the $N \neq 2^k$ decomposition method, to skip surplus outputs. This tests and toggles the state of components that can be avoided by directly implementing the N-toggle using selects. Another reason for the inefficiency is that every tested select element changes its state, i.e. there are at least $\lfloor \log_2 N \rfloor$ state changes after each input transition.

The basic arrangement of select elements in an N-toggle is to create a balanced binary tree of $N - 1$ select elements. The `T0!` and `T1!` outputs of each select element are used to generate the `T?` input on the next level of the select tree. The N 'intermediate' signals at the leaves of this select tree are then used to modify the state of the appropriate selects before generating the N-toggle's outputs. A binary vector may be used to represent each state by concatenating the states of each select tested for a given input. For a perfectly balanced tree when N is a power of two, all state vectors have the same length $\log_2 N$. Changing state from a state represented by the vector $\tilde{v_i}$ to the state vector $\tilde{v_{i+1}}$, requires setting all the selects that are not part of their common prefix. Hence a state assignment that maximizes the shared prefix length and only sets and resets each select once in the N-toggle's sequence (in order to avoid additional components) is required.

Optimal state assignments are achieved by assigning consecutive states to one subtree followed by consecutive states to the remaining subtree. Keller select elements at the leaf nodes should encode two consecutive states. This ensures that each select is only set and reset once in a sequence and that prefixes are maximized. When N is a power of two, the binary tree is perfectly balanced and all binary state vectors have the same length; $\log_2 N$ bits. For this case, using the binary representation of each signal's position in the sequence is an optimal state encoding. If such an optimal state encoding is used, an N-toggle is implemented using $N-1$ select elements. Each select is set and reset once, hence the period of an N-toggle is $2N-2$ state changes and $(k+2)N - 2^{k+1}$ state tests, where $k = \lfloor \log_2 N \rfloor$. This has the same $\mathcal{O}(\log_2 N)$ state tests per input transition as the toggle decomposition, but only $\mathcal{O}(1)$ average state changes per input transition.

The hidden reset circuitry may be reduced by noting that only the selects on the path encoding the initial state must be initialized. All the remaining selects elements are either set or reset before they are tested. If this optimization is used, the 'initial path' may be made shorter at the expense of performance. When $N \neq 2^k$, the state encoding should assign one of the short binary state codes to the initial state.

An example implementation of a 3-toggle using this decomposition method is given in figure 3–23. This 3-toggle is formed using only two select elements, only one of which must be initialized. The three internal states (outputs) are given the binary state vectors 0, 10 and 11 respectively. This circuit has a period of 4 state modifications and 5 state tests.

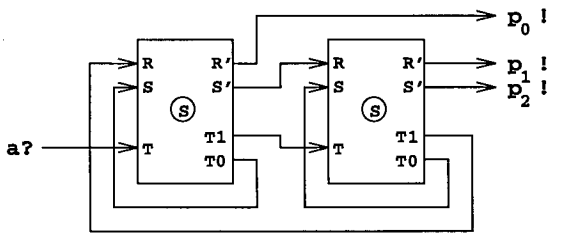


Figure 3–23: 3-toggle select decomposition

Hybrid Method

Another approach to decomposing N-toggles is to use a combination of the above techniques. When N is composite (i.e. not prime), an N-toggle may be decomposed into several generalized toggles components of lower order. For example, an nm-toggle (for positive integer constants n and m) may be decomposed into an n-toggle and n m-toggle components. The input signal is fed into the n-toggle, and each of its n outputs is fed directly into one of the m-toggles. This circuit will then cycle through the nm outputs of these m-toggles. The required n-toggle and m-toggle components can be implemented by any of the implementation methods described in the previous sections.

3.5 Arbitration Protocols

3.5.1 Mutual Exclusion Element

A common component used in asynchronous circuit design is the mutual exclusion element (ME). This is a minimal arbitration device with just one input `r?` and one output `a!` per subenvironment. The complete RGDA protocol is multiplexed onto these this pair of handshaking signals. A request to enter a critical region is initially sent on `r?` and granted on `a!`. Similarly, an indication to leave a critical region is also sent on `r?` and acknowledged on the same output `a!`.

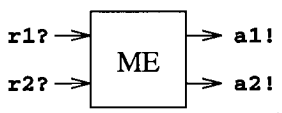


Figure 3–24: Mutual exclusion element

Figure 3–25 below shows how to convert an ME arbitration protocol into an RGDA arbitration protocol and vice versa. This allows the use of either of these components to be used as the primitive arbitrating component in a circuit basis. In practice this decision is based upon transistor level implementation details.

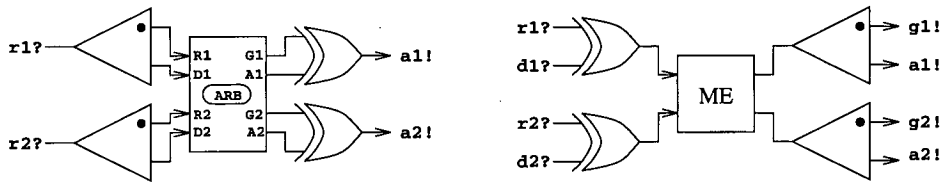


Figure 3–25: ME to RGDA and RGDA to ME protocol conversion

3.5.2 RGD Arbiter

A performance improvement to the RGDA arbiter results in an RGD arbiter. There is no reason for the subenvironment to have to wait for the final acknowledge in the above protocol. Once a subenvironment sends the `d?` to indicate it has left its critical region, the environment can immediately begin performing is non-critical actions. Subsequent requests are simply delayed until the arbiter receives a done `d?` signal indicating that the subenvironment has left the critical region. It should be noted that due to the delays in the wires connecting an RGD arbiter to its environment, a request to re-enter the critical region may be received before the signal indicating that that subenvironment has left the critical region. In this case, the arbiter waits until the appropriate done signal is received and then arbitrates between the re-request and any other requests that may be outstanding, acknowledging the winner as usual.

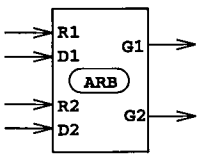


Figure 3–26: RGD arbiter

Figure 3–27a below shows how an RGD arbiter may be implemented using an RGDA arbiter and asymmetric Muller c-elements, and figure 3–27b shows how an RGDA arbiter may be constructed from an RGD arbiter. This equivalence of arbitration protocols allows either component to be used as the arbitrating member of a universal circuit basis.

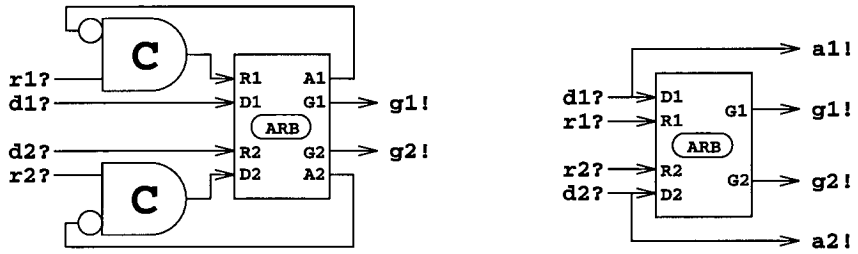
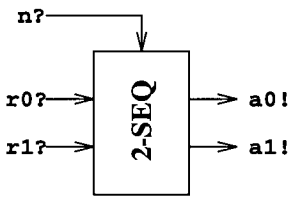


Figure 3–27: RGD to RGDA and RGDA to RGD protocol conversion

3.5.3 k-SEQ Component

Another Arbitration protocol examined by delay insensitive circuit researchers is Ebergen's class of k-SEQ components. These components have a single enable input `b?` and k pairs of handshaking signals $\mathtt{r}_i\mathtt{?}$ and $\mathtt{a}_i\mathtt{!}$, for $0 \leq i \leq k-1$. The circuit receives an arbitrary number of concurrent `r?` request signals and the enable input and generates a single acknowledge `a!` for one of the received request signals. For each subsequent enable input one of outstanding requests is granted. If there are no outstanding requests, the k-SEQ waits for one to arrive.




```
bi SEQ0  n?.(r0?.(a0!.SEQ0 + r1?.SEQ1) + r1?.(a1!.SEQ0 + r0?.SEQ1))
bi SEQ1  a0!.n?.(r0?.SEQ1 + a1!.SEQ0) + a1!.n?.(r1?.SEQ1 + a0!.SEQ0)
```

Figure 3–28: 2-SEQ component

The k-SEQ component operates much like a $k\times1$-decision wait element with concurrent row inputs. In fact, a $k\times1$-decision wait may be implemented by a k-SEQ component. An alternative form of this component is the initialized k-SEQ component which behaves as though an enabling `b?` input has initially be received. Schematically this is denoted by drawing a circle at the enable input of the k-SEQ symbol.

Much like the decomposition method given for decomposing $N\times1$-decision wait elements into $N-1$ 2×1-decision wait elements, Ebergen showed in section 6.2.6 of his doctoral thesis how to decompose a k-SEQ component into 2-SEQ components

and merge gates. Ebergen's initial analysis stated that the decomposition required less than $2k$ merge gates and $2k$ 2-SEQ components, but noted that several of the 2-SEQ components could be replaced/implemented by CAL (2×1-decision wait) components. A more detailed study shows that his decomposition requires $2N-4$ merge gates, $N-2$ initialized 2-SEQ components, $N-2$ 2×1-decision wait elements and a single standard 2-SEQ component. The decomposition is to combine two pairs of request/acknowledge signals into a single handshaking pair using the construction shown in figure 3–29. This method is used to construct a binary tree, which once again may be Huffman encoded to improve average case performance.

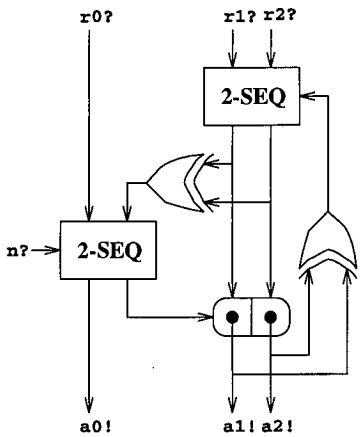


Figure 3–29: 3-SEQ decomposition

Ebergen also gave a methods for implementing a k-SEQ component using a k-input RGDA arbiter and a k×1-decision wait element and also for implementing a k-input RGDA arbiter using an initialized k-SEQ component and a k-input merge component in section 7.5 of his doctoral thesis. This equivalence means that Ebergen's 2-SEQ component could be used as the arbitrating element in a universal component basis.

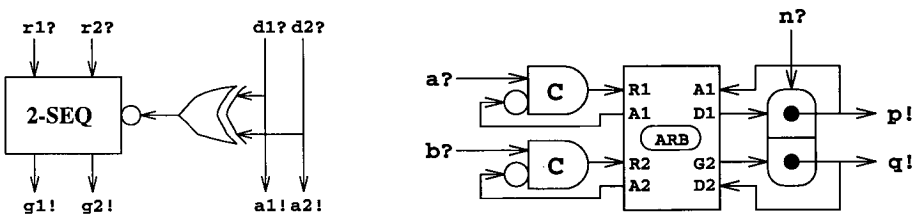


Figure 3–30: SEQ to RGDA and RGDA to SEQ protocol conversion

3.6 Initial Conditions

The question arises of the semantics of circuit specifications where the initial state of the signal transition graph is unstable, i.e. $s_0 \neq v_i$. It is assumed by convention that all wires in the circuit initially have a low voltage. The problem occurs when the initial actions in a circuit's behaviour state that some of these signals should rise. In real VLSI implementations it is far from clear at which point an asynchronous circuit begins operating. There are three potential solutions to this problem.

The first solution is to ignore the initial output actions and start the circuit with all wires low in the initial stable state, v_i. Similarly the DI synthesis system could enforce that the initial state of the circuit specification is stable.

The second approach is to implement a 'hidden global reset' input signal, `_Rst?`, that must be received before the circuit becomes operational. Upon receipt of this signal, transitions are sent on all output wires that occur in the maximal output event sequence from s_0 and the circuit enters the initial stable state v_i. This approach may naïvely be implemented by merging the appropriate outputs from the synthesized circuit with the global reset signal to generate the actual outputs. If synthesized circuit does not contain an output corresponding to a signal a in the maximal output sequence from s_0, the signal `_Rst?` may be wired directly to the output port a. Details of how to optimize the use of merge gates in delay insensitive circuits are discussed later.

The final interpretation to the semantics of initial output communications is to treat the wires involved as being initially logic high, instead of the conventional logic low. The naïve approach to such an implementation appends invertors to all the appropriate outputs of the synthesized circuits. If such an output does not exist, for a communication action in the initial maximal output sequence, this may be fabricated by directly connecting the output to a logic high voltage source. For use with restricted standard cell libraries, invertors may be implemented by 'merging' the required signal with a logic high source.

Each of the above approaches has its merits and disadvantages. The first uses less hardware than the remaining solutions. The choice between the last two options depends primarily on the intended environment for the circuit. If an output is intended to be connected to a transition based component such as a

select element, the second solution must be used; however, if the output is to be connected to a level-oriented component such as a C-element or standard logic gates, the final solution requires fewer transistors and pins of VLSI package. A typical implementation may use different approaches to the solution for each output in the initial maximal output event sequence as required.

3.7 Asynchronous VLSI Circuits

The implementation of asynchronous digital circuits is fraught with hidden complexity. This difficulty is caused by the design abstractions traditionally used by conventional circuit compilers. The ubiquitous use of clocks in digital circuits over the past 40 years has resulted in a number of assumptions that are taken for granted by current day designers and engineers. Not all of these implicit assumptions are well founded for asynchronous design styles. Hence, the implementation of digital circuits without the use of a global clock signal can often lead to anomalous and potentially hazardous behaviour. To date, several asynchronous VLSI circuits have been shown to be faulty when fabricated [12,77]. However, by fully understanding the underlying assumptions and abstractions used in the design process, it is possible to produce correct and reliable asynchronous circuits.

The principal pitfall with implementing asynchronous circuits is that the classical notions of delay and '*digital abstraction*' may no longer apply. Electronic circuits consist of components that manipulate voltages, current and capacitances as continuous, real-valued functions of time. To reason about such systems formally, circuit designers create abstract discrete models of this analogue behaviour. By using a clock signal, time is quantized into discrete intervals and voltages quantized into digital 'on' and 'off' values. This allows transistors to be idealized as switches, and switches idealized as boolean functions. These abstractions enable the physical semiconductors underlying a VLSI circuit to be viewed as a discrete mechanism capable of performing computation. The mathematical theory of finite state machines (FSMs) may then be used to model their behaviour.

In conventional MOS logic design, voltage values less than about 1.5 volts are considered to be 'false' or 'low', and voltages above about 3.5 volts to be 'true' or 'high'. In these regions MOS transistors operate as perfect *restoring* switches. These switches can then be organized to form boolean functions (combinational

logic) and state holding storage elements. In conventional synchronous design, finite state machines are implemented by combinational logic, evaluating the output and next state functions, and storage elements (flip-flops) that decouple the next state outputs from the current state inputs. On each global clock tick, the flip-flops latch the values of the next state outputs and present them as the current state inputs to the combinational logic. Hence each clock tick performs a FSM state transition. Using this universal model, it is possible to implement any computation given physical constraints on the maximum number of states.

This design style has several fundamental properties that enable the discrete abstraction to apply. The first is the 'voltage restoring' nature of the combinational logic and the latches.

The dynamic behaviour of a MOS VLSI circuit can accurately be described by a set of non-linear ordinary differential equations (ODEs). This basic equations are formed from Kirchoff's laws and *branch* equations describing the behaviours of individual devices. These equations represent the relations between voltages, currents, charges and fluxes and are often highly non-linear. This system of equations may be presented as a dynamical system in the following form:

$$\frac{\partial q}{\partial t}(v(t)) = -f(v(t), u(t)) \quad v(0) = V$$

where $v(t) \in \mathbb{R}^{n_1}$ is an n_1-dimensional vector of the unknown voltages at the internal nodes of the circuit and $u(t) \in \mathbb{R}^{n_2}$ is an n_2-dimensional vector of the known voltages at the circuit's inputs (both parameterized as a function of time $t \in \mathbb{R}^+$), q is the sum of charges due to the capacitors connected to a node, f is the sum of the currents charging each node.

$$\begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ q_n \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & C_{22} & \cdots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nn} \end{pmatrix} \begin{pmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{pmatrix}$$

The function f determines the current at each node, which may be decomposed using Kirchoff's current law to the sum of currents through each device connected to a particular node. The current through a particular device is then determined using an appropriate model of its behaviour. For example, the equations describe an ideal (Shichman and Hodges) model of a MOS N-type transistor [127].

Cutoff $V_{gs} - V_t \leq 0$

$$i_{ds} = 0$$

Linear $0 < V_{ds} < V_{gs} - V_t$

$$i_{DS} = -\beta \left((V_{gs} - V_t)V_{ds} - \frac{V_{ds}^2}{2} \right)$$

Saturation $0 < V_{gs} - V_t < V_{ds}$

$$i_{DS} = -\frac{\beta}{2}(V_{gs} - V_t)^2$$

where I_{ds} is the drain-to-source current, V_{gs} is the gate-to-source voltage, V_t is the device threshold and β is the MOS transistor gain factor, which is given by $\mu C_{tox} \frac{W}{L}$ where μ is electron mobility, C_{tox} is the thin oxide layer capacitance and W and L are the width and length of the device respectively.

3.8 Transistor-level Implementation

Each of the primitive components that form the basis of the described delay insensitive design methodology must itself be implemented in terms of logic gates or transistors. It has been shown that it is impossible to implement these delay insensitive primitives in a strictly delay insensitive way. Proofs of this result, showing that the class of entirely DI circuits is extremely restricted have been published by both Martin [79] and by Brzozowski and Ebergen [16,17]. To overcome this limitation different design methodologies distinguish themselves in their choice of compromises to pure delay insensitivity.

The design methodology presented in this document partitions the design and implementation of asynchronous circuits in two stages. The first decomposes the circuit specification in a network of primitive delay insensitive components. At this level of abstraction the correct operation is independent of the delays in the primitive operators and in wires connecting these operators (except that the delays be finite). This level of design is able to take advantage of the composition properties of self-timed circuits in order to reduce design complexity. This enables functionally equivalent modules or subcircuits to be interchanged without affecting the

operation of the circuit. The second stage then defines each of these primitive components as 'speed independent' network of transistors or logic gates. It is assumed that because the minimum number of primitive components is low, each of these operators is implemented as a standard or semi-custom cell.

This approach is advantageous for a number of reasons. The principal gain from splitting the methodology into two stages, is that global routing of wires between cells may be performed without regard to wire lengths without affecting the correctness of the circuit's behaviour. Obviously the circuit performance may be improved by shortening the lengths of the wires on the critical paths. Similarly, the only wires of the circuit whose relative lengths are critical to correct operation, are hidden inside the standard cells implementing the primitive components. These standard cells only need to be designed once, with the design engineer having complete control over the semiconductor geometry.

The correct physical implementation of speed independent standard cells is based upon the ability to fabricate 'isochronic forks' [76]. An isochronic constraint on a fork or branch in a wire requires the difference in propagation delays between branches to be negligible. This relies on two physical parameters, the capacitance difference between the wires composing the branches and the threshold voltages of the destination transistors. Reducing the difference in capacitances effectively requires the layout to ensure that wires are short and of approximately equal length. This is aided by the fact that the typical size of a standard cell is small enough to be considered an 'equipotential' region [106]. This is a portion of a circuit within which propagation delays are considered negligible. Techniques to achieve uniform logic threshold voltages have been investigated by van Berkel at Philips Research [117].

Several researchers have designed low level transistor implementations of the more common asynchronous primitive components. The merge elements is generally implemented as standard XOR gate. Example CMOS XOR gate designs may be found in Weste and Eshraghian [127] and similar text books. There exist fewer examples of Muller C-element designs, however possible CMOS implementations have been given by van Berkel [117], Brunvand [12], Sutherland [110] and an optimized version of a "Martin style" c-element is given by Burns and Martin [22]. Implementation of select elements has only been discussed by Keller [64] in his original paper, although Brunvand [12] details a similar level-based component.

Chapter 4

Stable State Synthesis Methodology

This chapter introduces an automatic method for transforming behavioural specifications into delay insensitive circuits. This technique forms the basis for the described compilation methodology. The first section describes an intermediate representation for delay insensitive behaviours, called the stable state graph. This formalism provides an efficient compact representation and allows a classification of delay insensitive circuit behaviours of increasing level of 'complexity'. The remaining sections of this chapter and the next describe the generation of circuitry from stable state graphs.

4.1 Stable State Theory

The first stage of the automated delay insensitive circuit synthesis methodology is conversion of behavioural specification into a *stable state graph* (SSG). A stable state graph is a compact representation of a delay insensitive directed labelled transition system (DLTS), as described in chapter 2, that makes use of the properties of delay insensitive circuit behaviours to reduce the amount of information explicitly stored. This representation also has a compact textual notation that makes it more suitable format for presentation than large transition graphs. A behavioural specification is typically converted into an SSG by an asynchronous circuit compiler, once it has been checked for delay insensitivity and strong determinacy. However, it is possible to generate an SSG directly from a suitable specification formalism or to use an SSG as the initial specification, thus potentially avoiding examination of a large proportion of the circuit behaviour's state space.

4.1.1 Stable States

Stable states are those states of a DLTS that have no out-going output transitions (formally introduced in definition 7 on page 31). Such states are of special interest in the synthesis/behaviour of delay insensitive circuits. In the absence of input excitations, a DI circuit will remain in a stable (or quiescent) state indefinately.

A state of a directed labelled transition system is *terminal* if it has no out-going transitions. A terminal state indicates a point in the circuit's behaviour when the circuit no longer wishes to send or receive signals from its environment. By definition, all terminal states are stable.

Definition 16 *The state $s \in S$ of the DLTS (S, s_0, I, O, T) is terminal, if there does not exist any transition $s \xrightarrow{a} s'$ for any input or output label $a \in I \cup O$.*

When in a stable state, an output signal may only be generated by a circuit once it receives one or more input signals that excite it into an unstable state. Such a sequence of input signals, that lead from a stable to an unstable state is termed a *stimulus* (or *cause*) sequence. Udding's rule $\mathbf{R_1}$ states that every permutation of inputs in a stimulus sequence will lead to the same unstable state, and all of these sequences must be present in the circuit's behaviour (DLTS). From Udding's rules $\mathbf{R_0}$ and $\mathbf{R_1}$, each input can occur only once in a firing sequence. This means that these stimulus sequences may be represented as stimulus (or cause) sets. A corollary of the fact that each input can occur only once, is that circuits with a bounded number of terminals have finite stimulus sets.

Definition 17 *A set, $\{a_1, a_2, \ldots a_n\} \subseteq I$, is a* stimulus set *(or* cause set*) of a stable state $s_1 \in S$ of a DI DLTS (S, s_0, I, O, T), iff $\forall a_i . 1 \leq i \leq n . s_i \xrightarrow{a_i} s_{i+1}$, for any intermediate states $s_2 \ldots s_{n+1} \in S$, and* **Unstable**(s_{n+1}).

Once in an unstable state, a 'live' circuit will generate the appropriate sequence of one of more output transitions and (in the absence of further inputs) settle in a subsequent stable state. In a symmetric way to the stimulus sets defined above, we can define the *response* (or *effect*) sets and *result states* of an unstable state. From Udding's rules, the ordering of outputs in a response sequence is unimportant as every permutation leads (can lead) to the same result state, and each output can occur at most once.

Definition 18 *A set,* $\{p_1, p_2, \ldots p_n\} \subseteq O$, *is a* response set *(or* effect set*) of an unstable state* $s_1 \in S$ *of a DI DLTS* (S, s_0, I, O, T), *iff* $\forall p_1 \,.\, 1 \leq i \leq n \,.\, s_i \xrightarrow{p_i} s_{i+1}$, *for any intermediate states* $s_2 \ldots s_{n+1} \in S$, *and* **Stable**(s_{n+1}). *The final stable state,* s_{n+1}, *is termed a* result state *of the response set.*

For deterministic DI circuits, both the outputs generated and destination stable state depend only upon the input signals received and the current state. For this class of DI circuit each unstable state has only a single response set and a single (unique) result state. Non-deterministic circuits may make an arbitrary choice of outputs and resultant stable state from the alternatives allowed by the specification. This means that a unstable state can have several response sets, and each response set may have more than one result state. However, all 'safe' strongly determinate behaviours (section 2.1.2) have a unique result state for each response set. The restriction to strongly determinate non-determinism is only for convenience, not a limitation of SSGs.

In the exposition above we have assumed that the circuit generates its output sequence "in the absence of further input excitation". Udding's rule $\mathbf{R_2}$ states that if an input signal (or signals) is received by the circuit while it is in an unstable state, then its behaviour is identical to receiving that input (or those inputs) once the circuit has reached the next stable state.

This feature of delay insensitivity suggests an implementation model of circuit behaviour. The behaviour of a delay insensitive circuit may be considered to be a number of transitions between stable states. At each stable state, the circuit receives an enabling 'stimulus' sequence of inputs and generates a resultant 'response' sequence of outputs and enters the destination stable state. If the generation of outputs and stable state transition is instantaneous, further input signals from the environment cannot interfere with the computation.

The effect of an instantaneous (atomic) stable state transition may be achieved by updating the state of the circuit after the inputs have arrived and before any outputs have been generated. The acknowledgement of the state transition may then be used to concurrently form the outputs. Although the environment may send further inputs after the receipt of the first output, the circuit (less the output [merge] circuitry) is already stable prepared for the next input. The delay insensitivity of the circuit specification means that although a further stable state transition may fire before one of the first's outputs has reached the environment, the second

stable state transition cannot generate that output (and therefore interfere with the first).

4.1.2 Stable State Graphs

With the 'reactive system' interpretation of delay insensitive circuit behaviour given in the previous section, it is reasonable to define a representation based on transitions between stable states. These representations have a number of advantages over directed labelled transition systems.

A stable state graph consists of a finite set of vertices, $v_0, v_1, \ldots v_n$, that denote stable states of a DLTS and a number of transitions between them. By convention, the initial stable vertex is denoted v_0. Each of the transitions between states is labelled by a pair of non-empty sets, called the *cause* and *effect* sets respectively. The cause set of a transition contains a finite number of distinct input signals and the effect set a finite number of distinct output signals. These sets correspond to the stimulus and response pairs of the reactive behaviour. Mechanistically, in a given stable state, a transition may *fire* after all of the input signals in its cause set are received in any order, in which case the circuit generates transitions on each of the terminals of the effect set and the circuit enters the destination stable state of that transition. Only a single transition from a vertex may 'fire' at a time (the state of a circuit is always denoted by a single SSG vertex) and if several transitions satisfy the above firing condition, a non-deterministic choice is made between them.

Definition 19 *A* stable state graph *(SSG) is a pair (V, T), where V is a finite non-empty set of vertices (stable states) and T is a labelled transition relation $T \subseteq V \times \mathbb{P}(I) \times \mathbb{P}(O) \times V$, where I and O are non-empty sets of input and output ports respectively and $\mathbb{P}$ denotes the non-empty powerset operator (i.e. $\mathbb{P}(x)$ denotes the set of non-empty subsets of x). For each vertex v of an SSG, we define an auxiliary function, $E(v) \subseteq \mathbb{P}(I) \times \mathbb{P}(O) \times V$, that describes the set of outgoing transitions from state v. For each element (transition), t, in $E(v)$, the terms* $\mathbf{cause}(t)$, $\mathbf{effect}(t)$ *and* $\mathbf{next}(t)$ *denote the first, second and third elements of the triple respectively.*

$$\forall v \in V \,.\, \forall t \in E(v) \,.\, (v, \mathbf{cause}(t), \mathbf{effect}(t), \mathbf{next}(t)) \in T$$

Labelling transitions between stable states by sets of input and output signals creates a very concise representation. Udding's rule $\mathbf{R}_1$ states that consecutive transitions of the same type may be arbitrarily ordered, which is reflected by the lack of ordering within a set. Hence a single set is used to represent all sequences that are permutations of the elements of that set. Similarly Udding's rule $\mathbf{R}_0$ requires that a delay insensitive behaviour contain no consecutive transitions on a single wire. When combined with rule $\mathbf{R}_1$, this means that no sequence of communication actions of the same type may contain repeated signals. Once again this is modelled by our use of sets that cannot contain duplicates. Because of the restriction on a circuit to have a bounded number of terminals, these sets are finite.

In this thesis, I shall make use of a textual notation for stable state graphs. This system of representation simply states the value of the auxiliary function $E(v)$ for each vertex v of an SSG. This concise notation is sufficient to uniquely denote a given SSG, and therefore the delay insensitive DLTS that it represents. The question mark and exclamation mark suffices of input and output signals are normally omitted as these may be inferred by membership of the cause and effect sets respectively.

An example stable state graph expressed in this notation is given in figure 4–1 below. This SSG precisely abbreviates the delay insensitive behaviour of the example circuit behaviour described in chapter 2. The behaviour of this SSG is identical to the transition graph given in figure 2–1, the CCS (process algebra) given in figure 2–2 and the Petri net of figure 2–4. Note that the SSG specification concisely describes the 8 transitions of the transition system and the 6 places of the Petri net in only 3 stable state transitions from 2 stable states.

$$\begin{aligned} E(v_0) &= \{(\{a,b\},\{q\},v_1), \\ &\quad\ (\{c\},\{p\},v_0)\} \\ E(v_1) &= \{(\{d\},\{q\},v_1)\} \end{aligned}$$

Figure 4–1: Example stable state graph (SSG)

A stable state graph is easily generated from a delay insensitive DLTS specification using depth first traversal of the transition system. The technique builds a mapping from stable states (states with no outgoing output transitions) from the DLTS to vertices of the SSG, since not every stable state of the DLTS corresponds

to an SSG vertex. Initially only the initial stable state of the DLTS is mapped to the initial SSG vertex. The construction of the rest of the SSG proceeds as a two phase depth first search from this start state. The first phase of the search recursively follows all input transitions exploring all states reachable through only input signals. At every unstable (and terminal) state reached by this search, the algorithm initiates the second phase search following output only transitions looking for stable states. If this destination stable state does not yet have a mapping, a new SSG vertex is created to represent it. A transition from the previous SSG vertex to this vertex is then added, labelled by the sets of inputs and outputs traversed by the first and second phases of the search respectively. This procedure is repeated until every SSG vertex has been examined.

It is possible to optimize the above SSG generation procedure by making use of Udding's rule $\mathbf{R}_1$ that all permutations of a set of signals of the same type lead to the same state in the normal form of a DLTS. Hence the algorithm can significantly prune its search space by avoiding those states that have already been examined. Continuing to search beyond these states would only find transitions between stable states that have already been discovered.

4.1.3 Unstable Initial States

One problem that arises from the use of a stable state transition model occurs when the initial state s_0 of a directed labelled transition system is unstable. In the proposed method, the initial state of an SSG, v_0, must always be stable. This means that the circuit should generate a number of output signals before reaching an initial stable state. As described in section 3.6, the generation of transitions as soon as power is applied to a VLSI circuit raises a number of implementation and semantic issues. There are several alternative solutions to dealing with this 'unstable initial state' problem.

The first solution is simply to ignore the initial output transitions and treat the result state of an arbitrary initial response set as the initial stable state. A improved solution is to attach IWire components to each of the output terminals of the above implementation that are in the choosen initial response set. The outputs of these IWire components then serve as the real output terminals. The semantics of the IWire component imply an initial output transition. If all outputs are assumed to initially have a low voltage, the interposition of IWire components

ensures that the appropriate outputs are high at the initial stable state. In the above solutions the appropriate number of outputs are maintained by creating low terminals for any output not implemented by the SSG.

Both of the above solutions make use 'static' non-determinism to choose at compile time which of the initial response sets to implement. When the initial unstable state has more than one response set, the circuits implemented by the SSG formed from the result state may be different. Hence it may be reasonable for an automated synthesis system to select the initial response set that leads to the stable state that may be implemented with the least hardware.

Another approach is to introduce a 'hidden global reset' input signal that must be received before the circuit becomes operational. Such a signal is probably already required in order to initialize c-elements and select elements. Using this approach the circuit designed can decide either to use a choice element to implement initial non-determinism or directly select an initial response set (and choose an implementation as described in the paragraph above). As will be explained in later sections, because the 'global reset signal' is used only once its implementation is quite straightforward. In the case when the initial outputs are deterministic, or when using static determinism to choose an initial output sequence, the reset input is merged with each of the appropriate outputs, or connected directly to an output if that output only occurs in this initial sequence. For dynamic non-deterministic solutions, the reset is fed into an N-output choice element, where N is the number of alternative output sequences, and each of these outputs are used prescribed above to generate the required initial outputs.

4.1.4 Terminal States

The second problem introduced by the use of the stable state transition model of circuit behaviour is the implementation of terminal states. The handling of terminal states is considered a special case by the design methodology, as these 'rare' states complicate the exposition of the synthesis method. Indeed, most behavioural specification formalisms are unable to describe terminal states, including Ebergen's DI commands, Martin's handshaking expansions and Josephs and Uddings' DI algebra. Interestingly, because the DLTS is usually state minimized before conversion into an SSG, there is only a single terminal state, as all terminal states are bisimilar.

Two types of terminal states are encountered in the generation of an SSG from a DI DLTS: The terminal states that are the destination vertices of stable state transitions and those terminal states that are reachable by stimulus sequences from an SSG stable state.

The first of these cases is handled by conceptually extending the set of stable states of an SSG to include the terminal stable state $\perp$. This state is only used as the destination vertex of stable state transitions, i.e. $\mathbf{next}(t) = \perp$. This requires the type T of the SSG (V, T) to be extended to $V \times \mathbb{P}(I) \times \mathbb{P}(O) \times (V \cup \{\perp\})$. The 'logical' stable state $\perp$ does not necessarily require additional state in synthesized hardware. One property of the terminal state, $\perp$, is that it is implemented by any stable state, using the implementation relation defined in section 2.4. Taking advantage of this property, stable state transitions leading to terminal states can internally perform a hardware state transition to any implemented stable state. Better still, stable state transitions to terminal states need not perform an internal (hardware) state transition at all. Such transitions need only generate the appropriate output signals.

The remaining case occurs when a stimulus set of a stable state of an SSG leads to a terminal rather than an unstable state. These 'partial' transitions must be represented in an SSG in order for a circuit to be correctly implemented by the proposed circuit synthesis methodology. Without such transitions, the methodology would not consider the arrival of the input signals from such a stimulus set at that stable state. The receipt of such signals may interfere with the operation of the circuit generating additional outputs, thereby breaking the behavioural specification.

In order to represent such 'partial' stable state transitions, the definition of an SSG is extended even further to allow empty effect sets, in addition to terminal destination states, i.e. $\mathbf{effect}(t) = \emptyset$ which implies that $\mathbf{next}(t) = \perp$. This requires the type T of the SSG (V, T) to be further expanded to $V \times \mathbb{P}(I) \times \mathbb{P}_0(O) \times (V \cup \{\perp\})$, where $\mathbb{P}_0$ denotes the usual powerset operator containing the empty set, $\mathbb{P}_0(x) = \mathbb{P}(x) \cup \emptyset$.

Such 'placing holding' stable state transitions are not really transitions at all. They simply state that the circuit, at the source stable state, should be able to receive the cause (or stimulus) set of inputs without generating any outputs. The proposed method typically implements these 'partial' transition, by treating them as normal transitions generating a signal corresponding to that transition firing

which is subsequently ignored. The inclusion of these partial state transitions, with empty effect sets, allows the use of specific optimizations such as terminal state collapsing (see section 6.2.2).

Stack Example

The stable state graph of the current delay insensitive stack element example is given in figure 4–2 below. This SSG precisely abbreviates the behaviour of the stack element presented in section 2.5. The given SSG concisely encodes the 19 state transition system in figure 2–8 in only 5 stable states, and 14 stable state transitions. Note that the signals `ack-push!`, `ack-pop!`, `dack-push?` and `dack-pop?` have been abbreviated for brevity throughout the rest of this thesis to `apush!`, `apop!`, `dapush?` and `dapop?` respectively.

$$
\begin{aligned}
E(v_4) &= \{(\{push\},\{full\},v_4),\\
&\quad (\{pop\},\{apop\},v_1)\}\\
E(v_3) &= \{(\{push,dapush\},\{apush,dpush\},v_3),\\
&\quad (\{push,dfull\},\{full\},v_4),\\
&\quad (\{pop,dapush\},\{dpop,apop\},v_2),\\
&\quad (\{pop,dfull\},\{apop\},v_1)\}\\
E(v_2) &= \{(\{dapop,push\},\{apush,dpush\},v_3),\\
&\quad (\{dapop,pop\},\{dpop,apop\},v_2),\\
&\quad (\{dempty,push\},\{apush\},v_1),\\
&\quad (\{dempty,pop\},\{empty\},v_0)\}\\
E(v_1) &= \{(\{push\},\{apush,dpush\},v_3),\\
&\quad (\{pop\},\{dpop,apop\},v_2)\}\\
E(v_0) &= \{(\{push\},\{apush\},v_1),\\
&\quad (\{pop\},\{empty\},v_0)\}
\end{aligned}
$$

Figure 4–2: Stack element stable state graph (SSG)

4.2 Properties of Stable State Graphs

It is possible to define a classification of stable state graphs based on properties of their stable states. This is similar to Udding's classification system based upon traces. The proposed synthesis methodology makes use of this classification system to determine the most appropriate synthesis strategy for parts or all of the stable state graph.

Definition 20 *A stable state graph* (V,T) *is termed* sequential *iff every stable state has at most a single outgoing transition,* $\forall v \in V \,.\, |E(v)| \leq 1$.

Definition 21 *The* reachable state set, **reachable**(t) *of a stable state transition,* t*, is defined to be the transition's destination state and all reachable states of that state,* **next**$(t) \cup$ **reachable**(**next**(t))*. The reachable state set,* **reachable**(s) *of a stable state,* v*, is defined to be the union of the reachable states of all of that state's outgoing transitions,* $\bigcup_{t \in E(v)}$ **reachable**(t).

Definition 22 *A stable state transition* t *from stable state* v *of a stable state graph* (V,T) *can only be fired once, written* **once**(t)*, iff* v *is not a member of* t*'s reachable set,* $v \notin$ **reachable**(t)*. A stable state* v *can only ever be reached once iff all of its outgoing transitions can only be fired once, i.e.* v *is not a member of its own reachable set.*

Definition 23 *An input signal* i *of a sequential stable state graph* (V,T) *is termed* stateless *iff every occurrence of that signal synchronizes with the same set of input signals to produce the same set of output signals.* $\forall t_1, t_2 \in T$*, if* $i \in$ **cause**(t_1) *and* $i \in$ **cause**(t_2) *then* **cause**(t_1) = **cause**(t_2) *and* **effect**(t_1) = **effect**(t_2)*. A sequential stable state graph is termed* stateless *iff all of its input signals are stateless.*

Definition 24 *An input signal* i *of a sequential stable state graph* (V,T) *is termed* trivially stateless *iff that input signal only appears in a single stable state transition from a single stable state,* $\forall t_1, t_2 \in T$*, if* $i \in$ **cause**(t_1) *and* $i \in$ **cause**(t_2) *then* $t_1 = t_2$*. A sequential stable state graph is termed* trivially stateless *iff all of its input signals are trivially stateless.*

Definition 25 *A stable state v of a stable state graph (V,T) is termed* concurrent *if more than one stable state transition may fire,* $\exists t_1, t_2 \in E(v)\,.\,\mathbf{cause}(t_1) \subset \mathbf{cause}(t_2)$. *A stable state that is not concurrent is termed* distinct, *where every potential set of permissible input signals uniquely identifies a single stable state transition.*

Definition 26 *A stable state transition $t_1 \in E(v)$ from a stable state v is termed* atomic, *written* $\mathbf{atomic}(t_1)$, *if there are no transitions from v whose cause sets are subsets of t_1's cause set.*

$$\neg\exists t_2 \in .E(v)\,.\,\mathbf{cause}(t_2) \subset \mathbf{cause}(t_1)$$

Definition 27 *Two distinct stable state transitions $t_1, t_2 \in E(v)$ from a stable state v are termed* concurrent, *written* $\mathbf{concurrent}(t_1, t_2)$, *if they can occur simultaneously.*

$$\exists t_3 \in E(v)\,.\,\mathbf{cause}(t_1) \subseteq \mathbf{cause}(t_3) \wedge \mathbf{cause}(t_2) \subseteq \mathbf{cause}(t_3) \wedge t_1 \neq t_2$$

In the definition of transition concurrency above, the existance of transition t_3 indicates that all of the inputs in its cause set may be sent to the circuit simultaneously. The concurrent arrival of these input signals is sufficient for transitions t_1 and t_2 to become enabled to fire simultaneously. It is the task of the circuit implementation to ensure that either the concurrent firing of transitions either does not violate the circuit's specification or to arbitrate between the two transitions.

Definition 28 *A stable state v of a stable graph (V,T) is said to exhibit* complex concurrency *if it has two concurrent atomic transitions that share one of more output signals in their effect sets.*

$$\exists t_1, t_2 \in E(v) \quad . \quad \mathbf{atomic}(t_1) \wedge \mathbf{atomic}(t_2) \wedge \mathbf{concurrent}(t_1, t_2) \wedge \mathbf{effect}(t_1) \cap \mathbf{effect}(t_2) \neq \emptyset$$

Definition 29 *A stable state v of a stable state graph (V,T) is* non-deterministic *if there exist two stable state transitions from that state with the same cause sets but with differing destination states or effect sets.*

$$\exists t_1, t_2 \in E(v) \quad . \quad \mathbf{cause}(t_1) = \mathbf{cause}(t_2) \wedge (\mathbf{effect}(t_1) \neq \mathbf{effect}(t_2) \vee \mathbf{next}(t_1) \neq \mathbf{next}(t_2))$$

A stable state that is not non-deterministic is termed deterministic. A stable state graph is deterministic if all of its states are deterministic and non-deterministic if any of its states are non-deterministic.

Definition 30 *A stable state v of a stable state graph (V,T) exhibits* dynamic non-determinism *if it arbitrates between two (or more) concurrent input signals. This may be detected at a stable state when the effects of one stable state transition are disabled by the concurrent arrival of additional input signals, that is*

$$\exists t_1, t_2 \in E(v) \;.\; \mathbf{cause}(t_1) \subset \mathbf{cause}(t_2) \wedge \mathbf{effect}(t_1) \not\subset \mathbf{effect}(t_2) \wedge \nexists t_3 \in E(v) \,.\, \mathbf{cause}(t_3) = \mathbf{cause}(t_1) \wedge \mathbf{effect}(t_3) \in \mathbf{effect}(t_2)$$

4.3 Signal Instance Graphs

An important abstraction used by the proposed circuit synthesis methodology is the *Signal Instance Graph* (SIG). Signal instance graphs are a reduced representation of the stable state graphs introduced in the previous sections. At each stable state, the environment is allowed to transition a subset of the available input signals, the union of the cause sets of transitions from that stable state. Although a given input signal may occur in more than one cause set, they refer to the same *signal instance*. In a directed labelled transition system, it is not clear whether two transitions, with the same label, refer to different instances or alternative interleavings of the same concurrent signal instance. A SIG is a data structure describing the 'use' (or effect) of a particular input signal during the delay insensitive circuit's behaviour. It records the stable states at which that input signal occurs and possible transition paths between them. In this way, a SIG may be considered the result of a projection (or hiding) operator, such as the restriction operator $\restriction$ in trace theory or the hiding operator $\setminus$ in CCS.

The SIG of a given input signal is a subgraph (embedded graph) of an SSG formed by only those vertices at which that input signal occurs. There is a directed transition between two vertices (stable states) of a signal instance graph, if it is possible to reach the destination vertex from the source vertex either directly or via intermediate stable states at which the input signal does not occur. Once an input signal transitions at a stable state, the next transition of that signal is at one of the stable states reachable by a single transition in that input's SIG.

Definition 31 *An input signal, $i \in I$, may be received at a stable state $v \in V$ of the stable state graph (V,T), written* **Occurs**(i,v), *iff* $\exists t \in E(v) \,.\, i \in$ **cause**(t).

Signal instance graphs are conceptually similar to the use of internal derivatives and observed derivatives of a composite state, introduced to aid the definition of circuit composition in section 2.3. In circuit composition, transitions used for internal communication between components are hidden. In SIGs, the synthesis task complexity is reduced by hiding states at which the given input does not occur.

Finally, to aid circuit synthesis and optimization, each SIG transition is labelled with additional information. Each label consists of a non-empty set of *signal instance paths*, where each signal instance path is a non-empty set of stable state transitions. The set of signal instance paths represent the connected paths through the SSG between the source and destination vertices of that SIG transition. Each signal instance path is represented by a set of SSG transitions, rather than a list or sequence, because ordering information is not required and this avoids problems with unbounded path lengths. Infinite transition paths can occur when an SSG contains a cycle, where the input does not occur at any of the states in the cycle. The DI circuit synthesis methodology makes use of these SIG transition labels when choosing which hardware signals are to be used to update internal state machines.

Definition 32 *A* signal instance transition sequence, *s, of a stable state graph, (V,T), and an input signal, $i \in I$, is a sequence of stable state transitions, $s = (t_1, t_2, \ldots t_n)$, that forms a connected path, $\forall 1 \leq m \leq n \,.\, \exists v_m \in V \,.\, t_m \in E(v_m)$ and* **next**$(t_m) = v_{m+1}$, *and the input signal i does not occur at any of the intermediate states,* $\forall 1 < m \leq n \,.\, \neg$**Occurs**$(i, v_m)$. *The notation* **src**$(s)$ *refers to the initial state v_1,* **dst**(s) *to the final state v_{n+1}, and* **set**(s) *to the set of transitions of s,* $\bigcup_{m=1}^{n} \{t_m\}$.

Definition 33 *The* signal instance graph *(SIG) of a stable state graph, (V,T), and an input signal, $i \in I$, is a pair (V_i, T_i), where $V_i = \{v \in V | \textbf{Occurs}(i,v)\}$. T_i is a transition relation between vertices of V_i labelled with a non-empty set of non-empty sets of stable state transitions in T, $T_i \subseteq V_i \times \mathbb{P}(\mathbb{P}(T)) \times V_i$. There is a transition between two states u and v, $(u, L, v) \in T_i$, iff there exits a signal instance transition sequence, s, starting at u and ending at v, i.e.* **src**$(s) = u$

and $\mathbf{dst}(s) = v$. *The label* L *of a transition* $t \in T_i$ *between stable states* u *and* v, *written* $\mathbf{label}(t)$, *is defined to be* $\bigcup\{\mathbf{set}(s)\}$ *for all signal instance transitions sequences,* s, *such that* $\mathbf{src}(s) = u$ *and* $\mathbf{dst}(s) = v$.

This abstraction provides a 'view' of the behaviour of an asynchronous circuit from the perspective of a single input signal. The action of an input signal at each stable state may be considered stateless, and independent of its occurrence at any other stable state.

4.4 Generic Implementation Strategy

Before presenting the proposed SSG-based circuit synthesis method, the following sections describe of simple (naïve) generic implementation strategy for any delay insensitive specifications. The 'generic' strategy serves as a suitable starting point, as the SSG method can assume in its circuit decomposition algorithms that any resulting circuit can always be implemented (inefficiently) by the generic implementation strategy. A widely known result in the delay insensitive circuit community is that any delay insensitive circuit may be constructed systematically from an m-input SEQ element, an $m \times n$-input decision wait element and a number of merge components [5,97]. The organization of these components is given below.

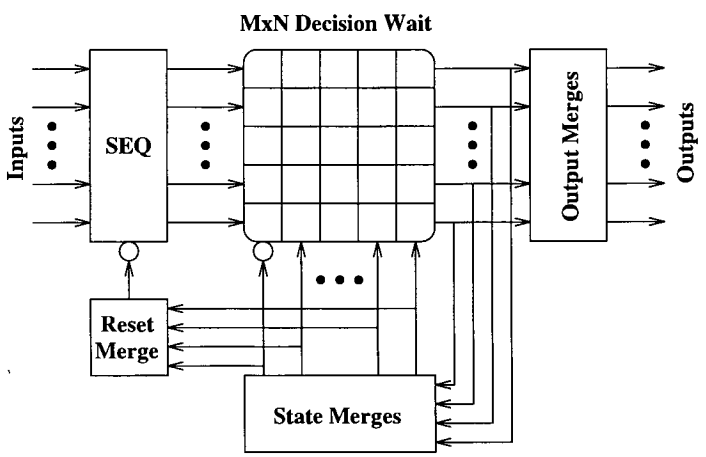


Figure 4–3: Generic DI circuit implementation

This solution uses the m-SEQ element to process the circuit's m inputs signals. This SEQ element arbitrates between all input signals, effectively removing all

concurrency from the specification, serialising the input transitions. Internally, all input signals transition one at a time. Once each input has been processed, the SEQ element is re-initialized and the next input is allowed to enter the circuit.

Inside the circuit, the circuit function or computation is performed by an $m \times n$ decision wait element, much like a PLA is often used to implement an arbitrary finite state machine in a conventional synchronous circuit. The row inputs (the m-input axis) consists of the 'serial' input transitions generated by the SEQ element. The columns (the n-input axis) maintain the current internal circuit state. The column corresponding to the initial state of the circuit has an initialized column input. The mn outputs of the decision wait are generated as a result of each input signal/state combination. These signals are then used to i) determine the 'next' internal state, ii) generate the appropriate output signals and iii) reset the m-SEQ component to allow the next input.

The number of merge gates required to generate the appropriate set of outputs and internal state signals is dependent upon the behaviour of the circuit. However, the signal required to reset the SEQ element may be obtained by merging the n internal state signals (or $n-1$ state signals if the initial state is never revisited).

The number of internal states, n, required by this method can be determined by analysis of the stable state graph of a circuit. The total number of states required by the SSG (V, T) is given by the expression

$$\sum_{v \in V} \left| \bigcup_{t \in E(v)} [\mathbb{P}_0(\mathbf{cause}(t)) \setminus \mathbf{cause}(t)] \right|$$

where $\mathbb{P}_0$ denotes the powerset operator containing the empty set, $\emptyset$, i.e. $\mathbb{P}_0(x)$ represents the set of all possible subsets of the set x.

This expression is derived from the fact that a state is needed for each intermediate interleaving of input signals. For a given stable state transition, t, the intermediate states are represented by the elements of $\mathbb{P}_0(\mathbf{cause}(t))$, less the state represented by the element $\mathbf{cause}(t)$ which corresponds to the output/active state when all prerequisite signals have arrived. These states must be summed for each transition from a stable state, taking into account the shared states that occur when $\mathbf{cause}(t_1) \cap \mathbf{cause}(t_2) \neq \emptyset$.

Each internal state may now be uniquely identified by the value (v, i), where v is a stable state, and i is the set of the input signals that have been received at

that state, $i \in \bigcup_{t \in E(v)} \mathbb{P}_0(\mathbf{cause}(t)) \setminus \mathbf{cause}(t)$. Note that there is no internal state corresponding to an 'unstable' state, of the form $(v, \mathbf{cause}(t))$, but instead it is represented by the destination of the stable state transition, $(\mathbf{next}(t), \emptyset)$.

Each internal state transition can be classified as either inter-stable state or intra-stable state. At a state, (v, i), when an input signal j is received, then if $i \cup \{j\} = \mathbf{cause(t)}$ for any $t \in E(v)$, then the corresponding decision wait output forms an inter-stable state transition and is used both to generate the outputs, $\mathbf{effect}(t)$, and to form the state input to the state $(\mathbf{next}(t), \emptyset)$. Otherwise, $i \cup \{j\} \in \bigcup_{t \in E(v)} \mathbb{P}(\mathbf{cause}(t)) \setminus \mathbf{cause}(t)$, the decision wait output is an intra-stable state transition to the state $(v, i \cup \{j\})$. For non-deterministic inter-state transitions an arbitrary transition is selected from the circuit's specification.

In the above definition, an inter-stable state transition takes precedence over an intra-stable state transition. This may cause some unreachable internal states when $\mathbf{cause}(t_1) \subset \mathbf{cause}(t_2)$ for transitions $t_1, t_2 \in E(v)$. For this reason (and others described in the next section), the value n determined above is only a close upper bound.

The decision wait element is often 'sparse' since the input j cannot occur at the state (v, i) when either $j \in i$ or $\forall t \in E(v), i \cup \{j\} \not\subseteq \mathbf{cause}(t)$. In such a case, the output of the decision wait element is not used.

Outputs are only generated by transitions between stable states. The circuit's output signals are implemented by merging together the internal inter-stable state transition signals of all such transitions that generate that output. Hence, the output o is created by merging together the inter-state transition signals t where $o \in \mathbf{effect}(t)$. For circuits with non-deterministic output behaviour, an arbitrary output sequence is chosen from the circuit's specification.

One major disadvantage of this generic synthesis approach is the combinatorial state explosion that results from concurrent input signals. Concurrent output signals, however, do not cause such a problem.

Consider the specification of the Muller C-element. This has only a single stable state and a single stable state transition. Using the generic design methodology, this circuit requires three internal states, which are denoted $(0, \emptyset)$, $(0, \{a\})$ and $(0, \{b\})$. The corresponding implementation is given below.

This example demonstrates that this method tends to generate decision wait

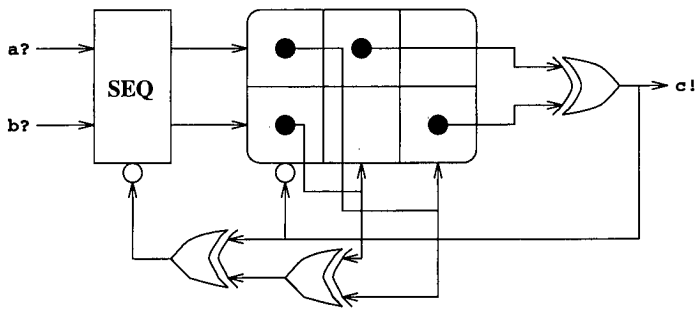


Figure 4–4: Generic implementation c-element

elements that are sparsely populated, due to the large number of invalid and unspecified input transitions at each state of the specification.

4.5 Improved Generic Solution

There are a number of improvements that greatly improve the efficiency of the generic implementation strategy described above. These improvements either reduce the size of the central decision wait element or reduce the complexity of the SEQ element.

4.5.1 Decision Wait Improvements

The size of the $M \times N$ decision wait element may often be reduced because it is typically sparsely populated. This is because the circuit's behavioural specification often guarantees that a given input signal (row) can never occur at a particular state (column). The corresponding output co-ordinate is therefore not live, and hence never used. Row/column compression (as described in section 6.1.6) implements two rows (or columns) of a sparse decision-wait element using a single row (or column). Two rows may be 'compressed', iff at every column position the two outputs are not both 'live'.

For example, column compression can be applied to generic implementation of the c-element given in the previous section. The sparse decision wait in figure 4–4 has two columns, corresponding to states $(0, \{a\})$ and $(0, \{b\})$, that may be compressed. The resulting circuit is given in the figure below.

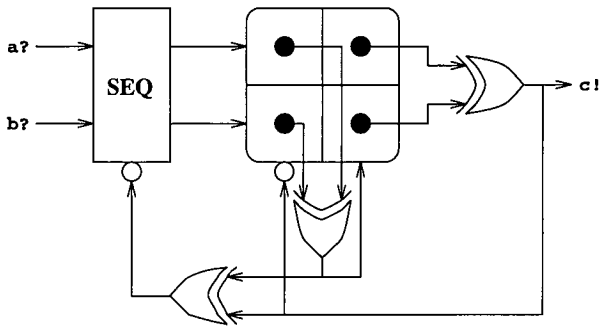


Figure 4–5: Improved generic implementation c-element

The above implementation can be improved further by recognising that the two rows are never distinguished, as their outputs are always merged together. This allows the row/column elimination optimization of section 6.1.5 to be applied. This circuit transformation replaces the 2×2 decision wait and two merge gates with a single 2×1 decision wait element. The final optimization is to recognize that the resulting 'cross-coupled' 2×1 decision wait implements a toggle element (as shown in figure 3–11).

Hence, starting with the generic c-element implementation given in figure 4–4, a delay insensitive circuit compiler can generate the circuit given below.

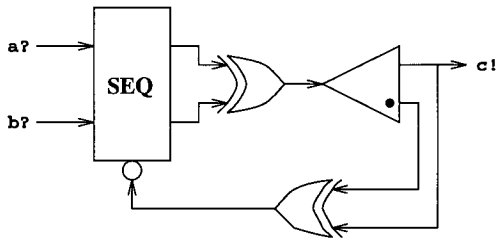


Figure 4–6: Final generic implementation c-element

The behaviour of this circuit has an obvious interpretation based on its implementation; the arrival of the second input, either `a?` or `b?`, causes the output `c!`. This is a correct implementation of a Muller c-element. Although still very inefficient, this implementation is an improvement upon the original generic implementation given in figure 4–4. Infact an arbitrating implementation of a Muller c-element similar to this one was presented in Keller's original paper [64]. In the next chapter, we present a synthesis methodology that implements the above specification by a single primitive component; a standard two-input c-element.

4.5.2 SEQ element Improvements

The role of the k-SEQ element in the generic implementation methodology is to remove all concurrency between the circuit's input signals. However, there are several situations when alternative methods can be used to ensure that input transitions occur sequentially.

The simplest case is where an input signal is constrained by the circuit specification to never occur concurrently with any of the other inputs. This occurs when the input signal, i, only occurs on its own in a cause set, $\forall t \in T$ if $i \in \mathbf{cause}(t)$ then $\mathbf{cause}(t) = \{i\}$. In this case, the input i does not need to be fed into the k-SEQ element, but may be passed directly to the row input of the decision wait element. Similarly, because the SEQ element does not perform an arbitration, the decision wait outputs from the corresponding row of the decision wait element are not merged into the SEQ-element's reset input.

A more advanced approach is to only arbitrate between those signals that could occur concurrently. However, because each signal is fed into only a single arbiter, all signals that can occur concurrently with a given signal (even at different stable states) must also share the same SEQ element. This induces a partitioning, $\pi : I \rightarrow \mathbb{N}$, of the input signals. Two signals are in the same partition if they can occur concurrently. Formally two input signals, i and j, of the SSG (V, T) are in the same partition, $\pi(i) = \pi(j)$, if $\exists t \in T$ such that $\{i, j\} \subseteq \mathbf{cause}(t)$. The synthesis method determines the unique maximum partitioning and uses a single SEQ element to arbitrate between that partition's input signals. This transformation improves the circuit because several small arbiters typically require less hardware than a single large arbiter. However, for $\mathcal{O}(n)$ decompositions (e.g. ring arbiters) this may not be much of an improvement.

For example, consider the stable state graph in figure 4–1 on page 88. This circuit has four input signals, $a?$, $b?$, $c?$ and $d?$, that may be partitioned into three sets, $\{a?, b?\}$, $\{c?\}$ and $\{d?\}$. Hence this circuit may be implemented by the generic implementation strategy using only a 2-SEQ element (to arbitrate between $a?$ and $b?$) instead of a 4-input SEQ component.

One consequence of splitting the SEQ element in the generic implementation strategy is that the resulting smaller SEQ elements need not always be reset. If the inputs of a SEQ element are guaranteed never to occur after a final arbitration is made, there is no need to reset that SEQ element. This is demonstrated in the

example used in the previous paragraph, taken from figure 4–1. In this example, once the SSG reaches the stable state v_1, the $a?$ and $b?$ inputs can never transition again. Hence, the single SEQ element needs only be reset by the transitions from $(0, \emptyset)$ to $(0, \{a\})$ and $(0, \emptyset)$ to $(0, \{b\})$. This saves the merge circuitry required on the $(0, \{a\})$ to $(1, \emptyset)$ and $(0, \{b\})$ to $(1, \emptyset)$ transitions.

4.6 Proposed Synthesis Methodology

This section outlines the proposed delay insensitive circuit synthesis methodology. The synthesis method compiles behavioural specifications described as stable state graphs (SSGs) into delay insensitive compositions of primitive circuit elements. Because the method uses a canonical graph-based representation of a circuit's global behaviour, the resulting circuit implementations are more efficient than those generated by existing syntax-directed translation based compilers.

4.6.1 Implementation Model

The great advantage of the SSG representation of a circuit's behaviour is that it distinguishes between the transitions that occur due to concurrency and those 'causal' transitions that prescribe the circuit's functional behaviour. This distinction of transitions reveals the important role of stable states in delay insensitive reactive systems. This forms an interesting asymmetry as stable states are more important in circuit synthesis than maximally unstable states.

The synthesis methodology is based upon the interpretation of delay insensitive circuit behaviour as a series of transitions between stable states. Conceptually, each stable state is implemented independently and these subcircuits combined to produce the desired global behaviour. This decomposition takes advantage of the fact that transitions occurring at different stable states represent distinct actions and not just concurrent interleavings of the same signal. These distinct actions of the same input signal may be described using a traditional state graph model and implemented using a delay insensitive finite state machine. Distinct actions of the same output signal may be trivially combined using 'merge' circuitry. In the middle, the subcircuit for each stable state receives all of the input for each of the external signals that can occur at that state and generates a single transition,

representing the stable state transition that 'fires'. The transition signal is then used to generate the required output signals and change the state of the routing circuitry to direct the appropriate inputs to the subcircuit implementing the next stable state.

This decomposition conceptually separates out the implementation of delay insensitive circuits into three layers; the routing layer, the synchronization (and arbitration) layer and the output layer. The *routing layer* receives external input signals and generates internal input signal instances for the synchronization layer. This layer is implemented as a number of delay insensitive finite state machines, one for each external input. This layer implements the main state holding functions of the final circuit. The *synchronization layer* performs the synchronization and arbitration of the final circuit. This layer is conceptually organized as a separate subcircuit for each stable state. Finally, the *output layer* combines the results of the synchronization layer to generate the outputs, the finite state transition signals for the routing layer and the resulting external outputs.

These layers are depicted in the proposed circuit "model" shown in figure 4–7 below. Boxes $R_1 \ldots R_n$ represents the routing layer circuitry, one subcircuit for each of the n external input signals. Boxes $S_1 \ldots S_n$ represent the synchronization layer subcircuits, one for each stable state in the SSG. Finally the box labelled O depicts the output layer circuitry. There is an output from each S_i to the output layer for each stable state transition that can occur at the stable state 'implemented' by S_i.

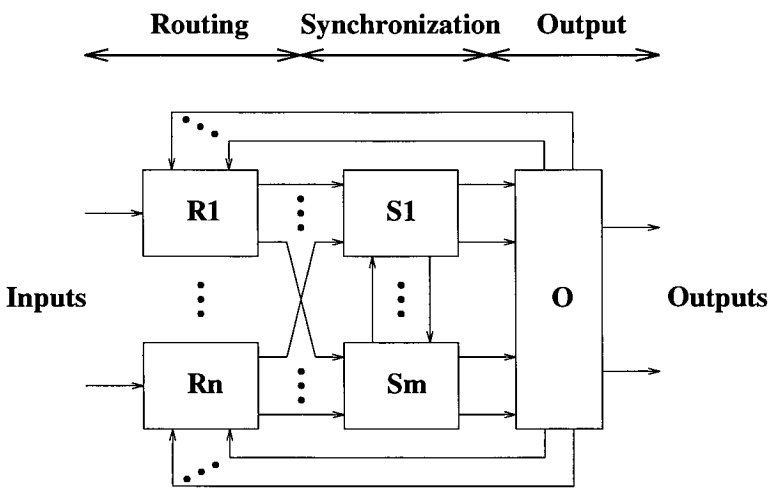


Figure 4–7: Proposed implementation model

This model only describes the conceptual decomposition of a circuit into components. In practice a circuit may not necessarily contain hardware for all of the above layers. For example, a circuit that performs no synchronization or arbitration between input signals will have no synchronization layer, and a stateless circuit will have no routing layer. In these cases the appropriate layer consists entirely of 'wiring' connecting the inputs to the outputs. For some classes of circuits, the next state of the routing circuitry is always known so there is no need for feedback signals from the output circuitry following a stable state transition. Finally, the physical separation of the circuit into distinct layers may be obscured by the circuit optimizations and other transformations applied during circuit synthesis. Hence it may be impossible in the final circuit to assign a component to a single layer in the above model.

Given this systematic decomposition strategy, it is possible to define the delay insensitive behaviours of each of the subcircuits. These subcircuit behaviours when composed, using the composition operator of section 2.3, implement the desired global behaviour, using the notion of implementation defined in section 2.4. One possible approach is to implement each of these subcircuit specifications using the (improved) generic implementation strategy described previously. However, there exist much more efficient implementation strategies for classes of these circuits, as will be described in the rest of this thesis. These strategies can be shown to be optimal for certain classes of circuits. For example, the behavioural specifications of a merge gate, a c-element and a select element result in implementations containing a single merge, c-element and select respectively. Similarly, the behavioural sepcifications of call elements and 2×1 decision wait elements, result in the standard decompositions into primitive elements described in chapter 3.

Unfortunately, there still exist some classes of subcircuits for which the only automated implementation strategy (to the author's knowledge) is the improved generic implementation approach. These are those specifications containing complex forms of arbitration. However, even for these cases, the initial decomposition into subcircuits allows the introduction of concurrency that cannot be exploited in a purely generic implementation.

4.6.2 The Committee Problem

Central to the synthesis methodology is the implementation of the synchronization layer subcircuits. Conceptually these correspond to individual stable states. As will be shown below, stable state implementation is identical to the well known *committee problem* in parallel program design [24]. This problem, capturing many synchronization and exclusion problems, is also sometimes known as the *n-party rendezvous* or *multirendezvous*. The problem is defined in Chandy and Misra's book in terms of professors and committees.

> *Professors at an academic institution organize themselves into committees. Each committee has an unchanging membership of one or more professors. From time to time a professor may decide to attend a committee meeting. He (or she) waits until a meeting of a committee of which he (she) is a member is started. The restrictions on meetings are as follows: (1) a committee meeting may be started only if* all *members of that committee are waiting; (2) no two committees may meet simultaneously if they have a common member. Given that all meetings terminate in a finite time, the problem is to devise a committee-meeting scheduler that satisfies these restrictions. The scheduler must also guarantee that when all members of a committee are waiting, at least one of them will attend some meeting.*

From the above description it is clear that the committee problem includes aspects of both synchronization and mutual exclusion. Synchronization is required because all members of a committee must be waiting before a meeting can start; mutual exclusion is required, because two committees with a common member cannot meet at the same time. This mutal exclusion also implies the need for arbitration for example when a common member of two otherwise waiting committees arrives (starts waiting) or when two members of mutually exclusive committees arrive. It can also be seen that the committee problem is a generalization of the *dining philosophers* problem.

The equivalence between single stable state implementation and the committee problem can be seen by considering each stable state transition as a committee and each permissible input signal as a professor. The membership of each committee is defined by the cause set of each state transition. It is also possible to extend

this equivalence to ensure that stable state transitions are mutually exclusive. This needs to be done for complex forms of arbitration. This is achieved by adding a common member to each committee, thereby ensuring that only a single transition can fire at a time. This condition severely restricts the concurrency of the resulting implementation. However, this restriction is only imposed on a small class of stable states.

Central to the proposed synthesis methodology is the ability to efficiently implement individual stable states and thereby find solutions to subproblems of the committee problem. This is essential to producing high quality circuit implementations requiring the minimal amount of hardware. One possible solution is to use the improved generic implementation strategy described above. An alternative solution is to make use of the token-ring based solution, recently presented by Benko [5]. However, there are a large number of stable states that do not require the generality of these solutions, and may therefore be implemented using fewer components.

At the very simplest, the problem defined by a single committee containing a single professor, is easily implemented by a wire. Any problem that is defined by only a single committee may be implemented by a c-element (synchronization) of all the members of that committee. This solution corresponds to the class of sequential circuits given in the next chapter. One property that is extremely useful in decomposition of the committee problem is that the problem may be partitioned into several smaller subproblems that share no professors in common. This means that if all professors only belong to a single committee, the solution may be implemented by a number of c-elements, one for each committee.

A large number of special cases apply to subproblems where no committee is a subset of any other committee. These correspond to the non-concurrent circuits described in the next chapter. Any set of N committees with identical membership, may be implemented by a single committee of the same membership, and when it meets (the state transition fires) the output may be fed into an N-choice element, making an arbitrary decision between the original committees. This class of circuits is also amenable to synchronization rollback and partitioning decomposition methods as described in sections 5.2.3 and 5.2.4 respectively.

Many of the synchronization layer decomposition methods described in this thesis represent novel solutions for special cases of the committee problem. As further new delay insensitive solutions are found to special cases of this problem,

these strategies/theorems may be included in this design style and improve the quality of the resultant implementations.

Chapter 5

Circuit Synthesis

This chapter presents a detailed description of the proposed delay insensitive circuit design methodology, outlined in the previous chapter. Each section describes synthesis 'rules' for an increasingly large class of circuit behaviours. These rules automatically determine the connectivity of networks of primitive components required to implement the specification. Although some of these circuit classes can be considered special cases of later ones, their incremental exposition makes the design methodology easier to understand. Each of the rules presented in this chapter is applicable to the whole class of circuits being examined. Techniques for efficiently handling special cases and circuit optimizations are discussed in the following chapter.

5.1 Sequential Circuit Synthesis

This section describes an algorithmic method for generating *sequential* delay insensitive circuits from stable state graph behavioural representations. The class of sequential delay insensitive circuits is those that have at most a single outgoing transition from each stable state, i.e. for every vertex v, $|E(v)| \leq 1$. Such a circuit permits no alternatives of either input or output signals beyond the non-deterministic re-ordering of concurrent signals. Hence both the specified circuit and its environment are completely deterministic. This restrictive class of circuit behaviour is a subset of Udding's $\mathbf{C_1}$ (synchronization) class of circuits. All circuits specified by handshaking expansions or taxograms (without environment partitioning) belong to this class.

The class of sequential circuit can be subdivided further into three smaller subclasses. These are acyclic sequential circuits, partially cyclic sequential circuits and completely cyclic sequential circuits. From the above definition, a sequential circuit behaviour is a connected finite stable state graph with at most one transition incident from each vertex. If there exists a state with no outgoing transitions then the circuit is *acyclic*, and each vertex is visited only once during a circuit's finite operation. If all vertices have a single incoming transition, then the circuit's behaviour is *completely cyclic*, i.e. its SSG consists of a simple cycle. In this case every vertex is visited repeatedly during the circuit's operation. The remaining *partially cyclic* class of sequential behaviours occurs when there are some 'initial' transitions that fire before the circuit enters a cycle of stable states. In this case the initial stable state has no incoming transitions and the first vertex of the cycle has two; one from the transition that enters the loop and the other from the last vertex of the cycle.

5.1.1 Stateless Sequential Circuits

The first category of sequential circuits that we shall consider is the class of *stateless* sequential circuits, those that do not require any 'state holding' components such as Keller select elements or decision waits. The class of trivially stateless circuits is characterized as those where transitions on a given input may only occur at a single stable state. This constraint may given as for every two distinct states v_1 and v_2 of an SSG, the cause sets of all their outgoing transitions are disjoint, $\forall t_1 \in E(v_1), t_2 \in E(v_2) \,.\, \mathbf{cause}(t_1) \cap \mathbf{cause}(t_2) = \emptyset$. From this definition, all sequential circuits with only a single stable state are trivially stateless.

Trivially Stateless Sequential Circuits

Construction of trivially stateless sequential circuits proceeds in two steps. The first step generates any required synchronization circuitry and the next generates the appropriate outputs from each transition. The aim of the synchronization in this synthesis step is to generate a single signal for each state to indicate that its outgoing transition has fired, called the *transition signal*. For all SSG transitions where $|\mathbf{cause}(t)| = 1$, the single input signal in its cause set may be used as its transition signal. For the remaining cases, $|\mathbf{cause}(t)| \geq 2$, the transition signal is generated by detecting the arrival of all the inputs in the cause set. This is done

by combining all N inputs with an N-input c-element and using the output of this c-element as the transition signal. The constraints on the c-element's environment are ensured by the circuit specification and the completion tree is independent of any delay in the input wires. If the generalized c-elements are constructed by building completion trees of 2-input Muller c-elements, the total number of c-elements required for synchronization in a trivially stateless sequential circuit is $|I| - |V|$, where I is the set of input signals and V is the set of vertices in the SSG.

Once a transition signal has been generated it must trigger the required set of output signals. We first consider the case where each output only occurs at single stable state (analogous to inputs in trivially stateless circuits): this is the case when transitions t_1 and t_2 from disjoint vertices of an SSG have disjoint effect sets, $\mathbf{effect}(t_1) \cap \mathbf{effect}(t_2) = \emptyset$. Each output is generated in response to the transition signal of the appropriate SSG transition. If only a single output is generated by a transition ($|\mathbf{effect}(t)| = 1$), the transition signal may be connected directly to this output terminal, otherwise several outputs need to be generated by the firing of an SSG transition ($|\mathbf{effect}(t)| \geq 2$), so the transition signal must be forked to branch the signal to the appropriate outputs.

In the remaining case, output signals may occur at several stable states of a circuit's behaviour. This requires a given output to be generated in response to (caused by) any of a number of transition signals. Because transition signals are mutually exclusive, such an output may be produced using an N-input merge to combine the transition signals of all the N vertices where that particular output must be generated. If a transition signal is required as an input for more than one merge element, it should be forked and a branch connected to each merge. The number of merge gates required by this general case construction can typically be reduced by applying the 'common subexpression elimination' circuit optimization described in section 6.1.3. The average case performance of the general merge elements in partially cyclic sequential circuits may also be improved using the 'Huffman decomposition' of section 3.4.2.

A demonstration of this synthesis procedure applied to this class of circuit is given below. Figure 5–1 contains the SSG for a trivially stateless sequential circuit and the resulting implementation derived from it. This stable state graph has three stable states, each with a single outgoing transition denoting a cyclic sequential circuit. Because there is no transition t such that $\mathbf{next}(t) = v_0$, it does not form a single cycle of all stable vertices and is therefore a partially cyclic sequential

$$
\begin{aligned}
E(v_0) &= \{(\{a\},\{p\},v_1)\} \\
E(v_1) &= \{(\{b,c\},\{q\},v_2)\} \\
E(v_2) &= \{(\{d\},\{q,r\},v_1)\}
\end{aligned}
$$

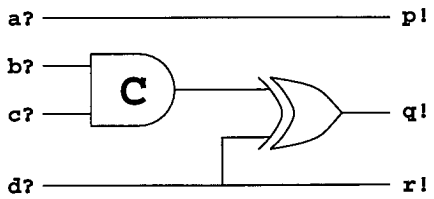


Figure 5–1: Trivially stateless sequential circuit example

circuit. Of the three transitions t_0, t_1 and t_2, from the stable vertices v_0, v_1 and v_2 respectively, only $\mathbf{cause}(t_2) \geq 2$, hence $a?$ is the transition signal for t_0, $d?$ is the transition signal for t_2, and $b?©c?$ is the transition signal for t_1 (where $x©y$ denotes the output of the Muller c-element with inputs x and y). The output $p!$ is only generated by transition t_0, so $a?$ is connected directly to $p!$. The initial stable state v_0 is implemented completely by this single wire. The output $q!$ is generated by two transitions, and is therefore implemented by merging $b?©c?$ and $d?$. The remaining output $r!$ is obtained from the other branch of the $d?$ fork.

One effect of the above synthesis procedure is that the behavioural specifications of the Muller c-element and the merge element are implemented by a single component of the appropriate type.

Non-trivially Stateless Sequential Circuits

In addition to trivially stateless sequential circuits, it is also possible to generate stateless sequential circuits using a special case of the 'state collapsing' optimization of section 6.2.2. The principal consideration in the synthesis of stateless circuits is that the operation of each stable state is independent of all the remaining stable states. This is obviously the case for sequential circuits that have no input signals shared between transitions (trivially stateless circuits). However, it is also the case when two stable states can be implemented using identical hardware. This is because there is no need to use a state holding device to distinguish between two 'identical' states. Hence we term a sequential circuit *stateless*, iff for every pair of transitions t_1 and t_2, if $\mathbf{cause}(t_1) \cap \mathbf{cause}(t_2) \neq \emptyset$ then $\mathbf{cause}(t_1) = \mathbf{cause}(t_2)$ and $\mathbf{effect}(t_1) = \mathbf{effect}(t_2)$.

Synthesis of circuitry for this class of circuits is almost identical to the method described for trivially sequential circuits above. Only a single representative synchronization layer implementation need be constructed for each set of 'identical' stable states, i.e. those stable states whose outgoing transitions have identical cause

and effect sets. The output layer circuitry then treats this set as one vertex with just a single transition signal. This has the net effect of removing all duplicate stable states from the SSG specification. Because the frequency of transition signals during a circuit's operation is now no longer equal but known, the average case performance of output layer general merge elements may improved using the 'Huffman decomposition' optimization described in section 3.4.2. Similarly, some of the output layer merge elements may be implemented using standard OR logic gates as described in section 6.1.9.

An example of this synthesis of a non-trivially stateless sequential circuit is given in figure 5–2. This SSG specification is an example of a completely cyclic sequential circuit. The transitions from states v_0 and v_1, and from v_2 and v_3 are equivalent. These pairs can both be *collapsed*, with the resulting vertices requiring no synchronization hardware and having transition signals $a?$ and $b?$ respectively. The output $p!$ is then produced by both these 'collapsed' vertices and is therefore generated using a two input merge of their transition signals. Although the SSG of the state minimized transition system contains four stable states, the resulting implementation consists of only a single merge element.

$$
\begin{aligned}
E(v_0) &= \{(\{a\}, \{p\}, v_1)\} \\
E(v_1) &= \{(\{a\}, \{p\}, v_2)\} \\
E(v_2) &= \{(\{b\}, \{p\}, v_3)\} \\
E(v_3) &= \{(\{b\}, \{p\}, v_0)\}
\end{aligned}
$$

Figure 5–2: Stateless sequential circuit example

The class of stable state graphs that generate stateless sequential circuits may be extended by performing the 'state combining' optimization of section 6.2.3. However, due to the complexity of this optimization, it is presented in its own section and not described as part of the general case design methodology.

5.1.2 State-Holding Sequential Circuits

State holding components are typically required to implement specifications where the effect of receiving an input signal is dependent upon when in the circuit's behaviour the signal occurs. The circuit requires some internal state to distinguish

between these signal occurrences in order to produce a different response to each. Hence a sequential circuit behaviour is termed *state holding* if for some two distinct transitions t_1 and t_2, $\mathbf{cause}(t_1) \cap \mathbf{cause}(t_2) \neq \emptyset$ and either $\mathbf{cause}(t_1) \neq \mathbf{cause}(t_2)$ or $\mathbf{effect}(t_1) \neq \mathbf{effect}(t_2)$. From this definition, a sequential circuit specification is either state holding or stateless and must contain at least two stable states to be state holding.

Note that the above definition does not determine whether the resulting circuit need contain any state holding primitive components. There are several optimization techniques, discussed in sections 6.2.3 and 6.2.4, which describe special cases of state holding circuits that may be implemented without using select elements or decision waits.

Cyclic Sequence Generators

The class of *cyclic sequence generators* (CSG) circuits is those state holding sequential circuits with a single input terminal (i.e. $|I| = 1$) that have a completely cyclic stable state graph. Cyclic sequence generators form an important classification of delay insensitive circuits as general state holding sequential circuits are typically decomposed into a number of CSGs. The predetermined operational behaviour of CSGs also enables detailed analysis of their performance and power consumption characteristics. For this reason CSGs, and in particular modulo-N counters, have been extensively studied by a number of researchers [41,104,114,118].

The starting point for synthesizing a cyclic sequence generator is an N-toggle circuit, where N is the length of the repeated sequence. This N-toggle may be implemented using any of the decomposition techniques given in section 3.4.7 on generalized toggle elements. The single input of the CSG is then used as the input to the N-toggle, and the N outputs are used as the 'transition' signals for each of the stable states of the CSG's behavioural specification. If every stable state transition t_1 has a single unique output, i.e. $|\mathbf{effect}(t_1)| = 1$ and $\mathbf{effect}(t_1) \cap \mathbf{effect}(t_2) = \emptyset$ for any transition $t_2 \neq t_1$, then the specification is directly implemented by the N-toggle and each of its outputs is used to generate one of the CSG's output signals.

For the remaining forms of cyclic sequence generator circuits, output circuitry consisting of forks and merges is needed. Generation of this circuitry from transition signals is identical to the approach described for trivially stateless sequential circuits above. Each transition is forked as appropriate and merged with other transition signals to form the required outputs. Common subexpression elimination allows these signals to be merged before they are forked, thereby reducing the number of merges required (section 6.1.3). In addition to this general decomposition, a number of CSG specific optimization strategies are described in section 6.1.10.

An example application of this synthesis procedure is described for a four phase toggle circuit below. A four phase toggle is a toggle component that uses a 'four phase' or return-to-zero handshaking convention with its environment. It is a good example of the ability of transition signalling to describe other signalling protocols, such as those used in 'conventional' or speed independent design. The stable state graph for a four phase toggle is shown in figure 5–3 together with its implementation. From its specification, this circuit has a completely cyclic sequential behaviour and hence is implemented as a cyclic sequence generator. The state holding portion of the circuit is a 4-toggle (implemented using the toggle decomposition method into three standard toggle components). The `p!` output is then generated by merging together the transition signals for v_0 and v_1, and the `q!` by merging those of v_2 and v_3. Hence the circuit is decomposed into a delay insensitive network of three toggle components and two merge gates.

$$
\begin{aligned}
E(v_0) &= \{(\{a\},\{p\},v_1)\} \\
E(v_1) &= \{(\{a\},\{p\},v_2)\} \\
E(v_2) &= \{(\{a\},\{q\},v_3)\} \\
E(v_3) &= \{(\{a\},\{q\},v_0)\}
\end{aligned}
$$

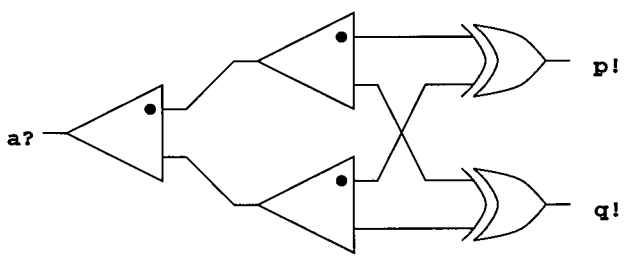


Figure 5–3: Four phase toggle example

The completely deterministic and data-independent behaviour of sequential circuits permits detailed performance analysis. Analysis of the four phase toggle circuit given in figure 5–3 reveals that the response time of the above circuit is $2\tilde{t} + \tilde{x}$ for all instances of the input `a?`, where $\tilde{t}$ and $\tilde{x}$ are the transition times for a toggle and a merge gate respectively. The period for a completely cyclic sequential beha-

viour is the sum of the response times of all transitions in the stable state graph. The period of the circuit for the current example is $8\tilde{t} + 4\tilde{x}$.

Sequence Generators

The class of *sequence generator* circuits is those sequential circuits with a single input. The name is derived from the fact that a single signal may be used to produce an acyclic, partially cyclic or completely cyclic sequence of output transitions. Sequence generators for completely cyclic specifications are implemented using the CSG techniques described in the previous section. A CSG circuit may also be used to implement acyclic sequence generator circuits, since the behaviour of the circuit after that prescribed by the specification is never examined. However, the decomposition techniques for N-toggle circuits, using the decision-wait and select element methods of section 3.4.7, may be optimized for acyclic sequence generators. This is to avoid the transition signal for the final stable state resetting the circuit to its initial state.

Partially cyclic sequential specifications consist of two parts; A sequence of stable state transitions that fire only once, called the *head* of the specification, and a repeated cycle of stable states, called the *loop* of the specification. Such specifications may be implemented using one of the decomposition strategies given below. These strategies reflect the ways in which generalized toggle circuits may be implemented.

The decision wait implementation method requires a $N \times 1$ decision and a two-input merge gate, where N is the total number of stable states in the acyclic sequential specification. Much like the decision wait N-toggle decomposition, the row is fed by the sequence generator input and each column represents the current stable state. The decision wait is initialized to be in the appropriate initial state. Each output of the decision wait is forked to produce the appropriate transition signal and also fed back as a column input to set the next state. The single merge gate is used to combine the two signals which set the state of the DW to the first state of the specification's 'loop'. Precisely, the other branches of the transition signals of the last state of the head and the last state of the loop are merged and used as the column input representing the first state of loop.

The composite implementation method decomposes the specification into an acyclic sequential subcircuit for the head and a completely cyclic sequential sub-

$$
\begin{aligned}
E(v_0) &= \{(\{a\},\{p\},v_1)\} \\
E(v_1) &= \{(\{a\},\{q\},v_2)\} \\
E(v_2) &= \{(\{a\},\{r\},v_1)\}
\end{aligned}
$$

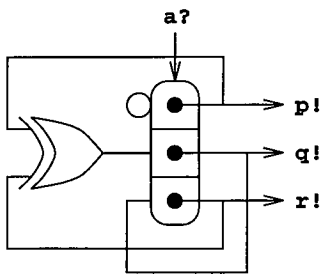


Figure 5–4: Decision Wait PCSG implementation

circuit for the tail. These subcircuits can be implemented using any of the methods described above. The behavioural specification of the head must ensure a distinct transition signal for the final state of the head. This constraint reduces the opportunity to apply any of the state reduction optimizations presented in section 6.2. This final transition signal is used to modify (set) the state of the switch element, redirecting the input signals from the head subcircuit to the loop subcircuit. The set acknowledge from the switch is then used as the transition signal for the final state of the head. Output level circuitry then merges and forks the transition signals from both subcircuits to generate the appropriate outputs. If Huffman encoding of merge trees is used, the loop transition signals are assumed to occur far more frequently than the head transition signals.

$$
\begin{aligned}
E(v_0) &= \{(\{a\},\{p\},v_1)\} \\
E(v_1) &= \{(\{a\},\{q\},v_2)\} \\
E(v_2) &= \{(\{a\},\{r\},v_1)\}
\end{aligned}
$$

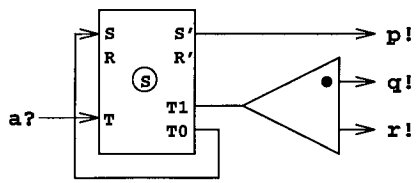


Figure 5–5: Composite PCSG implementation

Finally, partially cyclic sequence generators (PCSGs) may also be implemented using the select decomposition method. Much like the select decomposition of N-toggle circuits, this technique constructs a binary tree of Keller select elements. The test outputs of each select are used as the inputs to the next level in the tree, and the sequence generator input is fed into the root select of the tree. If the numbers of stable states in the head and the loop of the specification are denoted by N_H and N_L respectively, this decomposition requires $N_L + N_H - 2$ select elements and one switch element. The technique is to construct a binary tree of $N_L - 1$ elements to cycle through the loop of the specification. The initial output of the circuit is fed into a switch that initially feeds the input to a binary tree of $N_H - 1$

selects that implements the head of the specification. The transition signal of the last stable state of the head sets the switch element. The remaining output of the switch element acts as the transition signal for the first stable state of the loop, and is used to set the state for the second state of the loop. This decomposition strategy is illustrated by the PCSG example in figure 5–6 below.

$$
\begin{aligned}
E(v_0) &= \{(\{a\},\{p\},v_1)\} \\
E(v_1) &= \{(\{a\},\{q\},v_2)\} \\
E(v_2) &= \{(\{a\},\{r\},v_1)\}
\end{aligned}
$$

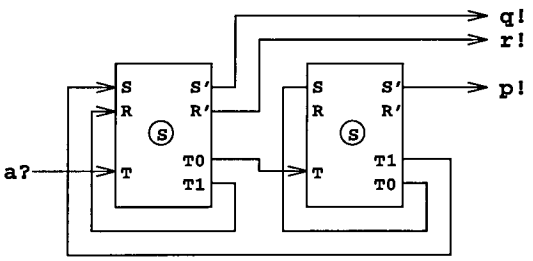


Figure 5–6: Select element PCSG implementation

General Sequential Circuits

We now consider the implementation of general sequential circuits, those whose only restriction is that $|E(v)| \leq 1$, for any state v. We initially restrict our discussion to circuits that only require a single input transition stimulus, i.e. $|\mathbf{cause}(t)| = 1$ for all stable state transitions t. In this case, the behaviour of the circuit may be decomposed into the parallel composition of several sequence generators, one for each of the circuit's inputs. The outputs of these sequence generator circuits are used as transition signals and combined using the standard constructions to generate the output signals. These sequence generator circuits may then be implemented by any of the methods above.

This partitioning of the circuit behaviour is based upon the *instance graphs* of each input signal, as defined in section 4.3. For a stable state graph, the instance graph of a given input signal is a subgraph whose vertices are the stable states of the SSG at which the given input may be received. The vertex v_i is connected by a directed arc to v_j in an instance graph if v_j may be reached from v_i by a sequence of stable state transitions in the SSG without passing through any of stable states in the instance graph. If the initial stable state is not a vertex of the instance graph, the initial states are those states that may be similarly reached from the initial stable state of the SSG. For sequential circuits, there is always a single initial state.

For a circuit specification with only a single input, the instance graph of this input is isomorphic to the circuit specification's stable state graph. All instance graphs of an acyclic circuit behaviour are acyclic and all instance graphs of a completely cyclic behaviour are completely cyclic. However, instance graphs of a partially cyclic behaviour may be either acyclic, partially cyclic or completely cyclic. The strict sequential nature of 'sequential' circuit behaviours ensures that stable state transitions can not be concurrent and that the state circuitry associated with each input is independent of the remaining inputs. Hence, the parallel composition of delay insensitive subcircuits implementing each instance graph (together with the necessary output circuitry) is delay insensitive and implements the original circuit specification.

Once a circuit behaviour is decomposed into several instance graphs, these instance graphs may not be 'state minimal' even if the original behaviour was minimal. Minimizing each signal instance graph can then improve the quality of the resulting circuit by reducing the number of states in each subcircuit. Because minimization preserves observable functionality, the final circuit obtained by composing these minimal subcircuits implements the original specification. The ability to do state minimization at several stages in the synthesis methodology results in more efficient circuits than those produced by syntax directed translation.

Given the subcircuits that implement the signal instance graphs of a sequential circuit behaviour, synthesis of the necessary synchronization and output circuitry is quite straightforward. All those stable states whose outgoing transition t contains more than one input in its cause set use general N-input c-elements (where $N = |\mathbf{cause}(t)|$) to perform the synchronization and generate the stable state transition signals. Each of these transition signals is then forked and merged to produce the required outputs.

An example of this synthesis procedure is given below. Consider the general sequential circuit behaviour described by the stable state graph given in figure 5–7 below. This SSG has four stable states, three inputs and is an example of a partially cyclic behavioural specification. The signal instance graph for the input `a?` has two vertices, which being identical (having no other inputs in the relevant cause sets and equal effect sets) may be state minimized to a single state. Hence the state holding subcircuit for `a?` consists of a wire and a single output (which happens to be the state transition signal for vertices v_0 and v_2). The signal instance graph for input `b?` is completely cyclic having two vertices, hence `b?`'s state holding circuitry

may be implemented by a toggle. Finally, c? has no state holding circuitry, only a wire, as its signal instance graph contains only a single vertex. The transition signal for state v_1 is the initial output of b?'s toggle element and the transition signal for state v_3 is obtained by combining c? and the remaining output of b?'s toggle in a Muller c-element. The output p! is produced at states v_1 and v_2 and is always generated by a?. Output q! is generated at states v_0, v_1 and v_2 and hence is obtained by merging a? with the initial output of the toggle. And finally, output r! is only produced at state v_3 and therefore takes its output from the c-element.

$$
\begin{aligned}
E(v_0) &= \{(\{a\}, \{p, q\}, v_1)\} \\
E(v_1) &= \{(\{b\}, \{q\}, v_2)\} \\
E(v_2) &= \{(\{a\}, \{p, q\}, v_3)\} \\
E(v_3) &= \{(\{b, c\}, \{r\}, v_1)\}
\end{aligned}
$$

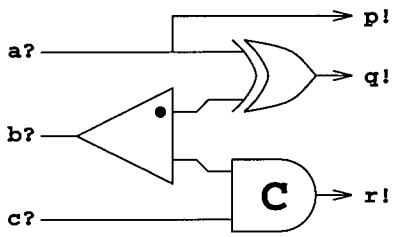


Figure 5–7: Sequential circuit example

Although this synthesis method is applicable to all general sequential circuits, there are large numbers of optimizations applicable to the implementation of sequential circuits where $|I| > 1$. These can result in a significant reduction in the amount of synchronization and state holding circuitry required to implement sequential circuits that meet the required criteria. These optimization techniques are described in chapter 6.

5.2 Non-Concurrent Circuits

A more general class of delay insensitive circuits than strictly sequential circuits is the class of *non-concurrent circuits*. Whereas sequential circuits are distinguished by having at most one transition from each stable state, non-concurrent circuits may have multiple transitions from each stable state, provided these transitions do not occur concurrently, i.e. they must be both distinguishable and mutually exclusive. This condition is formally characterized by the condition $\mathbf{cause}(t_1) \subseteq \mathbf{cause}(t_2) \Rightarrow t_1 = t_2$ for any pair of transitions $t_1, t_2 \in E(v)$ from a stable state v. By definition, this class also contains all sequential circuits.

The mutual exclusive nature of stable state transitions makes non-concurrent circuits an interesting class for circuit synthesis, permitting much simpler decom-

position than concurrent circuits such as those with non-deterministic behaviour. Because the environment of a non-concurrent circuit may choose which inputs to send at points in its behaviour, this class is a subset of Udding's $\mathbf{C}_2$ data communication class. However, because some Udding synchronization class behaviours are concurrent, they are not a superset of $\mathbf{C}_1$.

The general decomposition strategy for non-concurrent circuits is to divide the implementation of the circuit into three levels. These levels form the input routing circuitry, the synchronization circuitry and the output generation circuitry as described in chapter 4. Each input is fed into a routing circuit which generates an input instance corresponding to each stable state, these signals are then 'synchronized' to generate a single unique state transition signal from that state, which is used to update the state of the routing circuitry and generate the appropriate set of outputs by branching merging. The principal differences from the model of sequential circuits is that the synchronization circuitry may now be more complex than a single completion tree, and the stable state transition signal may now be required to update circuit state. This is because the 'next' state may depend upon which state transition fires, rather than being uniquely determined by the circuit's specification. The single unique stable state transition property makes the 'committee problem' much easier to deal with for generating the synchronization layer. It also makes synthesis of the routing layer much easier as the next input transition and all state change transitions are guaranteed to be non-concurrent. This means that there is no need for arbitration in the routing layer.

5.2.1 Non-Concurrent Routing Synthesis

General Implementation Strategy

This section considers the implementation of the routing layer circuitry for non-concurrent circuits. The routing layer of the proposed circuit synthesis methodology is responsible for converting an external input signal to one of n internal input signal instances where n is that number of stable states at which that circuit appears. Obviously an input signal that only ever occurs at a single stable state does not need any routing circuitry, and may be implemented as a single wire connecting the external input to the appropriate internal signal instance.

As described in section 5.1, the next stable state of a sequential circuit at which a given input signal occurs is always known from the circuit's behavioural specific-

ation. Hence, the routing circuitry for each input signal is able to determine for itself the next internal signal instance to generate in response to a transition on its external input. Such autonomy is generally not possible with non-sequential circuits. A routing layer circuit for an input to a sequential circuit, as designed above, would always have just a single input and n outputs, one for each stable state at which that input arrives. A routing circuit for an input to a non-concurrent circuit may require up to an additional n inputs and n outputs. These are organized as request acknowledge pairs, and set the state of the routing layer so that the next signal instance to be generated appears at the specified instance output terminal. The actual routing layer circuit for a non-concurrent circuit input may be implemented by a single general n state select element. The external input is connected to the select element test input, the n test output signals then form the internal input signal instances. The n set and set-acknowledge terminals then form the handshaking pairs for setting the state of the circuitry.

The proposed circuit methodology uses these additional terminals to change the state of each input using stable state transition signals. State transition signals are generated by synchronization layer circuits to indicate which of the possible state transitions has 'fired'. Before this signal is used to generate the outputs (effect set) associated with the appropriate state transition, it is used to update any internal routing layer state ready for the next stable state. As an initial implementation strategy, each transition signal is used to update the routing circuitry of all states that occur at the destination state. As described in section 6.1.8, the routing circuits of these inputs can either be updated sequentially or in parallel. Assuming the state changes occur sequentially, the state transition signal is fed into the appropriate set state input of the first routing circuit. The acknowledge output associated with this input is used to set the appropriate state of the next routing circuit, and so on, until the state change acknowledge of the final routing circuit is used to generate the state transition's effect set.

If more than one transition has a destination stable state that requires routing circuitry to be updated, i.e. has inputs that also occur at other stable states, then additional hardware needs to be used to share the state set and acknowledge terminals. This additional hardware consists of an n-call component, where n is the fan-in of the destination stable state, which multiplexes the state transition signals, triggers the updating of all the routing circuits (sequentially) and then generates a unique acknowledge for each transition signal. These unique acknowledges are

then used to generate the effect set outputs of each stable state transition. It is assumed that the routing circuitry of all inputs occurring at the initial stable state vertex, v_0, is initialized to its initial signal instance state. One further refinement is to notice that reflexive stable state transitions, i.e. those transitions whose destination state is the same as their source state, need not update the internal state of any input routing circuitry. This is because the routing circuits of all inputs occurring at that stable state are already set to the desired internal state.

As an example application of these synthesis steps, consider the stable state graph specification given in figure 5–8 below. This behaviour describes a non-concurrent circuit with three stable states and six stable state transitions. The circuit's interface has three inputs, $a?$, $b?$ and $c?$, and six outputs, one for each of the stable state transitions, $p!$, $q!$, $r!$, $s!$, $t!$ and $u!$.

$$\begin{aligned} E(v_0) &= \{(\{a\},\{p\},v_1),\ (\{b\},\{q\},v_0)\} \\ E(v_1) &= \{(\{b\},\{r\},v_1),\ (\{c\},\{s\},v_2)\} \\ E(v_2) &= \{(\{a\},\{t\},v_0),\ (\{c\},\{u\},v_1)\} \end{aligned}$$

Figure 5–8: Non-concurrent routing example SSG

Analysis of the above stable state graph reveals that all three input signals occur at exactly two stable states; $a?$ occurs at v_0 and v_2, $b?$ occurs at v_0 and v_1 and $c?$ occurs at v_1 and v_2. This indicates that the routing circuits of all three inputs will consist of a 2-select component (a standard Keller select element). Assuming that Keller select elements are initially reset, the $a?$ and $b?$ instances at v_0 are generated from the appropriate `T0!` Keller select outputs as both of these inputs occur at the initial state. A design decision is made that the v_1 instance of $c?$ occurs at output `T0!` and the v_2 instance of $c?$ occurs at output `T1!` of the $c?$'s routing circuit. Further analysis reveals that transition q and transition r (using the convention that each transition is identified by its unique output) are both reflexive. This means that once these transition signals are generated they are not required to change the states of any routing circuitry. Of the remaining stable state transitions, transitions p and u move to state v_1 and both must therefore set $b?$'s select and reset $c?$'s select, transition s moves to state v_2 and must set both $a?$'s and $c?$'s select, and finally transition t moves to state v_0 and must reset both $a?$'s select and $b?$'s select. In order for both p and u to update the same routing circuits to the same state they must be multiplexed using a standard two-input call

module. The resulting implementation generated using this decomposition method is given in figure 5–9 below.

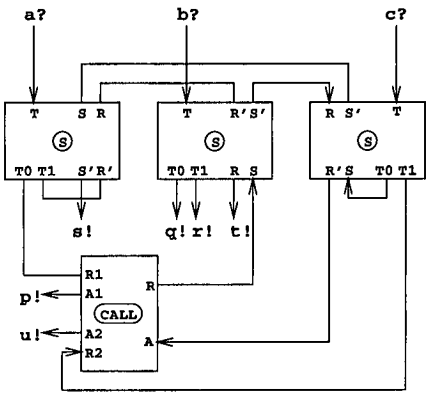


Figure 5–9: Non-concurrent routing example

Improved Implementation Strategy

From the above implementation strategy, it can be seen that each routing layer circuit is effectively an asynchronous finite state machine (FSM) implementing the appropriate input's signal instance graph (as defined in section 4.3).

The first optimization that may be applied to the implementation of non-concurrent routing is to observe that inputs with *sequential* signal instance graphs (SIGs) may be implemented by sequence generator circuits, as described in section 5.1.2. An SIG is sequential if at all vertices, all outgoing transitions have the same destination vertex. In this case, the next signal instance to be generated at any state is known, and the routing circuit may update its own state rather than rely on feedback from transition signals. In such a case, the implementation techniques described for the routing circuitry of sequential SSGs are applicable. As an example, consider the behaviour of the routing circuitry for signal $a?$ in the non-concurrent SSG specified in figure 5–8. The signal instance graph of this input has two states, where all transitions from the first lead to the second, and vice versa. This indicates that the routing circuitry for input $a?$ may be decomposed using the methods presented for cyclic sequence generators (section 5.1.2) and results in a single toggle component. The external input $a?$ is fed into the toggle's input and the instance of $a?$ at v_0 is generated at the initial output, and $a?$ at v_2 at the other output. The use of autonomous routing circuitry also simplifies the result of the

circuit, in the above example transition r need now only modify the state of $c?$'s routing circuit.

A similar optimization may be applied when only some of the vertices in an input's signal instance graph are sequential. In these cases, the routing circuitry is again able to update its own state, but just at the sequential vertices. This may result in a hardware saving if the destination vertex has a stable state transition fan-in greater than one. In such a case, a call component may be saved if this is the only input that needs to have its routing circuitry updated at that state.

It is also possible to implement some non-concurrent routing layer circuits using an $N\times1$ decision wait element, where N is the number of stable states at which the appropriate input occurs. The constraint on using a decision wait is that the external input signals and the state update signals must strictly alternate. This constraint is satisfied if all the states of the stable graph, (V,T), at which a signal i can occur, it must occur. This is expressed formally as the condition $\forall v \in V$ if $\mathbf{Occurs}(i,v)$ then $\forall t \in E(v) \quad i \in \mathbf{cause}(t)$. This is a weaker condition than sequentiality. If this condition holds, the routing circuit may be implemented using the external input i as the single row input, and generating the N internal signal instances from the decision wait outputs. The request-acknowledge pair state setting signals are implemented by forking each set signal, one branch forming a column input, the other producing the acknowledge. The acknowledge output may be generated immediately as the decision wait forces any subsequent input signal to wait until the next state signal arrives. The major advantage of decision wait routing circuitry is that call elements are not required to multiplex state update signals. Instead the transition signals are simply forked and merged with the merge output forming the decision wait column input. Another subtle difference is that reflexive stable state transitions must update the decision wait routing circuits of all the inputs in their cause sets.

Select Tree State Assignment

As described in section 3.4.4, a general N-input select element may be implemented using an $N\times2$ decision wait element, an $N\times1$ decision wait element and two N-input merge gates. This permits any non-concurrent routing circuit to be implemented in a decision wait basis. However, as will be described below, there are a number of optimizations that may be applied if routing circuits are decomposed in a Keller select element basis.

$$
\begin{aligned}
E(v_0) &= \{(\{a,b\},\{p\},v_1),\\
&\quad\ (\{a,c\},\{q\},v_0)\}\\
E(v_1) &= \{(\{a\},\{r\},v_0)\}
\end{aligned}
$$

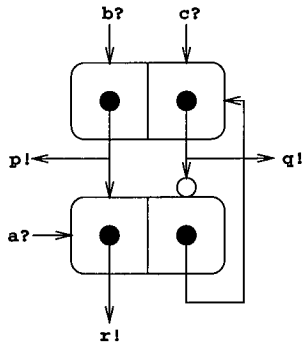


Figure 5–10: Decision wait routing layer implementation

For a select element implementation, an N state input routing circuit is organized as a central binary tree of Keller select elements, surrounded by a number of general call components to multiplex the state change signals (as described in section 3.4.4). The $N-1$ select elements are arranged as a tree where each test output, `T0!` and `T1!`, is either connected to the test input, `T?`, of another select element or forms one of the N instance signals. The external input is then used as the test input to the root select element of this tree. Note this tree need not be balanced, and the tree topology represents a state encoding. A binary vector may be used to represent each instance signal by concatenating the states of the selects on the path from the root to the given instance signal output. Each state vector $\tilde{v_i}$ has a minimum length of 1 and a maximum length $N-1$.

This select element topology may be used to reduce the amount of hardware required to multiplex and demultiplex set/reset signals and reduce the number of select elements that need change their states.

The first point to notice is that changing state from a state $\tilde{v_i}$ to a state $\tilde{v_j}$, requires setting all the selects of v_j that are not part of their common prefix. This means that if a state encoding is found that maximizes the common prefix between the source and destination states of a stable state transition, this will require less circuitry to be updated. The second point to notice is that in typical operation some state transitions are known to occur more frequently than others. Hence the performance of the resulting circuit may be improved by speeding up the commonly occurring transitions at the expense of the rare ones. Optimizing these constraints is the goal of *state assignment.*

State assignment is the task of finding a mapping from signal instances to state vectors (the select element topology) so as to minimize the required 'call' hardware and maximize performance. The quality of a state assignment may be judged on

three criteria. The first is the number of additional Call elements necessary to implement the state transitions, the second is the number of select element tests required to determine a signal's instance (biased by instance frequency) and finally, the number of select element state changes required to perform a state transition (biased by state transition frequency). The relative priorities of these criteria depends upon the additional hardware 'cost' of additional Call elements and the relative performance of modifying to testing a Keller select elements.

The algorithm currently used to determine a state assignment is to iteratively group sets of states together using heuristics. Initially, the N states of a given routing circuit are associated with an empty state vector (of length zero) and organized as N distinct sets each containing a single state. The state assignment proceeds by selecting two sets of states and combining them to form a single set, prefixing the vectors of all the states in one set by zero and all the vectors of the other set by one. The choice of which set is prefixed by a given value is arbitrary unless the speed of a select element is dependent upon its state. This step is applied N-1 times with the resultant and remaining sets until only a single set of states remain. This technique can generate any possible state assignment, the quality of which is dependent upon the selection criteria (heuristics) used to identify the two state sets to combine. Four heuristics have been implemented in a prototype circuit compiler, these (ordered by priority) are listed below.

1. **Maximize Internal Communication.** When combining vertices together, preference is given to strongly connected pairs of vertices. This means that state transitions tend to be kept locally in the same subtrees. Because a transition between two states changes all the select elements up to their common ancestor, keeping related vertices in the same subtrees improves performance.

2. **Minimize Incoming Transitions.** The total of hardware required by call trees to set or reset a Keller select element is proportional to the number of signals that need to be combined. This heuristic attempts to minimize this overhead. The other advantage is that it tends to encode the initial states that have no incoming transitions deep in the tree. This heuristic, however, is in direct conflict with the first heuristic.

3. **Minimize Total Depth.** This heuristic works by Huffman encoding the vertices into the select tree. In order to make the select tree as balanced/low as possible, this heuristic gives preference to combining shallow trees.

4. **Maximize Incoming Depth.** This final low priority heuristic is applied if all other factors are equal. It is used as a packing constraint to avoid imbalance in the select trees.

State Transition Selection

In the above synthesis strategy, it was assumed that every stable state transition would update the routing circuitry state of those input signals that occur at its destination stable state. In fact, the only requirement on the routing circuitry is that it is in the appropriate state when the external input signal arrives. Hence, any of the transition signals that fire between one instance of an input and the next may be used to appropriately update the routing circuitry. Indeed the state may be updated several times between instances provided that it is in the correct state when the next external input arrives. The task of determining which transition signals to use to update a routing circuits state is called *state transition selection.*

Alternative algorithms for state transition selection are not described in this thesis. The use of the final state transition before each stable state, as described above, suffices to demonstrate the circuit synthesis techniques described in this thesis. However, alternative state transitions, that may require less hardware, are possible through analysis of the signal instance graph (see section 4.3) of each external input signal.

Equivalent Instances

One final set of non-concurrent routing circuit optimizations are those based on signal equivalences. Informally two signals i and j are equivalent if they have the same observable effect on the circuit, written **equivalent**(i, j). The circuit synthesis method can make of the knowledge that two signals are equivalent to reduce the amount of hardware required by an implementation. For example, a call element need not be used to demultiplex a request into two equivalent signals. Because they both have the same effect on the rest of the circuit, they are indistinguishable and transitioning either will preserve the behaviour of a circuit.

The easiest case of detecting equivalent signals in a circuit are those signals that are fed directly into a merge element without branching. There is no point in distinguishing two signals that will subsequently be merged. This is in fact an instance of one of the circuit identity optimizations described in section 6.1.2.

Consider the case of two stable state transition signals that have the same effect sets. Once the state of any routing circuitry has been updated, the resulting signals are equivalent as they are both just forked and merged with the same set of external output signals. If the above two 'equivalent' signals emerge from a single call element, because they update the same routing circuits to the same state, they need not be distinguished but can be merged together before being used to update the state of the routing circuitry, the acknowledge being used to generate the effect set outputs. This means that two transition signals that have the same effect sets and update the same routing circuitry to the same states are equivalent. Finally, routing circuitry need not maintain distinct states for two equivalent input signal instances.

5.2.2 Disjoint Transitions

The simplest form of synchronization circuitry for a non-concurrent delay insensitive circuit is that required to handle *disjoint* transitions. The transitions from a given stable state are termed disjoint if the cause sets of each transition share no input signals. Hence for any stable vertex v, the transitions are disjoint iff

$$\forall t_1, t_2 \in E(v) \quad \mathbf{cause}(t_1) \cap \mathbf{cause}(t_2) \neq \emptyset \Rightarrow t_1 = t_2$$

This restriction has the effect that the circuit's environment may only send to those input signals involved in a single transition's synchronization. This avoids the synchronization tree of any other state transition receiving a subset of its cause set. Each state transition's synchronization circuitry may therefore be implemented by a single completion tree of the instances in its cause set. Each completion tree is implemented by a binary tree of $|\mathbf{cause}(t)| - 1$ two-input Muller c-elements. The transition signals is then generated at the output of each tree.

For example, consider the non-concurrent behavioural specification described by the SSG in figure 5–11 below. This specification has only a single stable state and hence requires no routing circuitry as each input has only a single instance. The cause sets of the two transitions are disjoint and are implemented by two

c-elements using the construction described above. The specification also requires no output circuitry or state transition feedback signals and hence the c-element outputs form the circuit's outputs.

$$E(v_0) = \{(\{a,b\},\{p\},v_0), (\{c,d\},\{q\},v_0)\}$$

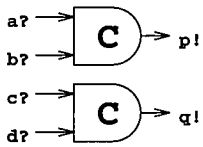


Figure 5–11: Disjoint transition example

5.2.3 Distinct Transitions

The largest class of non-concurrent circuits is those with *distinct transitions.* The only requirement on this class of circuits is the non-concurrency requirement for each stable state v, that the cause set of any state transition is not a subset of any other. This class of circuit behaviour requires extra circuitry to *roll-back* partially committed synchronization trees. The delay insensitivity constraints on a Muller c-element require that once one input is asserted it may not be retracted. Hence during a state transition, any c-element that receives only a subset of its inputs must be 'reset' to its initial state. This requires additional logic to generate inputs on each of the remaining inputs and to manage the outputs from each synchronization tree. This additional hardware has been termed 'forgetting' circuitry by several researchers. The class of distinct transitions is therefore characterized by the following condition.

$$\forall e_1, e_2 \in E(v) \quad \mathbf{cause}(e_1) \subseteq \mathbf{cause}(e_2) \Rightarrow e_1 = e_2$$

For disjoint transitions, each synchronization circuit consists of a single completion tree of $|\mathbf{cause}(t)|-1|$ c-elements. For non-disjoint transitions, each synchronization circuit also requires a number of merge gates and Keller select elements. The output of each completion tree, for a transition $t_1 \in E(v)$, is fed into the test input of an N-input initialized select component, where N is the number of transitions that have an input in common with t_1, i.e. $N = |\{t_2|\mathbf{cause}(t_1) \cap \mathbf{cause}(t_2) \neq \emptyset, \forall t_2 \in E(v)\}|$. For a stable state transition with distinct inputs, $N = 1$ and the N-input select is replaced by a wire generating the state transition input. Otherwise, $N \geq 2$, there are $N-1$ partially completed trees to be 'rolled back'. This

N-input select component is typically decomposed into $N-1$ standard 2-input Keller select elements.

When a completion tree 'fires', it tests the initialized select component at its output and generates one of the N-outputs. If a select component's initial output is generated, this is the first tree to fire and the required stable state transition. This 'initial' output signal is then used to set the state of the select components of the remaining $N-1$ partially committed completion trees, and provide the required 'stimuli' to cause them to fire. If the select component generates one of the remaining outputs, one of the other completion trees fired first, and caused this tree to rollback. This output is used to reset this select component to its initial state and acknowledge that this transition has been rolled back.

The false 'stimuli' required to fire a partially committed completion tree is formed by using a merge gate to combine the *rollback* signal with each of the inputs of the partially committed completion tree that are not in the cause set of the firing completion tree. Hence, for any pair of transitions $t_1, t_2 \in E(v)$, if $\mathbf{cause}(t_1) \cap \mathbf{cause}(t_2) \neq \emptyset$ then $|\mathbf{cause}(t_2) - \mathbf{cause}(t_1)|$ merge gates are required to stimulate t_2's completion tree after t_1 fires. This construction can lead to a very large number of merge gates, for complex stable states. In practice, however, this worst case bound can be significantly improved by applying a number of the optimizations described below.

There is an area/performance tradeoff that can be made when rolling back a number of completion trees. It is possible to rollback each partially committed tree either concurrently, sequentially or some combination of the two. Trees can be rolled back in parallel by forking the initial output of the firing transition's select component to form $N-1$ branches, one for each tree to be rolled back. Each branch is then used to set the state of the chosen transition's select component to indicate which transition fired. The set acknowledge is then forked to produce one rollback signal for each input to be stimulated, which is merged with that input to form the completion tree's inputs. This then excites the completion tree and generates the rollback acknowledge output. All $N-1$ rollback acknowledge outputs generated by the initial firing are then combined with an $N-1$ input c-element which generates the required state transition output. Alternatively, trees can be rolled back sequentially, using the rollback acknowledge of each tree to rollback the next (in an arbitrary order) and the rollback acknowledge of the last tree forming the state transition signal. This tradeoff is discussed in detail in section 6.1.8.

Consider the example specification given in figure 5–12 of a 2×1 decision wait element. This has two transitions that are not disjoint from a single stable state, the input a? is common to the cause sets of both transitions. Hence, once the completion tree for either state transition 'fires', the remaining tree is a partially completed state. Each c-element requires a single select element to differentiate between a true firing and a rollback initiated by the other c-element.

$$E(v_0) = \{(\{a,b\},\{p\},v_0), (\{a,c\},\{q\},v_0)\}$$

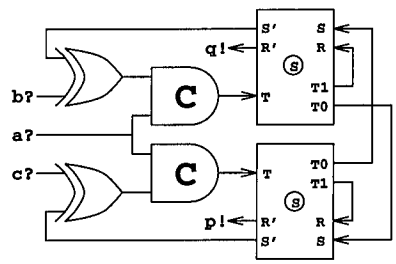


Figure 5–12: Distinct transition rollback example

The complex behaviour of the above example deserves a little more explanation. Consider the arrival of the two inputs signals $a?$ and $b?$ that are required for the first stable state transition to fire. The $a?$ input is forked and partially commits (primes) both Muller c-elements. The $b?$ input passes straight through the merge gate and fires the remaining input of the first c-element. At this point in the behaviour of this circuit, one of the c-elements will fire and the other will have a single $a?$ input commited. The correct delay-insensitive behaviour of the c-element requires that the remaining input on this second c-element must fire before the circuit can return to its initial state. Hence the select elements that appear after each c-element indicate whether this is the first firing of the c-element (the stable state transition signal) or the result of the other c-element firing first and attempting to reset/rollback this c-element. To continue our example, $a?©b?$ is used to test the top select element, which by convention is reset. The `T0!` output of this select indicates that the bottom c-element needs to be rolled back. This output is used to set the state of the bottom select element, the set-acknowledge is then merged with $c?$ to fake a pseudo-input into the bottom c-element. As the c-element has now received both $a?$ and a fake $c?$, it fires and the output $a?©c?$ is used to test the bottom c-element. This time the state is set, indicating that the c-element is being reset, the `T1!` output is then used to reset this bottom select element and the reset acknowledge is used to to generate the output signal $p!$ indicating that the stable state transition has completed. It can be seen that the circuit is symmetric and the behaviour of the second stable state transition is analogous to the first.

There are a number of optimizations that can be applied to this general synchronization level construction. Perhaps, the most powerful is the partitioned transitions optimization that is discussed in the following section. However, this optimization is only applicable to a particular class of behavioural specifications. Some delay insensitive circuits, such as n-of-m code detectors require rollback circuitry.

The two remaining 'rollback' implementation optimizations take advantage of the fact that not every c-element in a completion tree may have been committed. The first optimization is to notice that the false stimuli need not be inserted at the base (leaves) of a completion tree, but may be introduced in the middle of the completion tree. This optimization is a special case of the general 'common subexpression elimination' optimization described in section 6.1.3. The principle is illustrated by the circuits shown in figure 6–5 on page 156. As part of a larger completion tree, inputs `a?` and `b?` need to be rolled back by a false stimulus `c?`, this stimulus may be inserted higher in the completion tree. The other optimization is to notice that the select element need not always be placed at the root of a completion tree. It is possible to rollback only a subtree of the completion tree and place the select element at the root of this subtree. This removes the need to stimulate the inputs of the completion tree that are not part of the subtree.

Both of these optimizations are demonstrated in the circuit implementation given in figure 5–13 below. As can be seen from the stable state graph specification, this function of this circuit is symmetric with b, c and d, e being symmetrical. For the purposes of illustration, however, the implementation is asymmetric showing the two different optimization strategies described above. The completion tree of the first (top) transition is organized as $(a©b)©c$ and the completion tree for the second transition is organized $(d©e)©a$. Both completion trees need to be able to rollback the `a?` input signal. The top implementation makes use of the fact that the select element need not be placed at the root, but instead only rolls back the $a©b$ subtree, by inserting a false stimulus to b. The lower implementation makes use of the fact that the false stimulus may be inserted above all non-committed inputs in the tree, and hence rolls back the top c-element by inserting a false stimulus at $d©e$.

$$E(v_0) = \{(\{a,b,c\},\{p\},v_0), (\{a,d,e\},\{q\},v_0)\}$$

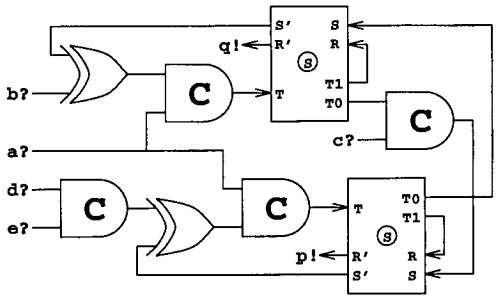


Figure 5–13: Distinct transition partial rollback example

5.2.4 Partitioned Transitions

An extremely useful decomposition technique for non-concurrent circuits is the *partitioned transitions* optimization. This optimization improves the quality of synchronization circuitry by avoiding the need for synchronization rollback (forgetting circuitry), as described in the previous section. This decomposition method may be applied when a set of mutually exclusive input signals can be found, such that one signal from this set occurs in each transition's cause set. This condition may be expressed formally by the statement below.

$$\exists x \subseteq I : \forall e \in E(v) \quad |x \cap \mathbf{cause}(e)| = 1$$

This input set, x, effectively partitions the transitions from a stable state into a number of discrete sets, one for each input signal, hence the name of the optimization.

Partition Signal

The obvious special case of this condition occurs when one signal occurs in the cause sets of all the transitions, i.e. $|x| = 1$. When this is the case, none of the transitions is allowed to fire until that single signal, $i \in x$, has arrived. This form of stable state may be decomposed by implementing the stable state formed by removing i from each cause set. The outputs, transition signals, from each of these n transitions may be fed into an $n \times 1$ decision-wait, whose single column input is the enabling input i. The n outputs of this decision wait then form the true outputs of the stable state. Alternatively, the $n \times 1$ decision wait may be implemented using an n-call component and a two-input Muller c-element.

A example application of this transformation to the specification of a 2×1 decision wait is shown in figure 5–14 below. Here the input signal $a?$ occurs in both of the stable state transitions. Once this input signal is removed, the two cause sets of the transitions become $\{b\}$ and $\{c\}$. These transitions are both trivially implemented by wires. The resulting implementation is then these wires fed directly into a 2×1-decision wait whose outputs are the required state transition signals. This is large improvement above the synchronization rollback implementation given in figure 5–12.

$$E(v_0) = \{(\{a,b\},\{p\},v_0), (\{a,c\},\{q\},v_0)\}$$

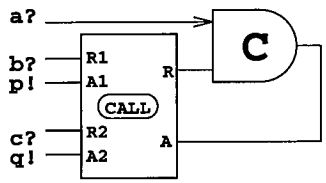


Figure 5–14: Partitioned transition example

A slightly more complex example is the SSG specification shown in figure 5–15 below. Once again this stable state has two transitions, both of which contain the single input signal $a?$ in their cause sets. The reduced synthesis task now becomes the cause sets $\{b,c\}$ and $\{d,e\}$. These can be seen as disjoint transitions and implemented as a pair of 2-input c-elements. With the substitution of wires for c-element outputs, the complete circuit is almost the same as the one presented above. As can be seen by comparison with figure 5–13, this circuit is once again an improvement upon the synchronization rollback implementation.

$$E(v_0) = \{(\{a,b,c\},\{p\},v_0), (\{a,d,e\},\{q\},v_0)\}$$

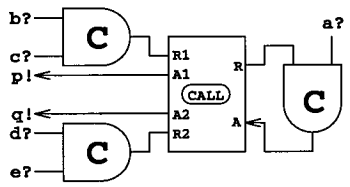


Figure 5–15: Partitioned transition example

Partition Set

When the partitioning set x contains more than a single input signal, a similar but more complex decomposition strategy can be applied, provided that we place a constraint upon the partitioning set x. This simple constraint requires that after the removal of the appropriate symbol from the cause sets, that no cause set is a subset of another. This constraint may be expressed as

$$\forall t_1, t_2 \in E(v) \quad (\mathbf{cause}(t_1) - x) \not\subset (\mathbf{cause}(t_2) - x)$$

This case is always met when $|x| = 1$ for non-concurrent stable states. Note that in the above expression it is permitted to have one of the new cause sets equal another. Once a suitable partition is found, the stable state can be decomposed using a $M \times N$-decision wait element, where the number of column inputs M equals $|x|$ the number of signals in the partition set, and the number of row inputs N equals the number of unique new cause sets. Once again, the stable state formed by removing the appropriate member of x from each cause set is implemented, and the N unique outputs fed into the row inputs of the decision-wait element. Similarly, each of the M members of x is fed into the column inputs. The output transition signals are then generated from the appropriate cell in the decision wait matrix, i.e. the transition signal for transition $t \in E(v)$ is generated from the column corresponding to $\mathbf{cause}(t) \cap x$ and the row corresponding to $\mathbf{cause}(t) - x$.

An application of this optimization is the decomposition of a 2×2 decision wait element. The stable state graph specification of this component is given in figure 5–16 below. This specification has 4 inputs and 4 outputs. The two row inputs are labelled `a0?` and `a1?` and the two column inputs are labelled `b0?` and `b1?`. The four outputs, `c00!`, `c01!`, `c10!` and `c11!`, are generated upon receiving one row input and one column input. The arrival of $a_i?$ and b_j results in the generation of $c_{ij}!$.

$$\begin{aligned} E(v_0) = \{&(\{a0, b0\}, \{c00\}, v_0), \\ &(\{a0, b1\}, \{c01\}, v_0), \\ &(\{a1, b0\}, \{c10\}, v_0), \\ &(\{a1, b1\}, \{c11\}, v_0)\} \end{aligned}$$

Figure 5–16: 2×2 decision wait SSG

A suitable partition set for this stable state is the set $\{a0?, a1?\}$ as each signal occurs once in each transition's cause set. Upon removing the partition signals, the there are only two unique cause sets $\{b0?\}$ and $\{b1?\}$. Obviously from these two sets, the subset condition is met and hence the partitioned transitions decomposition may be applied. Both reduced cause sets only contain a single input

signal, and hence are both implemented by wires. This then results in the final implementation being a 2×2-decision wait with the wires from $b0?$ and $b1?$ as the row inputs and $a0?$ and $a1?$ as the column inputs. Using the Keller select element basis, the resulting implementation of the above SSG specification is the circuit shown in figure 5–17.

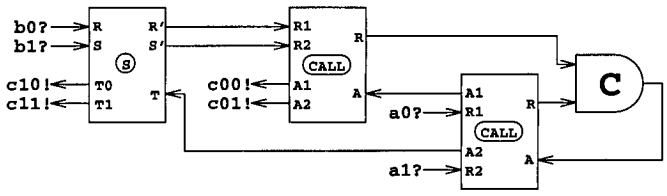


Figure 5–17: 2×2 decision wait implementation

Because the decision waits resulting from this implementation method are often sparsely populated, they can often be optimized further using row/column elimination (section 6.1.5), row/column compression (section 6.1.6) and decision wait splitting (section 6.1.7).

5.2.5 Synchronization Decomposition

There are a number of optimizations to the generic 'distinct' transitions implementation strategy described above that help to simplify the task of generating synchronization layer circuits.

The first simplification is that the committee problem may be subdivided into smaller problems, provided that each smaller problem does not contain any members (input signals) in common. If it is possible to find a set of transitions, $E \subset E(v)$ from a stable state v, such that for any transition $t_1 \in E$ and any transition $t_2 \in E(v) - E$, then $\mathbf{cause}(t_1) \cap \mathbf{cause}(t_2) = \emptyset$, then the stable state may be implemented by parallel composition of the implementation of the stable states (transition sets) defined by E and $E(v) - E$.

The next property is that any set of input signals that only ever occur together in the causes sets of the transitions at a stable state, may be replaced by a single input signal, that is generated by synchronizing all of those inputs together with a c-element. [This is very similar to the global input clustering described in section 6.2.1].

In non-concurrent circuits, any transition whose cause set contains only a single input signal may be implemented by a single wire, where the transition signal for that transition is generated from the single input. In non-concurrent circuits, any input signal that only occurs in a single transition's cause set, where that cause set with the signal removed is not a subset of any other transition, may be removed from the cause set and combined with the resulting state transition signal using a Muller c-element.

The final optimization is dependent upon transitions that have the same result on the state of the circuit and the same generated output signals. Typically, this is the case for transitions t_1 and t_2, if they have the same effect sets, $\mathbf{effect}(t_1) = \mathbf{effect}(t_2)$ and $\mathbf{next}(t_1) = \mathbf{next}(t_2)$. However, the condition is neither necessary nor sufficient for some routing layer synthesis strategies. The result of a transition on a circuit is not necessarily determined by its next state, but by whether and how it is used to update the routing layer circuitry. This is explained in greater detail in the discussion on transition signal selection, which is covered in section 5.2.1. If this criterion is met, the fact that the two transitions t_1 and t_2 have the same 'effect' or 'result' is denoted as $\mathbf{equivalent}(t_1, t_2)$. Two transitions from the same stable state are not equivalent if they leave the circuit in different internal states, or generate different outputs.

The final complex optimization is termed *input merging*. Two input signals a and b that occur at a stable state v may be merged, if they do not occur in the same cause sets, and for every cause set that one of them appears in, the other appears in an equivalent transition with an identical cause set except for the substitution of the first input for the second. Having identified such a pair of input signals, all equivalent transition pairs may be replaced by a single transition containing $a \oplus b$. The statement that the two inputs a and b never occur in the same cause set, may be expressed as $\forall t \in E(v) \,.\, |\mathbf{cause}(t) \cap \{a, b\}| \leq 1$. The fact that they occur in equivalent transition pairs may be expressed as $\forall t_1 \in E(v)$ if $a \in \mathbf{cause}(t_1)$ then $\exists t_2 \in E(v)$ such that $\mathbf{equivalent}(t_1, t_2)$ and $\mathbf{cause}(t_2) = (\mathbf{cause}(t_1) - \{a\}) \cup \{b\}$, and the symmetric condition for b.

An example application of this optimization is shown in figure 5–18 below. The top two stable state transitions in the given stable state graph are 'equivalent'. They both produce the same output, and because the SSG only has a single stable state there is no routing layer circuitry to be left in a different state. The two input signals `b?` and `c?` may be 'input merged' as they only occur in the

top two stable state transitions and the cause sets are otherwise identical. These top two stable state transitions can therefore be replaced by the single transition $(\{a, b \oplus c\}, \{p\}, v_0)$. The resulting stable state graph is equivalent to figure 5–14, and the resulting implementation is as shown below.

$$\begin{aligned} E(v_0) = \{&(\{a,b\},\{p\},v_0),\\ &(\{a,c\},\{p\},v_0),\\ &(\{a,d\},\{q\},v_0)\} \end{aligned}$$

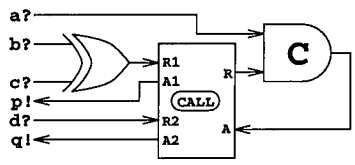


Figure 5–18: Input merge example

5.3 Concurrent Circuits

This section describes the decomposition of the remaining (most general) classes of delay insensitive circuits, *concurrent circuits*. Whereas state transitions of non-concurrent circuits cannot occur simultaneously, concurrent stable states may have transitions with identical cause sets or transitions whose causes sets are subsets of other cause sets. In addition to traditional concurrency, this section also describes the synthesis of non-deterministic circuits such as those that perform arbitration between signals.

One consequence of considering concurrent stable state graphs is that the behaviour of stable states is no longer independent. For two transitions t_1 and t_2 from stable state v such that $\mathbf{cause}(t_1) \subset \mathbf{cause}(t_2)$ there are several things that can be said about the stable state transitions from the stable state $\mathbf{next}(t_1)$. One property that is used through out the derivation of many of the synthesis rules described below is that $\forall t_1, t_2 \in E(v)$ if $\mathbf{cause}(t_1) \subset \mathbf{cause}(t_2)$ then $\exists t_3 \in E(\mathbf{next}(t_1))$ such that $(\mathbf{cause}(t_2) - \mathbf{cause}(t_1)) \subseteq \mathbf{cause}(t_3)$. Simply stated, if the environment can send more input signals than are required to fire a stable state transition, then these outstanding signals must be accepted in the destination stable state of that transition. This is effectively enforced by Udding's rule $\mathbf{R}_3$ in its different forms. One consequence is that both the current state $E(v)$ and the state $E(\mathbf{next}(t_1))$ must have circuitry to handle the arrival of the signals $\mathbf{cause}(t_2) - \mathbf{cause}(t_1)$. One driving principle in many of the synthesis methods introduced below is the ability to reuse the same hardware for implementing both stable states.

5.3.1 Static Non-determinism

The first class of concurrent circuits to be considered is those containing static non-deterministic stable states. Static non-determinism in a circuit specification occurs when the circuit may make an arbitrary choice between the outputs that it can generate and this non-determinism is not caused by 'racing' concurrent signals (dynamic non-determinism). Non-deterministic circuits may be considered a simple extension of non-concurrent circuits, where otherwise distinct, mutually exclusive stable state transitions may have identical cause sets and differing effect sets (or next states). This class of circuits may be defined formally by the constraint $\mathbf{cause}(t_1) \subseteq \mathbf{cause}(t_2) \Rightarrow \mathbf{cause}(t_1) = \mathbf{cause}(t_2)$, i.e. the only 'concurrency' in the circuit is caused by transitions with identical cause sets.

This implies that simple static non-determinism occurs when the same stimulus may have several potential responses, i.e. there exist two transitions $t_1, t_2 \in E(v)$ for any stable state v, such that $\mathbf{cause}(t_1) = \mathbf{cause}(t_2)$ and either $\mathbf{effect}(t_1) \neq \mathbf{effect}(t_2)$ or $\mathbf{next}(t_1) \neq \mathbf{next}(t_2)$. Such circuits may be implemented using the techniques described elsewhere in this thesis, replacing the statically non-deterministic transitions with one with the same cause set. Once the corresponding state transition signal is generated it may be used as input to an N-choice element where each of the N outputs form the stable transition signals for the original non-deterministic outputs. These state transition signals are then used to update the appropriate routing circuitry and generate the external output signals as discussed previously.

One of the useful properties of static non-determinism is that, with an appropriate implementation operator, the result of the non-deterministic choice may be made at compile time, i.e. when the circuit is designed or fabricated. Dynamic non-determinism, on the other hand, is an intrinsic characteristic of the 'run-time' behaviour of asynchronous circuits. Hence, a circuit specification that contains dynamic non-determinism must be implemented by a circuit containing an arbiter or similar dynamic component.

Taking advantage of the above 'alternative implementation' property of static non-determinism, a circuit compiler may attempt to generate several alternative deterministic (or dynamically non-deterministic) implementations and output the resulting implementation with the best performance/cost. Making such choices at design time can drastically reduce the amount of hardware required to synthesize

all potential non-deterministic behaviours. The resulting implementation remain observationally identical to the original circuit specification.

5.3.2 Premature Concurrency

An input signal i at a stable state v is said to exhibit *premature concurrency* if it may occur at v but the circuit does not acknowledge its receipt by generating any additional (or different) output signals. This can be tested by the following condition. $\forall t_1 \in E(v)$ such that $i \in \mathbf{cause}(t_1)$ then $\exists t_2 \in E(v)$ where $\mathbf{cause}(t_2) = \mathbf{cause}(t_1) - \{i\}$. The name 'premature' is due to the fact that such signals are received by the circuit before it is ready to process it.

An example of an SSG specification exhibiting premature concurrency is shown in figure 5–19 below. In this example, the input signal $a?$ at stable state v_0 is premature. The arrival (or not) of this signal has no effect on the behaviour of that stable state, the arrival of input $c?$ generates output $q!$ regardless.

$$
\begin{aligned}
E(v_0) &= \{(\{c\}, \{q\}, v_1), \\
&\quad\ (\{a, c\}, \{q\}, v_2)\} \\
E(v_1) &= \{(\{a, b\}, \{p\}, v_0)\} \\
E(v_2) &= \{(\{b\}, \{p\}, v_0)\}
\end{aligned}
$$

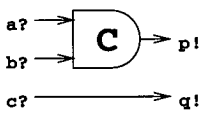


Figure 5–19: Premature concurrency example

The appropriate implementation strategy for such stable states is to ignore all stable state transitions in which the premature input occurs, but instead handle the arrival of that input in later stable states. In this example, input $a?$ is processed correctly in stable state v_1. This behavioural transformation relies on the routing circuitry for the premature input to be correctly updated before this state. This is trivially achieved in this example as the input $a?$ only occurs at a single stable state in the transformed stable state graph. The transformed stable state graph, i.e. the one that effectively needs to be implemented in shown in figure 5–20 below. Notice that this specification removes all stable states that are only reachable from deleted stable state transitions. The resulting circuit has only two stable states instead of the original four and requires no routing circuitry. The resulting implementation of this specification is given with the original SSG specification in figure 5–19 above.

$$E(v_0) = \{(\{c\},\{q\},v_1)\}$$
$$E(v_1) = \{(\{a,b\},\{p\},v_0)\}$$

Figure 5–20: Transformed premature concurrency SSG

5.3.3 Classical Concurrency

One of the most common classes of delay insensitive circuit specification is non-arbitrating concurrent circuits. However, correctly identifying the behavioural class of concurrent circuits is a complex task. The principal forms of concurrency classified in this thesis are static non-determinism, classical concurrency, simple concurrency, synchronization roll-back concurrency, complex concurrency and dynamic non-determinism (arbitration). The potential presence of combinations of any of these forms at a single stable state make their classification difficult.

The first step is to decompose a single specification stable state vertex into a number of independent transition sets (stable states) using the rules introduced in section 5.2.5. This results in several smaller stable states such that there must be an interdependence of the signals that can occur in that cluster. All premature concurrent signals (as described in section 5.3.2) are then removed from each stable transition. All stable state transition clusters that can immediately be classified as non-concurrent (possibly with static non-determinism as described in section 5.3.1) may be implemented independently using the decomposition strategies described previously in this chapter.

The second step in the decomposition is to classify the remaining stable state transitions as either *atomic* or *composite*, as defined in definition 26 in section 4.2. A stable state transition t from a stable state v is atomic if there are no transitions from v that have a cause set a subset of t_1's cause set, i.e. $\neg\exists t_2 \in E(v)\,.\,\mathbf{cause}(t_2) \subset \mathbf{cause}(t_1)$. Any transition that is not atomic is classified as composite.

Classical concurrency occurs when the stable state transitions in a given transition set T may be partitioned into two independent transition sets, T_1, T_2 and a set of composite state transitions, T_3 that occur due to the simultaneous firing of the transitions in T_1 and T_2. The partitioning constraint may be expressed as the conditions $\forall 0{<}i{\leq}3\,.\,T_i \subset T$, $\forall 0{<}j{\leq}3\,.\,T_i \cap T_j = \emptyset$ and $T_1 \cup T_2 \cup T_3 = T$. The independence of inputs occurring in T_1 and T_2 may be expressed as the condition

that $(\bigcup_{t\in T_1} \mathbf{cause}(t)) \cap (\bigcup_{t\in T_2} \mathbf{cause}(t)) = \emptyset$. The composite constraint on T_3 is expressed as $\forall t \in T_3 \,.\, \mathbf{composite}(t)$.

The simultaneous independent firing of T_1 and T_2 may be expressed by the following set of conditions. For every transition $t_3 \in T_3$, there exists $t_1 \in T_1$ and $t_2 \in T_2$ such that $\mathbf{cause}(t_1) \cup \mathbf{cause}(t_2) = \mathbf{cause}(t_3)$, $\mathbf{effect}(t_1) \cap \mathbf{effect}(t_2) = \emptyset$ and $\mathbf{effect}(t_1) \cup \mathbf{effect}(t_2) = \mathbf{effect}(t_3)$. Finally in the case that t_1 and/or t_2 are composite, the subsets must also be independent and simultaneous: $\forall t_4 \in T_i$ such that $\mathbf{cause}(t) \subset \mathbf{cause}(t_i)$ then $\exists t_5 \in T_3$ such that $\mathbf{cause}(t_5) = \mathbf{cause}(t) \cup \mathbf{cause}(t_j)$ and $\mathbf{effect}(t_5) = \mathbf{effect}(t) \cup \mathbf{effect}(t_j)$ and $\mathbf{effect}(t) \cap \mathbf{effect}(t_j) = \emptyset$.

These conditions may at first seem cryptic but allow the partitioning of many of the forms of concurrency that can occur at a stable state. For example, consider the stable state specification given in figure 5–21. This specification describes the behaviour of two concurrent wires, which satisfies the conditions described above. This specification may be partitioned into the two independent 'wire' stable state transitions, and the third transition that is the composite behaviour of both wires firing simultaneously. In this case, the recognition of the stable state behaviour as being classically concurrent allows the two independent partions (wires) to be implemented independently.

$$\begin{aligned} E(v_0) \;=\; & \{(\{a\},\{p\},v_0), \\ & (\{b\},\{q\},v_0), \\ & (\{a,b\},\{p,q\},v_0)\} \end{aligned}$$

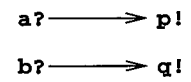


Figure 5–21: Classical concurrency example

However the portion of a stable state graph specification given in figure 5–22 is not separable by classical concurrency due to the repeated output signal. This signal must be implemented using the general arbitration method described at the end of the chapter.

Given that a stable state may be partitioned into classically concurrent signals, the resulting implementation may be fabricated as separate synchronization layer subcircuits. The only potential hazard occurs if transition signals from different concurrent synchronization subcircuits are required to update the routing layer circuitry of the same input. In this case, a SEQ element is required to enforce mutual exclusion of the updating of the routing circuit.

$$\begin{aligned} E(v_0) = \{&(\{a\},\{p\},v_0),\\ &(\{b\},\{p\},v_0),\\ &(\{a,b\},\{p,q\},v_0)\} \end{aligned}$$

Figure 5–22: Classical concurrency counter example

Altough this property is not formally proved here, a simple justification for it exists. The conditions above are sufficient to create two partitions of the stable state transitions that contain no common input or output signals. With the constraint that the circuit implementing one partiton need not update the other's routing layer, these two circuits are completely independent with no communication between them. Similarly each of these circuits is completely specified by a stable state whose only stable state transitions are those in its partition. We now consider the behaviour of the parallel composition of these two independent subcircuits using the defintion given in section 2.3. The concurrent firing of stable state transitions in each subcircuit results in "observable" stable state transitions that have causes sets that are the union of the subcircuit transition's cause sets and effect sets that are the union of the subcircuit transition's effect sets. These composite transitions form a superset of the composite partition described above.

5.3.4 Simple Concurrency

A simple non-classical form of concurrency in delay insensitive circuits occurs when a set of input signals is synchronized, but a set of outputs may be generated by synchronization of a partial subset of these signals. Behaviours in Udding's $\mathbf{C}_1$ synchronization class of circuits are typically sequential, because at each stable state there is a fixed set of inputs that must eventually be received. However, a circuit may be permitted to generate a set of output signals once a subset of this fixed input set has been received. Hence the stable state graph of a synchronization class component may have several outgoing transitions from a stable state. For a behaviour to belong to Udding's synchronization class, there must exist a 'maximal' transition $t_{max} \in E(v)$ from any stable vertex v, such that for any transition $t \in E(v)$, $\mathbf{cause}(t) \subseteq \mathbf{cause}(t_{max})$.

For the time being we place a strange constraint on concurrent behaviours in that for any three transitions $t_1, t_2, t_3 \in E(v)$ from a stable state v, such that

$\mathbf{cause}(t_1) \subset \mathbf{cause}(t_3)$, $\mathbf{cause}(t_2) \subset \mathbf{cause}(t_3)$ and $\mathbf{cause}(t_1) \cap \mathbf{cause}(t_2) = \emptyset$ then $\mathbf{effect}(t_1) \cap \mathbf{effect}(t_2) = \emptyset$. We shall call this constraint the *Verhoeff condition* which is covered in detail in section 5.3.6.

Given this condition, the most restricted form of concurrency occurs when after a 'non-maximal' partial synchronization has occurred, the only inputs that may occur are those necessary to complete the synchronization. This condition is called *simple concurrency*. This is when for any transition $t_1 \in E(v)$ from a stable state v such that $\mathbf{cause}(t_1) \subset \mathbf{cause}(t_{max})$ where t_{max} is the maximal transition from v, then $\mathbf{next}(t_1)$ has a maximal transition t_2 with cause set $\mathbf{cause}(t_{max}) - \mathbf{cause}(t_1)$.

An example of an SSG specification with this property is given in figure 5–23 below. In this specification, the stable state v_0 has a maximal transition $\{a, b\}$, but the circuit is able to generate outputs after the receipt of the signals $a?$ and $b?$. Simple concurrency circuit specifications may be implemented very easily. As shown in the implementation of the above example, only the maximal transition from the initial 'simple concurrent' stable state need be physically implemented. All the remaining subset stable state transitions and their reachable stable states are implemented by forking the appropriate signal in the synchronization tree. This implementation strategy may effectively remove a large number of stable states from a specification SSG and result in much smaller routing circuitry. For example, the above circuit specification has three stable states, but only one needs to be implemented physically. Hence the routing circuitry for $a?$ and $b?$ may be trivially implemented by wires rather than Keller select elements.

$$
\begin{aligned}
E(v_0) &= \{(\{a\}, \{p\}, v_1), (\{b\}, \{q\}, v_2), \\
&\quad\ (\{a, b\}, \{p, q, r\}, v_0)\} \\
E(v_1) &= \{(\{b\}, \{q, r\}, v_0)\} \\
E(v_2) &= \{(\{a\}, \{p, r\}, v_0)\}
\end{aligned}
$$

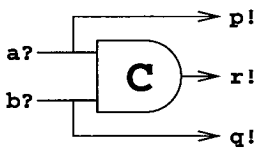


Figure 5–23: Simple concurrency example

This decomposition strategy even applies when the subset stable state transitions share input signals in common. A more complex example of simple concurrency is shown in figure 5–24. Such common signals require multiple instances of individual input signals in the same synchronization tree. For example, the input $b?$ in the current example needs to be forked to allow completion trees for both $\{a, b\}$

and $\{b, c\}$. In the general case, simple concurrent circuits may be implemented by a number of c-elements, one for each transition signal.

$$
\begin{aligned}
E(v_0) &= \{(\{a,b\},\{p\},v_1),(\{b,c\},\{r\},v_2),\\
&\quad\ (\{a,b,c\},\{p,q,r\},v_0)\}\\
E(v_1) &= \{(\{c\},\{q,r\},v_0)\}\\
E(v_2) &= \{(\{a\},\{p,q\},v_0)\}
\end{aligned}
$$

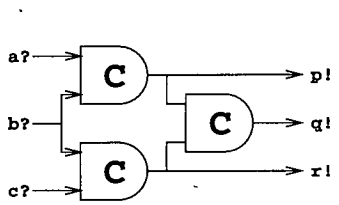


Figure 5–24: Simple concurrency example

5.3.5 Synchronization Rollback

In the above class of non-classical concurrent behaviours, the maximal transition t_{max} is always guaranteed to eventually fire by the circuit's behavioural specification. However, a similar implementation strategy may be applied even when a central synchronization tree is not guaranteed to complete. This may be achieved using the *synchronization rollback* techniques first introduced in section 5.2.3.

A good example of this type of concurrent circuit behaviour is demonstrated by Ebergen's RCEL component. The "RCEL with two replicated inputs" was first defined by Ebergen in section 2.2.3 on page 29 of his Ph.D. thesis by the trace command given below [36].

$$\text{pref}\,[\,(a?;p!)^2 \mid (b?;q!)^2 \mid (a?;p!\|r!)\|(b?;q!\|r!)\,]$$

The particular interest in this circuit results from the fact that its behaviour cannot be expressed in the most expressive delay insensitive grammar, $\mathcal{L}(G4)$, developed by Ebergen as mentioned on section 4.7 on page 72 of his thesis. This means that it can not automatically be decomposed by his synthesis methodology, and therefore no automatic delay insensitive decomposition, syntax-directed or otherwise, has yet been presented in the literature.

The stable state graph specification for this component is given in figure 5–25 below. The principal stable state of interest is v_0. At this state, a synchronization is required by both signals $a?$ and $b?$ to generate the set of outputs $p!$, $q!$ and $r!$. However the circuit may also generate the outputs $p!$ and $q!$ in response to

receiving the inputs $a?$ and $b?$ respectively. This stable state is identical to state v_0 of the simple concurrency example given in figure 5–23 above. The difference is that once $a?$ or $b?$ is acknowledged, the circuit enters a stable state where the remaining half of the synchronization does not have to occur. In the case of the RCEL sending a second $a?$ without sending a $b?$ (or sending a second $b?$ without sending an $a?$) results in a stable state transition where the synchronization of the c-element is never completed.

$$\begin{aligned} E(v_0) &= \{(\{a\},\{p\},v_1),(\{b\},\{q\},v_2), \\ &\quad (\{a,b\},\{p,q,r\},v_0)\} \\ E(v_1) &= \{(\{a\},\{p\},v_0),(\{b\},\{q,r\},v_0)\} \\ E(v_2) &= \{(\{a\},\{p,r\},v_0),(\{b\},\{q\},v_0)\} \end{aligned}$$

Figure 5–25: SSG specification of Ebergen's RCEL

As described in section 5.2.3, an implementation strategy involving rolling back (un-committing) synchronization inputs has already been developed. This strategy makes use of Keller select elements at the output of the completion tree and the use of merge gates at appropriate c-element inputs to generate a stimulus to fire the c-element. Complete details of such implementations can be found in section 5.2.3. It is sufficient to state that the stable state transition signals at the point when the synchronization is known to be abandoned is used to rollback the c-element circuitry. Although not discussed here, all of the circuit optimizations for rolling back only parts of a completion tree, as described at the end of section 5.2.3, may also be applied. The resulting circuit implementation is given in figure 5–26 below. It is interesting to compare this complex implementation with the relatively trivial implementation of the almost identical stable state graph given in figure 5–23.

5.3.6 General Arbitration

In this final section on the proposed delay insensitive circuit synthesis methodology, a generic decomposition strategy is suggested for any stable state graph specification. This decomposition strategy often results in circuits that are far more efficient than those produced by the generic and the improved generic implementation strategies described in sections 4.4 and 4.5. However, there remains a small number of pathological circuit specifications for which the improved generic

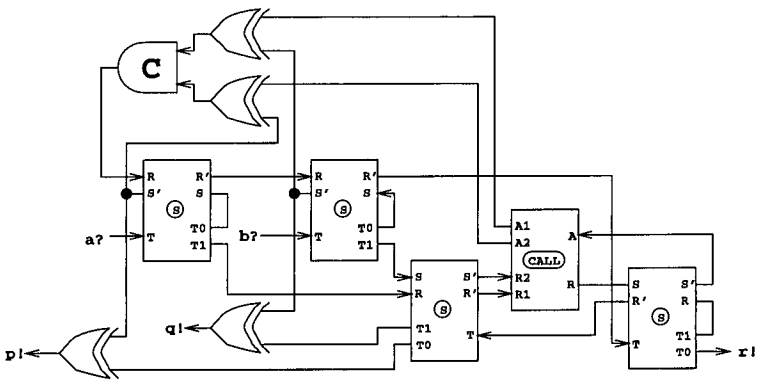


Figure 5–26: Implementation of Ebergen's RCEL

method developed in section 4.5 results in better circuits than those given here. One possible solution to an asynchronous circuit compiler is to compare the results of both strategies and output the best, thereby taking advantage of this method for the majority of cases where it produces better results. However, whenever possible, a circuit compiler should attempt to use the appropriate decomposition strategy for that class of circuit described previously in this chapter.

The central circuit model remains the same as originally described in chapter 4. The decomposition strategy decomposes a circuit behaviour into a number of subcircuits implementing distinct stable states in the circuit's behavioural specification. Each stable state, for which a special case decomposition has not been described in this chapter, is implemented using a generic 'committee problem' circuit. As mentioned in section 4.6.2, potential delay insensitive implementations of such circuits include Benko's token-ring based solution or the improved generic implementation described in section 4.5. These subcircuits consider each input signal transition sequentially and generate a single unique state transition signal as soon a stable state transition can fire.

The principal enhancement made for general arbitration circuits is the correct handling of concurrent circuits. In non-concurrent circuits, once a stable state transition fires the transition signal simply updates any required routing layer circuitry and generates the prescribed effect set external outputs. The non-concurrent nature of such circuits guarantees that another input signal will not be received until the outputs reach the external environment. This introduces two great simplifications; first that testing and updating of routing of circuits never occurs simultaneously and secondly that no further input signals arrive at the stable state

circuit once it has fired until the circuit behaviour reaches that stable state once again.

Routing Layer Race Conditions

The general arbitration circuits under consideration in this section can not depend upon such simplifications. In the first case, a transition signal may need to update the routing circuit of an input that still may be sent to the current stable state. In which case, there is a race condition between updating the routing layer circuitry and the arrival of the test input. The solution to this problem is to use a SEQ element to keep these signals mutually exclusive. Such a 2-SEQ element effectively implements a hardware semaphore enforcing a critical region around a routing layer circuit. One input to the 2-SEQ element is formed by the external input, the other input is generated from a CALL element of all the transition signals that must update the state of the routing circuitry that 'race' the test input. State change requests that are guaranteed never to occur simultaneously with the state change circuitry need not respect the critical region, and can update the routing circuitry directly. If the test input wins the arbitration and enters the critical section, it tests the state of the routing circuitry and generates the appropriate instance signal. This instance signal is then forked, one branch is fed to the appropriate input of the stable state subcircuit requiring that instance. The remaining branches from all input signal instances are merged together and used to re-prime the 2-SEQ element. If an update request wins the arbitration, the 2-SEQ output is fed back to the CALL element such that the required transition signal is demultiplexed, this signal then performs the updating of the circuitry and once acknowledged is forked. Once again one fork is used to generate the effect set outputs, the other branches are merged together and again re-prime the 2-SEQ element. This construction guarantees that the routing layer circuitry has no interface violations.

Input Signal Instance Forwarding

The remaining simplification that needs to be handled for general delay insensitive circuits is caused by the arrival of signals at a stable state once that subcircuit has already fired. Consider, for example, the effect of the input signal winning the race condition arbitration described in the paragraph above. In this case, the routing

circuitry is tested and the signal instance for the stable state that has just fired is generated.

The solution to this problem is *signal instance forwarding.* In order to maintain the correct behaviour of the global circuit, any input signals arriving at a stable state after it has fired need to be directed to the appropriate destination stable state subcircuit. Of course in the case where the stable state transition that fired is reflexive, no forwarding need be performed as the new signal instance will be processed at this stable state (synchronization layer circuit). The mechanics of forwarding an input signal instance is relatively straightforward. A transition can simply be merged with the signal instance (generated by the routing layer) that is fed into the synchronization circuit of the destination stable state. The delay insensitivity of the circuit's behaviour guarantees that another external instance cannot possibly arrive until the previous one has been processed and acknowledged by the circuit's behaviour.

Previously the functional interface of a synchronization layer circuit has been an input for each of the input signals that can occur at that state and an output for each of the stable state transitions from that state. This simple behaviour now has to be augmented to handle the potential race conditions introduced by arbitrary arbitration protocols. A synchronization layer circuit now needs an output for each input that needs to be forwarded, and an additional input to indicate that this is now the currently active stable state.

Conceptually, the circuit model now exists as a token-passing implementation. Initially, the circuit is initialized such that the synchronization layer circuit implementing the initial stable state has the 'token'. As stable state transitions fire, the token is based to the destination stable state. Any inputs that arrive at a stable state that currently holds the token are processed and the appropriate stable state transition signals generated. Input transitions that arrive at a stable state without the token are forwarded (redirected) to the destination stable state of the last stable state transition to fire at that stable state. In this way, forwarded input signals follow the 'token' possibly over several stable states if the token has since moved on.

At the circuit level, token passing is easily implemented using the transition signals that are already generated by the synchronization layer. In these circuits, a transition signal needs to update the appropriate routing layer circuitry, signal the token to the next stable state and finally output the appropriate effect output

signals. Typically, many stable states of a circuit specification may be implemented using the less general decomposition strategies described in this chapter. In such cases, these stable state implementations need not receive a token signal. The concept of a token is only used at general arbitration states. Conceptually, a circuit may lose a token on a stable state transition to a special case stable state (such as a sequential or non-concurrent state), and needs to generate one on return to general arbitration.

One implementation issue to be aware of is the possibility of 'live-lock'. There is an instant in a delay insensitive token passing structure as described above when the token is effectively in flight between synchronization layer circuits. As detailed above, a signal instance arriving at a stable state without the conceptual token forwards it to the next state. If the next state does not have the token, this state forwards it further. Due to arbitrary delays in the wires connecting subcircuits, it is possible for a forwarded instance to be passed in a continuous cycle until the token eventually arrives at the destination stable state. Provided that any arbitration performed between the incoming token signal and a forwarded input is *fair*, i.e. the token is eventually processed then the circuit should never 'live-lock'. In practice this should be quite rare as the time taken for a token to cycle between states is fairly large, allowing a synchronization layer circuit plenty of time to process a token. An alternative approach is to provide an acknowledge signal in the token passing circuit such that inputs are not forwarded until an acknowledgement has been received that the destination stable state has accepted the token.

Unfortunately, the schematic diagrams of circuits resulting from this decomposition are complex even for the smallest circuit specifications requiring general arbitration. Hence, no examples are presented to illustrate this generic decomposition strategy. However, the above textual description provides enough information to reproduce results presented here and generate a delay insensitive circuit implementation of an arbitrary delay insensitive behavioural specification.

Chapter 6

Advanced Synthesis

The following sections detail a number of improvements that can be made to the proposed synthesis methodology described in the previous chapter. The improvements described here are classified into two categories, circuit level optimization and behavioural optimization. Circuit level optimization improves the quality of the resulting circuit once a stable state graph has been translated into a network of components. Behavioural optimizations are transformations of the delay insensitive behaviour that preserve the implementation relation described previously in this thesis. These transformations preprocess the circuit's SSG into a form more suitable for efficient implementation, for example reducing the number of stable states.

6.1 Circuit Level Optimization

Peep-hole optimization is a technique commonly used in conventional software compiler design [1]. This optimization improves the quality of the generated machine code by recognizing commonly occurring inefficient sequences of instructions and replacing them by improved, functionally equivalent sequences.

The circuit and behaviour transformations and optimizations described in this section are presented as examples of many potential optimizations that are available in delay insensitive circuit synthesis. Rather than be an encyclopedic catalogue, this section describes many of the transformations used in this thesis and implemented in the prototype circuit compiler.

The correctness of these transformations can be verified using trace theoretic model checking, as implemented by several existing automated systems [30,34, 39]. Unfortunately, there does not currently exist a formalism for mathematically proving these rules in their generality. However, the sections below outline the reasoning behind each transformation and many individual instances have been verified using the Concurrency Workbench (CWB) and the other tools described above.

6.1.1 Component Generalization

Although technically not an optimization (unless used with technology mapping as explained below), *component generalization* is a semantics preserving circuit transformation that can enable the application of many of the optimizations described in this chapter. Component generalization takes advantage of the associativity and commutativity of merge gates and c-elements, to allow networks of these components to be abstracted as a single generalized component (as described in section 3.4). Component generalization can be applied whenever the output of a merge element (or c-element) is fed directly into another merge element (or c-element respectively) without being forked/branched to other components or external outputs. When such a topology is identified it may be considered a single generalized $n-1$-input component, where n is the sum of the components' inputs before the transformation. An example of this transformation is shown in figure 6–1 below.

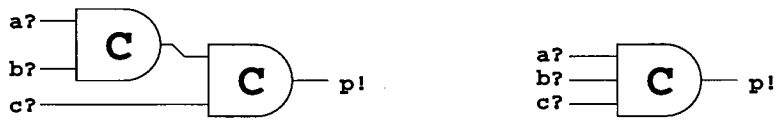


Figure 6–1: Component generalization transformation

With merges and c-elements, the generalized components can take advantage of commutativity and associativity to identity further optimizations, such as circuit identities and subexpression elimination described below. This process may be considered the inverse of decomposing a generalized component into a circuit basis. The original circuit topology may be recovered using a generalized circuit decomposition.

Similar generalization transformations may also be applied to other components to identify generalized call elements, generalized select elements, (sparse) decision waits, CSGs and k-SEQ components. However, these require more complex tests to correctly identify the appropriate functionality.

6.1.2 Circuit Identities

Perhaps the simplest type of circuit optimization is the detection of circuit identities. There are a number of times when the local composition of primitive components results in circuits with no function. These subcircuits may be replaced with the appropriate wiring connecting the inputs directly to the output.

The simplest occurrence of a circuit identity is when a signal is forked and fed into a Muller c-element, as shown in figure 6–2 below. The forked signals are synchronized and produce an output signal functionally equivalent to the input signal. Using the associativity and commutativity of c-elements and the component generalization transformation described above, it is possible to detect this identity in large component networks.

Figure 6–2: C-element circuit identities

Further examples of circuit identities involving toggles and merge gates are shown in figure 6–3. These local circuit topologies are often generated during the synthesis of cyclic sequence generators (CSGs). By merging together the two outputs of a toggle element, the actual state retained by this component is never used. Hence both the toggle and the merge gate may be replaced by a single wire. The remaining example in the figure below, shows one of the outputs of a toggle being merged with the toggle's input. In this case, the state is again lost as the toggle is forced to alternate, always reproducing the behaviour of a single wire.

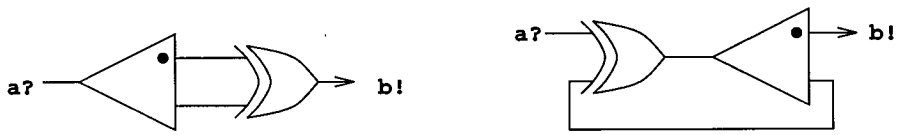


Figure 6–3: Toggle circuit identities

The first toggle example above is actually a representative of a class of similar circuit identities. If both test outputs of a Keller select element are merged without branching, the select element's state is similarly lost. In this case, the select element and merge gate may be replaced by three wires. One connects the test input to the output generated by the merge gate, the other two connect the set and reset inputs to the set and reset acknowledge outputs respectively.

Another (very rare) circuit identity is discovered by considering the select element implementation of a toggle element. When expanded this consists purely of a select element and a merge gate. However, because the test outputs are used to set and reset the select element's state before the acknowledge outputs are merged, this circuit is not recognized as a circuit identity using the previous rule.

The combination of the standard basis decompositions and the select element circuit identity leads to a large number of circuit optimizations. For example, row/column elimination (described below) can be considered an application of the select element circuit identity, when the select element is used to implement a decision wait element. Similarly, call elements (and generalized call elements) may be simplified when two (or more) acknowledge outputs are merged together, as there is no need to maintain a 'state bit' identifying which of the two corresponding requests originated the call. Similarly, the select element optimization may be used to reduce the states of generalized select elements.

One final circuit identity occurs from merging together both outputs of a choice element. Here, once again, because the actual choice that has been made is ignored, the choice element and merge gate may be replaced with a single wire.

6.1.3 Common Subexpression Elimination

For delay insensitive circuits (sub-circuits) that contain only merge gates and Muller c-elements, the outputs of a circuit may be considered a 'function' of its inputs. Common subexpression elimination detects circuitry that generates 'equivalent' signals (i.e. computes the same function) and removes this duplication of hardware by sharing the common outputs.

For example, consider the two circuits given in the figure below. The outputs `p!` and `q!` are generated by both inputs `a?` and `b?`. The standard construction for generating output circuitry is to use a pair of merge gates combining `a?` and `b?`, one for each output. Both input signals are forked and fed into each merge gate

which generates a single output. By removing the common circuitry, the inputs are fed directly into a single merge gate whose output is forked to produce `p!` and `q!` outputs.

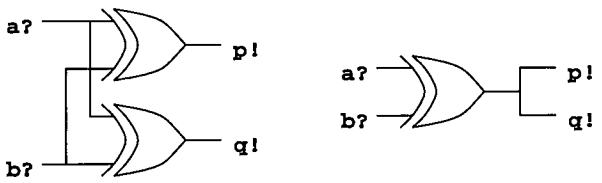


Figure 6–4: Subexpression factorization example

We use the symbol $\oplus$ as binary infix operator denoting the merge of two inputs. Similarly, the symbol $\copyright$ is used to denote the output of the c-element combining its two operands. From the properties of these primitive components, both these operators are commutative and associative. Similarly $\oplus$ distributes over $\copyright$, but $\copyright$ does not distribute over $\oplus$. These properties allows us to use an algebraic notion of subexpression equivalence.

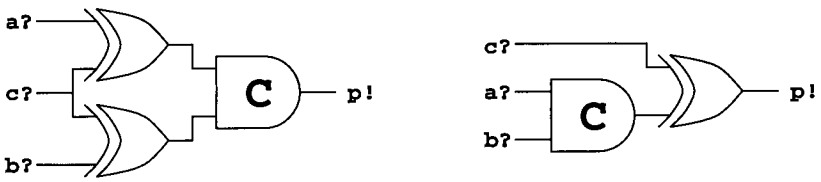


Figure 6–5: Subexpression factorization example

6.1.4 Technology Mapping

Hardware synthesis systems typically contain a *technology mapping* stage during circuit synthesis. This technology mapping process is analogous to the generation of machine-dependent instruction sequences in a conventional software compiler. Technology is typically applied as there may exist libraries of efficiently implemented components that may be used in place of subcircuits of primitive basis components. For example, if a circuit synthesis system had an efficient standard cell corresponding to a toggle element, a better circuit may be constructed by using these toggles rather than implementing them using select elements of 2×1

decision wait elements. However, in a different implementation technology or with an alternative parts library, a select element may be a better decision than a toggle.

The decision of which components (and subcircuits) to replace with functionally equivalent library parts (and subcircuits) depends both upon the parts available and the relative merits of each implementation. The word 'merit' in the previous sentence may be measured in terms of (or á combination of) performance, circuit area, cost, power consumption, testability or any other metric desired by the designer. Hence, a delay insensitive circuit synthesizer should be able to perform general peephole optimization using 'subgraph matching' to identify all potential instantiation candidates.

Many of the basis level implementations described in chapter 3 may be considered suitable circuit transformations. Each circuit may be implemented by its decomposition and (often) its decomposition may be implemented by the original circuit. This class of circuit optimization is similar to the circuit identities described above. However, it is assumed that circuit identity optimizations should always be applied as a functionally equivalent implementation requiring physically less hardware is always considered to be an improvement over its original unoptimized circuit.

6.1.5 Row/Column Elimination

An optimization that can be applied to both one and two-dimensional decision-wait elements is *row/column elimination.* This optimization is used to remove redundant rows or columns from the decision wait element. Rows (columns) in decision wait elements are used to distinguish transitions between different input signals. However, if all the outputs from a row (or column) are merged with the corresponding outputs from a second row (or column), before being used as inputs to other components, then no distinction is made between the two row inputs. In such a case, the inputs of these two rows (or columns) may be merged and fed into a single row (or column) in the decision-wait element. The outputs of this single row then replace the outputs of the original merge elements. Hence this transformation effectively removes a whole row (or column) from the decision wait.

This optimization can easily be checked/tested by algorithmic methods on circuits as a 'peephole' optimization. Although this optimization is unlikely to occur

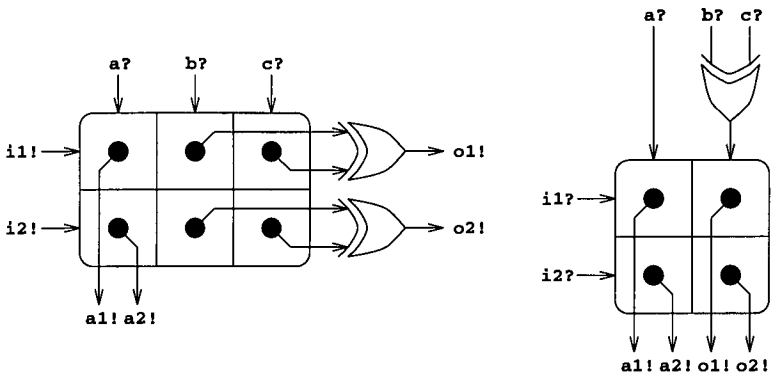


Figure 6–6: Row/Column elimination

in hand-designed circuits, it often arises through the use of automated synthesis methods, such as the generic implementation strategy given in section 4.4.

Note that multiple row/column eliminations may be applied in any order, since the application of the optimization does not prohibit its application on adjacent rows or columns. The use of merge associativity and symmetry is useful to identify potential optimization targets.

6.1.6 Row/Column Compression

Unlike one-dimensional decision-wait elements, it is often the case that not all of the outputs of an $M \times N$ decision wait element are used. In a one-dimensional decision wait element, an output is unused only if there can never be a transition on its associated input. In such a case, the $N \times 1$ decision wait element may be replaced by an $(N-1) \times 1$ decision wait.

An output of a two-dimensional decision wait element may never transition if that particular combination of row and column inputs can never occur together. This often occurs when rows and columns encode data and/or state information. For example, in the generic implementation strategy of section 4.4 some inputs are known not to occur in a given state.

Row/column compression makes use of the unused circuitry that implements non-live outputs to implement some of the live outputs. This is done by *compressing* (or folding) two rows (or columns). Two rows (or columns) may be 'compressed' if they do not both contain a live output in the same column (row) position. This means that the combination of either of those rows and a given

column position is either not live or is formed by the live row output. Hence the inputs to the two rows may be merged together as the input to a single 'compressed' row and the outputs of this row correctly assigned at each column position.

To clarify this transformation, consider the example sparse 2×3 decision-wait element given in figure 6–7 below. [This example is taken from the generic implementation of the Muller c-element in figure 4–4]. Of the six possible outputs, only four are used/live. This is because the input combinations `b?` and `i2?`, and `i1?` and `c?` can never occur in the circuit's behaviour.

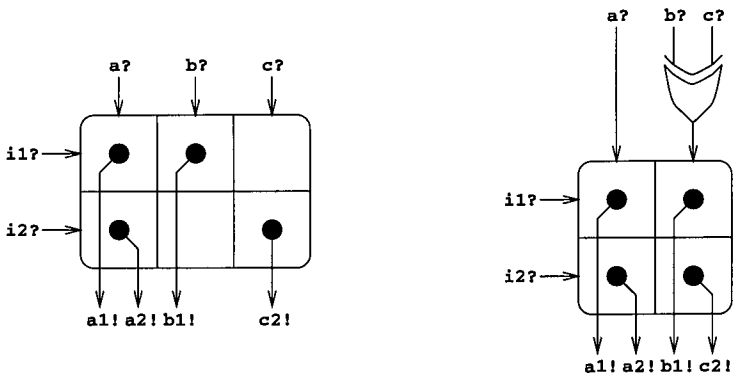


Figure 6–7: Row/Column compression

In this example, columns `b?` and `c?` may be compressed since they share no common live row positions. Notice that the two rows cannot be compressed because both `a1!` and `a2!` are live, and that the `a?` column (being non-sparse) may not be compressed with columns `b?` or `c?`. In the above example, a 3×2 decision wait element is replaced by a 2×2 decision wait element and a single merge gate. However, much larger savings are possible with larger sparse decision wait elements.

The order in which row/column compression optimizations are applied can affects the quality of the resulting circuit. Applying this optimization may prohibit its use on other pairs of rows or columns. This can often lead to less than 'optimal' circuits. One solution to this problem is to use a depth first search to determine the optimal sequence of row/column compressions. The search recursively considers each pair of rows/columns that can be compressed, and records the smallest circuit when no further compressions can be applied.

One further refinement of this transformation is to combine row/column compression with row/column elimination. This allows two rows to be compressed

when they both have live outputs at a given column position, provided that the two outputs are immediately merged together. For example, in figure 6–7 above, the two rows could be combined if the outputs `a1!` and `a2!` are merged together and not used as inputs to any other components or outputs. This can be used to replace the 3×2 decision wait with a 3×1 decision wait.

6.1.7 Decision Wait Splitting

Decision wait splitting is an optimization that is applicable to small and very sparsely populated 2-dimensional decision wait elements. The aim of decision wait splitting is to divide a large sparse decision wait element into two smaller independent decision waits. The set of row inputs to the large decision wait forms the set R and the set of column inputs the set C. Such a decision wait may be decomposed into two smaller decision waits, with row inputs R_1 and R_2 respectively and column inputs C_1 and C_2 respectively if R_1 and R_2 partition R, C_1 and C_2 partition C and all of the outputs formed by $R_1 \times C_2$ and $R_2 \times C_1$ are not live. Two non-empty sets x and y partition a set z iff $x \cap y = \emptyset$ and $x \cup y = z$.

Consider as an example the sparse 3×2 decision wait given in figure 6–8. This circuit has three column inputs, $a?$, $b?$ and $c?$, and two row inputs, $i1?$ and $i2?$. These inputs generate the three outputs $a1!$, $b1!$ and $c2!$. Analysis of the distribution of live outputs reveals that the column inputs $C = \{a, b, c\}$ may be partitioned into $C_1 = \{a, b\}$ and $C_2 = \{c\}$ and that the row inputs $R = \{i1, i2\}$ may be partitioned into $R_1 = \{i1\}$ and $R_2 = \{i2\}$ with the prescribed properties. This means that the 3×2 decision wait may be split into a 2×1 decision wait and a 1×1 decision wait (2-input Muller c-element).

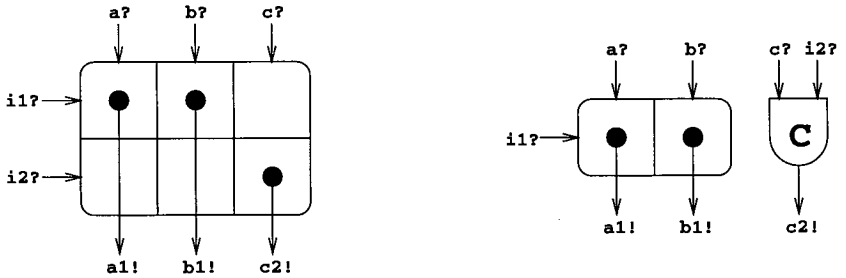


Figure 6–8: Decision wait splitting

6.1.8 Serial-Parallel Tradeoffs

A large number of the circuit constructions that have been described previously use a single signal, such as a stable state transition signal, to set or reset several Keller select elements. In the examples given, these select elements have their states modified sequentially. The set or reset acknowledge of each select element is used to set or reset the next. An alternative approach is to perform all N state changes concurrently by forking the signal to each of the set/reset inputs and combining the corresponding acknowledges using a c-element completion tree. This improves the performance of this (state transition) operation at the expense of additional hardware in the form of an N-input c-element. The time taken for a state transition operation is reduced from $N\tilde{m}$ to $\tilde{m} + \log_2 N\tilde{c}$, where $\tilde{m}$ and $\tilde{c}$ are the time taken to set/reset a select element and the transition time for a c-element respectively. In fact, for small values of N and slow c-elements it may be faster to perform the state modifications sequentially.

Typically, design constraints force an intermediate trade-off between speed and area/power consumption. This may be achieved by choosing an integer value $M < N$ and dividing the task of updating the N selects into M parallel tasks, updating approximately M/N selects each. This results in a state transition time of $\lceil N/M \rceil \tilde{m} + \lceil \log_2 M \rceil \tilde{c}$ at the hardware expense of an M-input c-element. For example, by choosing $M = 2$, the time to update a large number of select elements is approximately halved at the modest cost a c-element.

The example of updating select elements above is most common in updating routing layer circuitry with a transition signal before any outputs can be generated. However, there are a number of cases where independent subprocesses are known by the synthesis methodology to be initiated sequentially. These two can be performed in parallel, if appropriate. Typically, a combination of serial and parallel operations is made, with several approximately equal length sequences of operations being performed concurrently. The number of concurrent sequences depending upon the urgency of the operations and the transmission delay of the c-elements used.

In the proposed synthesis method, such decisions are postponed until technology mapping where the appropriate trade-off is finally made. For intermediate representation purposes, the synthesis methodology inserts a *sequencer* pseudo-component. This component effectively maintains a list of request and acknowledge pairs that are to be performed by a given signal. When the circuit is finally

implemented this 'sequencer' may be implemented either by wires, simply connecting the acknowledge of one pair to the request of the next, or appropriately using forks and c-elements.

6.1.9 Standard Logic Gates

One of the principal arguments against the use of delay insensitive circuits is that each of the 'primitive' components is much larger than those used in conventional synchronous design. One possible opportunity for improving the quality of synthesized is detecting subcircuits where standard logic gate implementations may be used. Conventional AND and OR gates may be used in delay insensitive circuits as a specialized form of merge element. This specialization constrains the environment of the merge to very restrictive behaviour.

The general rule is that a conventional logic gate can be used in those modes where an output is generated to acknowledge the arrival of each input or set of inputs. For example, consider an OR gate that initially has all of its inputs and output low. On the arrivial of the first input, the output transitions to acknowledge the arrival. Now its behaviour is restricted; sending the other input high has no effect on the output, hence there is no indication as to whether the input has yet reached the gate or has been delayed. If it hasn't yet arrived, retracting the first input may result in a spike or glitch in the output (communication interference). Hence the valid delay insensitive modes of operation of a conventional OR gate is to raise one of the inputs, wait for the output change, lower that input and again wait for the output change. Repeating this behaviour with either of the inputs.

Consider the CCS process algebra specifications given below.

```
bi XOR   a?.c!.XOR + b?.c!.XOR
bi OR    a?.c!.a?.c!.OR + b?.c!.b?.c!.OR
bi ORNOT a?.c!.OR
bi AND   a?.b?.c!.OR
```

Figure 6–9: Logic gate specifications

As has been described previously, the conventional logic XOR gate is synonymous with the delay insensitive merge gate. An OR gate implements a merge gate that guarantees that each input will transition an even number of times, before the

other input transitions (an even number of times). The AND gate initially performs a synchronization before continuing to behave as an OR gate. Conventional boolean algebra may be used to manipulate the invertors required to establish the initial conditions in the circuit. If these invertors are placed around standard logic gates, the invertor(s) and logic gate may be replaced with the appropriate complex gate, using fewer transistors in the final circuit. The use of OR gates is also particularly useful in reducing the multiplexor costs of buses that use the four phase handshaking convention.

Unfortunately, checking each merge gate to determine whether it can be implemented by a conventional logic gate is a computationally difficult problem. The task is to consider all possible execution traces of the input behaviour to guarantee, for example, the double aternating behaviour of the OR gate. The two signals that have to be proven to alternate may be complex functions of the circuit's state, and all possible execution paths must account for all possible valid environment and circuit data communication and arbitration events.

However, there are a number of merge gates that may be readily tested. These are predominantly the output layer merges used to generate output signals, whose behaviours may be analyzed by examining local stable states.

As an example, consider the implementation of the stable state graph given in figure 6–21 on page 174. The given implementation uses requires a single c-element and a two input merge gate. From the specification, the $a?$ input signal only occurs once, after this, only the other input to the merge gate ever transitions. In this case, the merge gate may be implemented using the standard logic gate AND(a,NOT(p)). Alternatively, a much better implementation would be to implement the whole circuit using the single standard logic gate AND(a,b).

6.1.10 N-Toggle Optimization

There are a number of circuit optimization strategies that are only applicable to a restricted class of circuits. A example of this form of optimization is the synthesis of N-toggle circuits. Consider the implementation of a 5-toggle given in figure 6–10 below. This is the circuit generated by the decomposition method described in section 3.4.7, that merges together the outputs of a $2^{\lceil \log_2 N \rceil}$-toggle.

The first improvement to notice is that it contains a toggle that has both of its outputs merged. As described above this forms a circuit identity and the

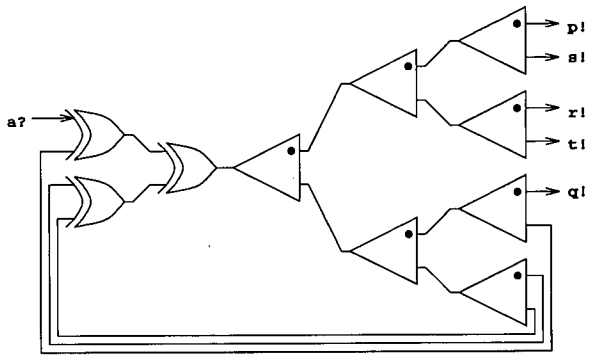


Figure 6–10: Original 5-Toggle implementation

appropriate toggle and merge gates may be replaced by a single wire. By choosing the appropriate $M - N$ outputs of the M-toggle to be merged together, where $M = 2^{\lceil \log_2 N \rceil}$, the above optimization may be used to significantly improve the size and speed of N-toggle circuits. The number of toggles and merges saved is given by the recurrence function **optsize**(n) given below.

$$\begin{aligned}\textbf{optsize}(n) = \text{if } n < 2 \text{ then } 0 \\ \text{else let } t = 2^{\lfloor \log_2 n \rfloor} \\ \text{in } \textbf{optsize}(n-t) + t - 1 \\ \text{end;}\end{aligned}$$

Note that this recurrence equation has the following properties, $\textbf{optsize}(2^k) = 2^k - 1$ and $\forall n \geq 0 \,.\, \textbf{optsize}(n) \leq n$. Using this recursive function, the number of components required to implement an N-toggle is $(M-1) - \textbf{optsize}(M-N)$ toggles and $(M-N) - \textbf{optsize}(M-N)$ merge gates.

Another point to note is that the inputs to the merge gates now occur with uneven frequencies, hence the tree may be rebalanced using Huffman encoding. The external input always occurs most frequently, and the remaining inputs should be ordered by their depth in the toggle tree. We can also determine the performance improvement of these optimizations over the equations given in section 3.4.7. The reduction in the period of the circuit is given by the expression $\textbf{optspeed}(n)(\tilde{x}+\tilde{t})$, where $\tilde{x}$ and $\tilde{t}$ are the average transition times for a merge gate and a toggle respectively, and **optspeed**(n) is given by the recurrence function below.

$$\begin{aligned}\textbf{optspeed}(\text{n}) = \text{if } n < 2 \text{ then } 0 \\ \text{else let } k = \lfloor \log_2 n \rfloor\end{aligned}$$

in **optspeed**$(n - 2^k) + 2^k k$
end;

The result of these optimizations produces the 5-toggle implementation shown in figure 6–11 below, which is an improvement over the original implementation shown in figure 6–10 above. The period of the original 5-toggle was $24\tilde{t} + 16\tilde{x}$ (or $24\tilde{t} + 13\tilde{x}$ with a Huffman encode merge tree), whereas the optimized circuit now has a period of $22\tilde{t} + 11\tilde{x}$.

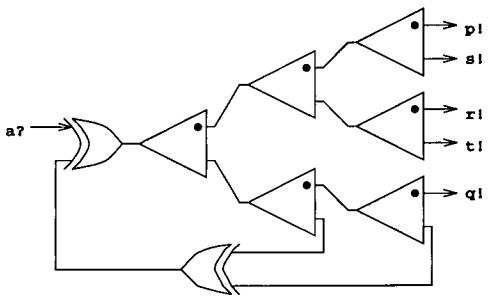


Figure 6–11: Optimized 5-Toggle implementation

6.1.11 CSG Optimization

Another example of a set of optimizations that are applicable to a small class of circuit is synthesis of cyclic sequence generators (CSGs). In the general case, CSGs are implemented using the standard strategy described in section 5.1.2, based upon the use N-toggles. As has been shown in the previous sections, these N-toggles themselves may be improved.

The simplest form of improvement for CSGs is simply to generate an optimal N-toggle as above and then merge the appropriate outputs, where N is the length of the cycle of output signals to be repeated. If there exist any toggles where both output signals merged these can be removed using the circuit identity optimization described in section 6.1.2. An example, of this synthesis applied to the cyclic sequence 'PRQR' is shown in figure 6–12 below. In this section, we shall refer to CSGs by a string of capital letters, where each letter represents a different output signal. The first letter is the first output to be generated, and so on. The stable state graph corresponding to 'PRQR' is also shown below.

$$
\begin{aligned}
E(v_0) &= \{(\{a\},\{p\},v_1)\} \\
E(v_1) &= \{(\{a\},\{r\},v_2)\} \\
E(v_2) &= \{(\{a\},\{q\},v_3)\} \\
E(v_3) &= \{(\{a\},\{r\},v_0)\}
\end{aligned}
$$

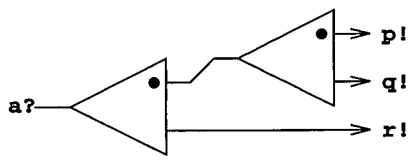


Figure 6–12: Optimized 'PRQR' cyclic sequence generator

For the case, where all of the inputs in a sequence are the same, the circuit identity option will reduce the implementation to a single wire. In the event where all the outputs in a sequence are unique, the N-toggle implementation is optimal. The circuit given above for the cyclic sequence 'PRQR' happens to be an optimal implementation for that sequence. However, there are two cases where circuit implementations can be found, that are better than those using an independently synthesized N-toggle.

The first case occurs for sequences that are not powers of two, and therefore the N-toggle is implemented starting from an M-toggle, where $M = 2^{\lceil \log_2 N \rceil}$. In this case, the choice of the $M-N$ feedback signals in the N-toggle may affect the quality of the resulting circuit. Consider for example, the two possible implementations of the CSG 'PQRPQRS' shown in figures 6–13 and 6–14 below.

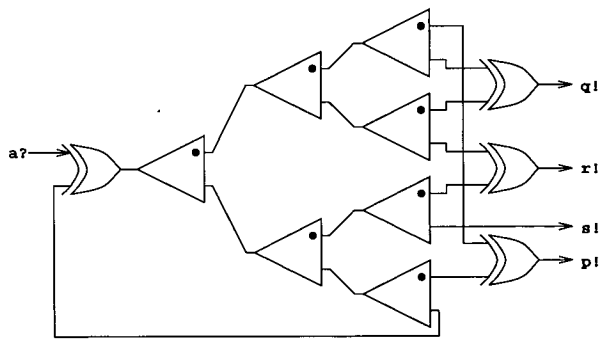


Figure 6–13: Original 'PQRPQRS' CSG implementation

The difference between the two implementations is the choice of M-toggle outputs to be used as feedback to create the N-toggle. In the example, choosing the last output of the 8-toggle results in a circuit requiring 7 toggles and 4 merges, while choosing the penultimate output requires only four toggles and a single merge. The approach used by the current circuit compiler is to consider all possible com-

binations of outputs to determine the best resulting circuit. The current choice (permutation) of M-toggle outputs is represented by a single of length M, that contains the CSG specification string, with the feedback outputs represented by the character "-". The M character positions in the string represent which output or feedback signal is associated with each of the M-toggle's outputs. The single restriction is that the string must contain the characters of the CSG specification in order, but the "-" characters can occur at any position. For example, the implementation in figure 6–13 is denoted by the sequence 'PQRPQRS-' and figure 6–14 by the sequence 'PQR-PQRS'. For a given sequence, it is a straightforward task to determine how many circuit identities are available in the resulting circuit. A simple 'butterfly' permutation can be used to order the signals as they would occur in the schematics shown. Pairs (quadruples, octuples...) at the appropriate positions in the sequence indicate that a toggle/merge pair may be eliminated from the resulting circuit.

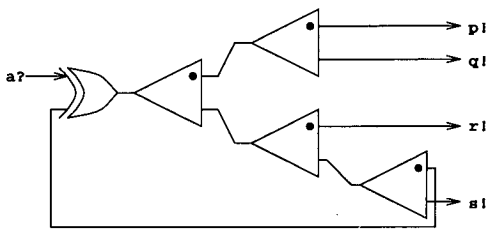


Figure 6–14: Optimized 'PQRPQRS' CSG implementation

The other case where improved CSG implementations can be found occurs when the sequence length is not a prime number. As an example, consider the implementation shown below for the CSG 'PQPRPS'. In the CSG implementation methodologies given above this circuit would be based upon an 8-toggle with the appropriate number of feedback signals. The best possible implementation using this strategy would result in an implementation requiring 6 toggles and one merge gate.

The given implementation requires only four toggles and a single merge gate. Close inspection of the circuit reveals its organization. It is composed of a single toggle, with one of the outputs fed into a 3-toggle. This decomposition can be applied to any non-prime cyclic sequence generator. A CSG of length mn may be decomposed into a single m-toggle and m CSGs of length n. These smaller CSGs consist of every m^{th} output, each CSG starting at the next position in the

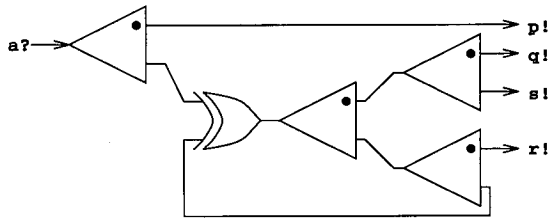


Figure 6–15: Optimized 'PQPRPS' cyclic sequence generator

sequence. These CSGs may then be recursively decomposed using this method or implemented using any of the synthesis methods above. For the 'PQPRPS' CSG, the above implementation made use of a toggle with the first output being fed into a 'PPP' CSG, and the second output being fed into a 'QRS' CSG. As mentioned above, a 'PPP' CSG is trivially implemented by a wire.

At the risk of increasing the search space to find CSG implementations, a further optimization can be applied to this CSG factoring optimization. By merging together some of the outputs from the initial n-toggle, it is possible to produce a variety of smaller CSGs with lengths multiples of n. For example, the CSG 'PQRQPRQR' could be implemented using the methods above using toggles and a single merge gate. However, it may be decomposed into a 4-toggle and four CSGs of length two. These CSGs generate the sequences 'PP', 'QR', 'RQ' and 'QR'. As can be seen the first CSG may be implemented by a wire. If the remaining three outputs of the toggle are merged together (eliminating a toggle from the 4-toggle), they may be fed into a single CSG that implements 'QRQRQR'. This CSG can be seen to be implemented by a single toggle. The resulting implementation, shown in figure 6–16 below, requires only three toggles and one merge rather than the six toggles and one merge required by the original.

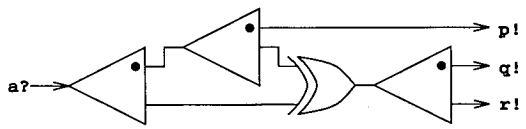


Figure 6–16: Optimized 'PQRQPRQR' cyclic sequence generator

One nice feature of implementing these CSGs is that their specifications may be represented by finite length strings. These strings can then be easily manipulated

to specify the intermediate CSGs required by the recursive decomposition strategy above.

6.1.12 Constant Time Counters & CSGs

One of the results of the research into delay insensitive modulo-N counters has been the design of constant response time counters by several groups [104,41,40, 114,118]. These methods are based upon speculative evaluation. Because modulo-N counters (or CSGs) have only a single input, the next result of the counter may be determined in advance and stored in a 2×1 decision wait element (or $N\times1$ for a CSG). The arrival of the next external input signal fetches the result from the decision wait and triggers the next speculative evaluation. If such a 'lookahead' scheme is implemented at regular intervals in the decomposition, the resulting circuit can be shown to have a constant response time to an external request. In typical operation of such a circuit, many of the subunits of a module-n counter (or CSG) subunits are precomputing the results of the next input transition in parallel.

This optimization has been included in the current circuit synthesis system. The compiler can be told to use constant response time implementations where appropriate. The compiler uses a threshold on the maximum number of outputs a 'cached' CSG can have. For modulo-N counters, there are always two potential outputs $p!$ and $q!$ at any stage of the decomposition, which are easily held in a 2×1 decision wait element. However with CSGs, there may be a large number of potential outputs, requiring a large one-dimensional decision wait. Obviously there is a threshold above which the time saved by pre-evaluating a query is lost by the decision wait overhead in area and time. Finally, it should be noted that not all CSGs, even with small numbers of outputs, have constant response time implementations. Unlike modulo-N counters, the recursive CSG decomposition strategies described above are only applicable when the length of the sequence was not prime. This leads to the conclusion that the performance of a CSG is proportional to the largest prime factor of its cycle length.

6.2 Behavioural Transformation

The following sections describe optimizations that are made before the synthesis of a circuit from a behavioural specification. The transformations are designed to produce specifications that 'implement' the original specification, but are (by some measure) easier to synthesize into efficient circuits. For example, many of these optimizations attempt to reduce the number of stable states in a circuit specification. As described in the previous chapters, the proposed design methodology is dependent upon the number of stable states in the circuit specification. However, it is possible for a circuit with a number of stable states, to implement a circuit containing more.

6.2.1 Input Clustering

The first and conceptually most simple transformation of a stable state graph is clustering of its input signals. This transformation attempts to identify sets of input signals that only ever occur concurrently in the cause sets of stable state transitions. At every stable state where such input sets occur, the circuit implementation attempts to synchronize the signals as part of determining which stable state transition to fire. When this is the case, the clustered set of input signals may be abstracted to a single 'external' input signal. This signal may be generated by an external c-element that synchronizes the true external input signals.

A pair of input signals a and b, of a stable state graph (V, T) may be clustered together, if they only ever occur together in all stable state transitions of T. i.e. $\forall t \in T \,.\, a \in \mathbf{cause}(t) \leftrightarrow b \in \mathbf{cause}(t)$. Clustering of all input signals can be performed in time proportional to the number of stable state transitions by maintaining a compatibility matrix of input signals. Such a matrix may even be constructed as the stable state graph is generated. As an example application of input clustering consider the stable state graph and its implementation given in figure 6–17 below.

This is the implementation that would be generated for the input stable state graph using the proposed design methodology. The SSG specification is recognized as state-holding, sequential circuit and is generated using the method described in section 5.1.2 for general sequential circuits. The routing circuits for both inputs $a?$ and $b?$ are implemented using a cyclic sequence generator on their signal instance

$$E(v_0) = \{(\{a,b\},\{p\},v_1)\}$$
$$E(v_1) = \{(\{a,b\},\{q\},v_0)\}$$

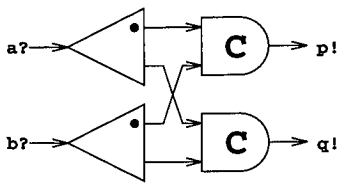


Figure 6–17: Input clustering example

graphs, both being implemented by toggles. The two sequential stable states are then implemented with two input Muller c-elements to generate the appropriate outputs.

Analysis of this stable state graph reveals that inputs $a?$ and $b?$ always occur together in transition cause sets. These two external input signals may therefore be combined using a Muller c-element. Its output, denoted here as $a©b$, is then replaces both inputs $a?$ and $b?$ in all the cause sets of the original SSG in which they appear. The resulting circuit specification is shown in figure 6–18 below. This stable state graph can be seen to be the specification of a toggle element. The resulting, functionally equivalent implementation, is also shown in the same figure.

$$E(v_0) = \{(\{a©b\},\{p\},v_1)\}$$
$$E(v_1) = \{(\{a©b\},\{q\},v_0)\}$$

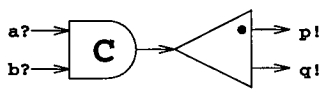


Figure 6–18: Input clustered implementation

6.2.2 State Collapsing

Potentially one of the most powerful stable state graph transformations is *stable state collapsing.* Conceptually, this transformation is quite simple: two stable states may be implemented by a single synchronization layer circuit if that single subcircuit implements all of the stable state transitions of both original states. This optimization makes use of the implementation operator defined in section 2.4 where a valid implementation may implement additional stable state transitions that are not prescribed by the circuit specification. More formally, two stable states v_1 and v_2 may be collapsed to produce a single state v_3 where $E(v_3) = E(v_1) \cup E(v_2)$, if no cause set of a transition from v_1 is a subset of a cause set of a transition from v_2, and vice versa, any transitions from v_1 that have identical cause sets to transitions of t_2 have identical effect sets and next states. This is expressed by the following

conditions $\forall t_1 \in E(v_1).\forall t_2 \in E(v_2).\mathbf{cause}(t_1) \subseteq \mathbf{cause}(t_2) \rightarrow t_1 = t_2$, where $t_1 = t_2$ if $\mathbf{cause}(t_1) = \mathbf{cause}(t_2) \wedge \mathbf{effect}(t_1) = \mathbf{effect}(t_2) \wedge \mathbf{next}(t_1) = \mathbf{next}(t_2)$.

Technically, the above condition on the next set of the two states is overly restrictive. For example, consider the SSG specification given in figure 6–19 below. The two states satisfy all the criteria for state collapsing except that the next state of $\{a, b\}$ is different between them. However, these two next states are those states being collapsed. This then forms a circular argument, the two destination states are the same if the two states are collapsed. Hence the above condition may be relaxed to include identical next states or states that will be collapsed by this transformation. In the current example, the two states may be collapsed to produce a stable state graph with a single state having three transitions with cause sets $\{a, b\}$, $\{c\}$ and $\{d\}$ respectively.

$$
\begin{aligned}
E(v_0) &= \{(\{a,b\},\{p\},v_1),\\
&\quad\ (\{c\},\{q\},v_1)\}\\
E(v_1) &= \{(\{a,b\},\{p\},v_0),\\
&\quad\ (\{d\},\{r\},v_0)\}
\end{aligned}
$$

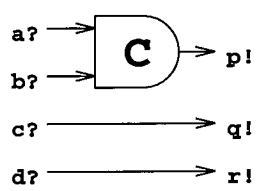


Figure 6–19: State collapsing example

During circuit synthesis further states may be collapsed once the state transition signals have been selected to update routing layer circuitry (as described in section 5.2.1). At this later stage, two stable states may be combined with the above conditions except that two transitions with identical cause sets need only have identical effect sets and update the same routing layer circuits to identical states. This allows the above optimization to be applied to an even larger class of circuits specifications. It also creates the need for powerful 'state transition selection' algorithms as the set of signals selected to update routing circuit state directly affects the number of routing layer states.

Finally, above definition prohibits the state collapsing of stable states where one or both identical transitions from each stable state exhibit static non-determinism. It is possible to extend the above conditions to correctly handle such cases, where $\mathbf{cause}(t_1) = \mathbf{cause}(t_2)$, depending upon the implementation operator being used. If using Ebergen's satisfaction relation, $\sqsupseteq$, the two statically nondeterministic stable state transitions may only be merged if they have identical non-deterministic

choices, i.e. $\forall t \in E(v_1).\mathbf{cause}(t) = \mathbf{cause}(t_1) \rightarrow \exists t' \in E(v_2).\mathbf{cause}(t') = \mathbf{cause}(t_2) \wedge \mathbf{effect}(t) = \mathbf{effect}(t') \wedge \mathbf{next}(t) = \mathbf{next}(t')$. Using this implementation operator, the resulting collapsed state contains a single non-deterministic transition with the same choices. However, using the improved implementation relation proposed in this thesis, $\sqsupseteq_1$, it is possible to implement a non-deterministic choice with a more deterministic implementation. Using this operator, two statically non-deterministic transitions with identical cause sets may be collapsed if they have at least one output choice in common, i.e. there is a valid 'firing' that implements both choices. This may be expressed by a condition $\exists t \in E(v_1).\mathbf{cause}(t) = \mathbf{cause}(t_1)$ and $\exists t' \in E(v_2).\mathbf{cause}(t') = \mathbf{cause}(t_2)$ such that $\mathbf{effect}(t) = \mathbf{effect}(t') \wedge \mathbf{next}(t) = \mathbf{next}(t')$. The resulting collapsed state contains transitions for each choice outcome both stable states have in common. $\forall t_3 \in E(v_3).\mathbf{cause}(t_3) = \mathbf{cause}(t_1)$ then $t_3 \in E(v_1)$ and $t_3 \in E(v_2)$ [with the less strict constraints on $\mathbf{next}(t_3)$].

As an example of the use of the introduced implementation operator to simplify the resulting circuit by specifying potential design alternatives, consider the SSG specification given in figure 6–20 below. This specification contains two statically non-deterministic stable states. At state v_0 the input $a?$ creates an arbitrary choice between outputs $p!$ and $q!$, and at state v_1 the input $a?$ makes a choice between outputs $p!$ and $r!$. Using Ebergen's implementation relation this circuit would be implemented by a toggle element feeding two choice elements that produce the appropriate outputs. However, using the new improved implementation operator introduced in section 2.4, the circuit specification below may be state collapsed and implemented by a single wire connecting the input $a?$ to the output $p!$.

$$\begin{aligned} E(v_0) &= \{(\{a\},\{p\},v_1),(\{a\},\{q\},v_1)\} \\ E(v_1) &= \{(\{a\},\{p\},v_0),(\{a\},\{r\},v_0)\} \end{aligned}$$

Figure 6–20: Static non-determinism state collapsing

6.2.3 State Combining

Another powerful SSG optimization technique is *state combining*. A stable state may be combined with another if their behaviours are equivalent after a number of input signals have been received. This is a more general form of the state collapsing

optimization described in the previous section. This is a commonly occurring case in real circuit specifications, such as the stack element example given in section 7.1.

This thesis makes use of the notation $v\backslash i$ to denote the behaviour of the stable state v after receiving the set of input signals $i \subset I$. This expression is well formed if no state transition from v may be fired by the input signal set i, $\forall t \in E(v)$ $\mathbf{cause}(t) \not\subseteq i$, and that there remain some stable state transitions that can fire on upon receipt of further signals, $\exists t \in E(v) \,.\, i \subset \mathbf{cause}(t)$. The $\backslash$ operator is similar to that used in Josephs and Udding's DI algebra [59].

The simplest form of state combining when a single state v_1 already implements a second state v_2 after a set of inputs i has been received, i.e. $v_2 \sqsupseteq_1 v_1\backslash i$. This may be tested with the conditions that for every transition t in $E(v_2)$, $\mathbf{cause}(t) \cap i = \emptyset$ and $\exists t' \in E(v_1) \,.\, \mathbf{cause}(t') = \mathbf{cause}(t) \cup i \wedge \mathbf{effect}(t') = \mathbf{effect}(t) \wedge \mathbf{next}(t') = \mathbf{effect}(t)$ and there are no spurious output signals generated. The test for spurious signals is identical to that described for such signals in the previous section, section 6.2.2, on state collapsing. This even applies to statically non-deterministic signals as discussed in that section.

An example stable state graph that is amenable to this transformation is given in figure 6–21 below. This SSG contains two stable states, where the behaviour of v_1 is identical to the behaviour of v_0 upon receipt of the single input $a?$, i.e. $v_1 = v_0\backslash\{a\}$. In such cases, state v_1 may be implemented using the synchronization layer circuitry for state v_0. The only change at the circuit implementation level is that state transition signals in addition to updating any necessary routing layer circuitry must also generate the required pseudo-inputs to the destination synchronization layer circuit. These pseudo-inputs are generated by simply merging a transition with the appropriate internal input signal instance before it is fed into the synchronization circuit. In the example below, every time the Muller c-element fires, the pseudo-input $a?$ needs to be generated and this is achieved using a merge gate on the trivially stateless $a?$ input.

$$
\begin{aligned}
E(v_0) &= \{(\{a,b\},\{p\},v_1)\} \\
E(v_1) &= \{(\{b\},\{p\},v_1)\}
\end{aligned}
$$

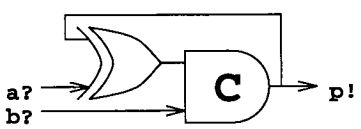


Figure 6–21: State combining example

6.2.4 Initial State Combining

If the optimization described above is used to combine the initial stable state v_0 with another vertex, additional hardware must be used to establish the circuit's initial conditions. This is done by placing iwire components at the initial instance of all the input signals occurring in the 'initialization set'.

Several optimizations may be performed to move the iwire components around in a circuit. For example, if iwire components are placed at all the inputs of a Muller c-element, they may removed and replaced by a single iwire component at its output. Similarly, if only a subset of c-element inputs have iwire components on them, the c-element and the iwire(s) may be replaced by an asymmetric c-element. An example of such a transformation is given in the initial state combining example shown below in figure 6–22.

$$E(v_0) = \{(\{a\}, \{p\}, v_1)\}$$
$$E(v_1) = \{(\{a, b\}, \{p\}, v_1)\}$$

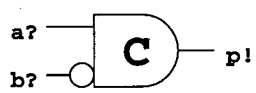


Figure 6–22: Initial state combining example

An iwire component placed at a set or reset input of a Keller select element may be implemented by using a select element initialized to the appropriate value with an iwire component on the appropriate acknowledge output. Similarly, an iwire component placed at the test input of a select element may be moved to the appropriate output depending upon its current (initial) value. Care must be taken with busy waiting circuits whose iwires are inserted in the 'waiting' loop. In this case, a compiler may attempt to move an iwire component an unbounded number of times 'abstractly interpreting' a non-terminating computation. This may be avoided by placing an arbitrary upper limit on the number of such iterations.

Chapter 7

Case Studies

This chapter demonstrates the application of the circuit synthesis strategy described in the previous chapters on some example circuit specifications. Many of the examples used in the previous section were chosen to demonstrate a particular stable state configuration or semantics preserving transformation. In the following sections, we describe the synthesis of several larger circuits whose behavioural specifications have been taken from delay insensitive circuit literature.

7.1 Stack Element

A good example of the application of the proposed design method to a complex specification is the stack element. The delay insensitive stack element was first used as an example by Josephs and Udding [60]. In their paper, they describe the derivation of an implementation using their CSP-based DI algebra. Their formal derivation (and the full verification [61]) of the hand designed circuit are quite involved covering several pages. This design was considered a good test study to compare against the presented fully automated procedure. Similar delay insensitive stack element designs have also discussed by other researchers [62,65].

The formal specification of the behaviour of this stack element has already been presented in section 2.5 on page 40. In this section, a shorthand description of a stack element's functional behaviour (given in CCS in figure 2–6) is translated into a complete delay insensitive description of its behaviour with all signal leavings explicitly represented (given in CCS in figure 2–9). This specification may then be converted directly into a stable state graph. The SSG corresponding to the full

behaviour is given in figure 4–2 on page 92. This SSG, with 5 stable states and 14 stable state transitions, acts as the input to the circuit synthesis part of the circuit methodology.

Analysis of each the stable state graph reveals that this class of circuit is a non-concurrent state-holding circuit. The four input signals `dempty?`, `dapop?`, `dapush?` and `dfull?` all only occur at a single state, and hence their routing circuitry will be trivially stateless. However, the two remaining inputs `push?` and `pop?` occur at all five stable states. The next step in the synthesis process is the behavioural transformations described in section 6.2. Input cluster analysis reveals that none of the inputs can be clustered, but state reduction is able to greatly reduce the number of required stable states. The initial state v_0 may be state combined with v_2 as $v_2 \backslash \{dempty\}$, state v_1 may be combined with either state v_2 or v_3 as $v_2 \backslash \{dapop\}$ or $v_2 \backslash \{dapush\}$ respectively, states v_2 and v_3 may be state collapsed (both before and after state combining), and state v_4 may be combined with v_3 as $v_3 \backslash \{dfull\}$. The result of applying all of these behavioural transformations is the modified stable state graph shown in figure 7–1 below.

$$
\begin{aligned}
E(v_1) \;=\; & \{(\{push, dapop\}, \{apush, dpush\}, v_1), \\
& (\{push, dapush\}, \{apush, dpush\}, v_1), \\
& (\{push, dfull\}, \{full\}, v_1 \backslash \{dfull\}), \\
& (\{push, dempty\}, \{apush\}, v_1 \backslash \{dapush\}), \\
& (\{pop, dapop\}, \{apop, dpop\}, v_1), \\
& (\{pop, dapush\}, \{apop, dpop\}, v_1), \\
& (\{pop, dfull\}, \{apop\}, v_1 \backslash \{dapop\}), \\
& (\{pop, dempty\}, \{empty\}, v_1 \backslash \{dempty\})\} \\
E(v_0) \;=\; & E(v_1) \backslash \{dempty\}
\end{aligned}
$$

Figure 7–1: Optimized stack element SSG

This modified stable state graph can be shown to implement the original stack element SSG given in figure 4–2 using the implementation operator defined in section 2.4. The principal advantage of these transformations is that the resulting stable state graph effectively only contains a single stable state. This means that the resulting circuit will require no routing level circuitry, as all 6 inputs are now trivially stateless. The next step in the synthesis is to implement this single stable

state. The first point to notice is that the inputs `dapop` and `dapush` are 'input mergeable' as described in section 5.2.5. The resulting circuit can then be implemented directly using the partitioned transition method described in section 5.2.4. The initialization conditions are then handled by abstract interpretation, creating an asymmetric c-element and several initialized Keller select elements. The resulting circuit is shown in the figure below. In this design, all select elements are assumed to be initialized to zero (reset) and all call elements initialized to `R1`. [The call schematic symbol is used to represent a Keller select element where both set and reset acknowledge are fed into a merge gate].

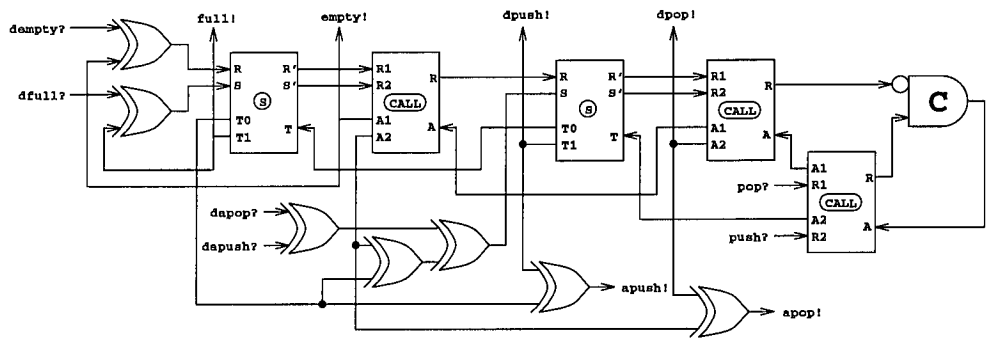


Figure 7–2: Stack element implementation

The partition transition method applied to this circuit generates the above circuit, based upon a 3×2 decision wait element. The above schematic diagram is shown using the select element basis, where 5 Keller select elements and a Muller c-element are configured to implement the decision wait. However, the partition method can also instantiate a 3×2 decision wait directly.

Interestingly, this circuit is identical to the hand designed implementation given by Josephs and Udding in their original paper [60].

7.2 Modulo-N Counters

An interesting common test case in the field of delay insensitive design are the class of modulo-N counters. This class of circuit has been investigated by a number of researchers using different design styles [41,104,114,118,120]. A modulo-N counter has a single input, $a?$ and two outputs $p!$ and $q!$. In response to the first $N-1$ transitions on the input $a?$, the circuit generates a single output transition on

$p!$. On the N^{th} transition of the input, the circuit generates a single output on $q!$. Hence the function of the circuit is to count the number of occurrences of an input signal modulo the constant value N. The behaviour of a modulo-N counter, $N > 1$, is specified by the trace command below.

$$\mathbf{pref}\,[(a?; p!)^{N-1}; a?; q!\,]$$

Translation of this circuit specification into a stable state graph results in a strictly sequential cyclic SSG with N vertices. Using the proposed design methodology, this class of circuit is identified as a non-concurrent, state-holding sequential circuit and implemented using the techniques described for cyclic sequence generators given in sections 5.1.2 and 6.1.10. Using these techniques the optimized implementation of a modulo-17 counter, specified by the trace command $\mathbf{pref}*\,[(a?; q!)^{16}; a?; p!\,]$, is shown in figure 7–3 below.

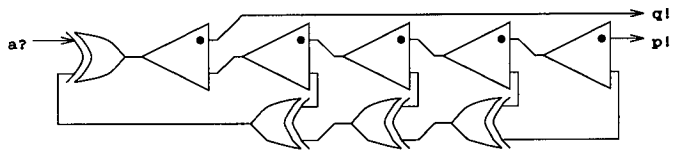


Figure 7–3: Modulo-17 counter

As has been described in section 6.1.10, this implementation is based upon a 32-toggle, where 16 outputs have been merged to form the $p!$ output, 15 outputs have been merged to form a 15-toggle and the remaining output used to generate $q!$. Circuit identities (both outputs of a toggle fed into a merge gate) are then identified and replaced with wires, and the remaining merge gates Huffman encoded. This automatic decomposition results in a circuit that is composed of 5 toggles and 4 merges.

It is interesting to compare this automatically generated circuit with the hand-designed one presented by Ebergen (on page 139) in his Ph.D. thesis [36]. In his implementation, he constructs a modulo-17 counter from two modulo-3 counters, a toggle and two merge gates. The standard construction for a modulo-3 counter is shown in figure 7–4 below.

Using this modulo-3 counter as a building block, Ebergen decomposes the modulo-17 counter circuit into the implementation given in figure 7–5. This

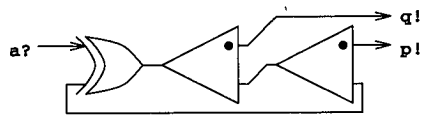


Figure 7–4: Modulo-3 counter

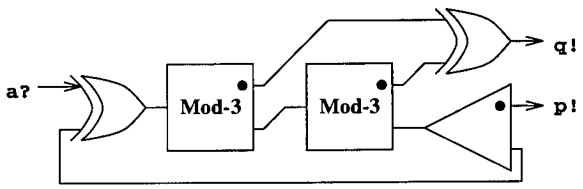


Figure 7–5: Ebergen's modulo-17 counter

modulo-3 counter is exactly the same implementation as is generated by the proposed synthesis methodology.

Interestingly, although the two compositions look quite different, they both use exactly the same hardware, 5 toggles and 4 merge gates. With a little thought, it can be seen that this is the minimum possible hardware that could implement the required specification. A modulo-N counter must uniquely represent N possible states, this requires a minimum of $\lceil \log_2 N \rceil$ bits of a binary encoding. This value is both necessary and sufficient. Each of these bits may be maintained by a primitive two state component such as a toggle or a Keller select element. Similarly, because a modulo-N counter has one input and two outputs, an implementation with this many toggles requires $\lceil \log_2 N \rceil - 1$ two input merge gates to maintain the correct number of inputs and outputs. Each toggle has one input and two outputs and each merge has two inputs and one outputs, assuming that not input or output is left unconnected, there must always be one less merge gate than toggle element to construct a circuit with a single input and two outputs (i.e. a modulo-n counter).

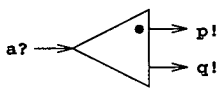
Because both modulo-17 counter implementations use exactly the same hardware, the remaining metric to compare the two circuits is performance. Using the notation that $\tilde{t}$ represents the average transition delay for a toggle, and $\tilde{x}$ represents the average transition delay for a merge gate, the period (the time to cycle the counter back to its initial state) was determined for each design. The generated circuit has a period of $62\tilde{t} + 57\tilde{x}$ and Ebergen's circuit has a period of $50\tilde{t} + 66\tilde{x}$. If we assume that $\tilde{t} = \tilde{x}$, Ebergen's circuit is faster than the compiler generated

example with 116 rather than 119 transitions (in reality $\tilde{t} > \tilde{x}$, so Ebergen wins by a bigger margin).

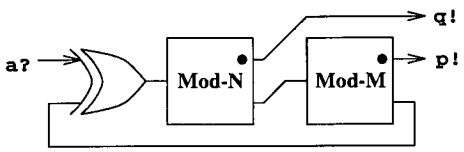
Given the performance of Ebergen's modulo-17 counter, it was decided to determine a systematic method for generating more efficient modulo-N counters and adding this procedure to the synthesis methodology. According to Ebergen in his thesis, the decomposition of his modulo-17 counter is based on the calculations $17 = 18 - 1$ and $18 = 3 \times 3 \times 2$. It turns out that his hand design was effectively solving a set of recurrence equations to generate a modulo-17 counter.

The efficient design of modulo-N counters may be solved by the solution of a recursive recurrence equations. There are three possible cases that may be applied. These cases and their hardware implementations are shown below. These equations are more general forms of those presented by Ebergen *et al.* [40,41] who only consider the cases where $m = 2$.

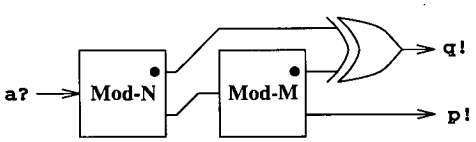
[1] Case 2



[2] Case $mn - (m - 1)$



[3] Case mn



The first case is the base case of the recursion; a modulo-2 counter is implemented as a single toggle. The next recursive case constructs a modulo-$(m(n-1)+1)$

counter from a modulo-m counter, a modulo-n counter and a single merge component. The final recursive case constructs a modulo-mn counter from a modulo-m counter, a modulo-n counter and a single merge component. It is easy to show that all possible values of $N \geq 2$ are covered by these equations by assuming that in the recursive cases the value m is always 2 (using case 1). Then the two recursive cases have the form $2n - 1$ and $2n$. These forms can then be used to decompose modulo-N counters for odd and even values of N respectively.

The performance of each of these cases can also be given in terms of a set of recursion equations. We denote the period of a modulo-n counter as $\mathbf{period}(n)$. Then each of the decomposition cases given above have periods given by the three equations below respectively.

$$\begin{aligned} \mathbf{period}(2) &= 2\tilde{t} \\ \mathbf{period}(m(n-1)+1) &= m\mathbf{period}(n) + \mathbf{period}(m) + mn\tilde{x} \\ \mathbf{period}(mn) &= m\mathbf{period}(n) + \mathbf{period}(m) + (mn-1)\tilde{x} \end{aligned}$$

Using the above recursion equations, a compiler can determine the fastest possible implementation of a modulo-N counter for a given value of N. In addition, the circuit compiler can make use of frequency information in the decomposition process to Huffman encode each of the merge trees in the circuit. Using this strategy, the circuit compiler currently generates a Huffman balanced implementation of Ebergen's circuit with a period of $50\tilde{t} + 55\tilde{x}$. This circuit is believed to currently be the fastest known delay insensitive modulo-17 counter implementation (of this type).

One benefit (side-effect) of this work on modulo-N counters was the inclusion in the proposed design methodology of the ability to implement constant response time CSGs (see section 6.1.12).

7.3 One Place Buffer

The next circuit synthesis example, the four-phase handshaking one-place buffer, is taken from Martin's paper "The Limitations to Delay-Insensitivity in Asynchronous Circuits" [79]. Martin describes this circuit as a basic building block of asynchronous circuit design, since it is used to implement the sequencing of two (four-phase communication) actions. The original specification for the circuit's behaviour, given in his CSP-like specification language, are given below.

$$*[\,[xi];\ xo\uparrow;\ [\neg xi];\ xo\downarrow;\ yo\uparrow;\ [yi];\ yo\downarrow;[\neg yi]\,]$$

Figure 7–6: One-place buffer handshaking expansion

The circuit has two inputs `xi?` and `yi?` and two outputs `xo!` and `yo!` respectively. The CCS version of this specification and a more detailed description of the circuit's intended behaviour are given in section 2.6. As described in that section, the above single handshaking expansion is insufficient to specify the circuit intended behaviour precisely, and interface partitioning must be applied. After this transformation has been applied the buffer's transition graph is as shown in figure 2–12 on page 47. This corresponds to the stable state graph shown in figure 7–7 below.

$$\begin{aligned}
E(v_4) &= \{(\{yi\},\{xo\},v_1)\} \\
E(v_3) &= \{(\{xi,yi\},\{xo\},v_1)\} \\
E(v_2) &= \{(\{yi\},\{yo\},v_3), \\
&\qquad (\{yi,xi\},\{yo\},v_4)\} \\
E(v_1) &= \{(\{xi\},\{xo,yo\},v_2)\} \\
E(v_0) &= \{(\{xi\},\{xo\},v_1)\}
\end{aligned}$$

Figure 7–7: One-place buffer SSG

Initial analysis of the 5 vertex stable state graph reveals that the circuit exhibits premature concurrency. Otherwise the specification is a cyclic, state holding sequential circuit. Behavioural transformation reveals that no inputs may be

combined, but that the initial state v_0 may be 'initial state combined' with v_3 as $v_3 \backslash \{yi\}$ (state v_4 may also be combined with v_3 but is eliminated due to the premature concurrency). The resulting effective stable state graph has three vertices (corresponding to v_1, v_2 and v_3 in the SSG above). The input signal `xi` occurs in states v_1 and v_3 and its signal instance graph (SIG) forms a sequential cycle of two states and hence may be implemented as a single toggle. The implementation as a toggle satisfies the transition signal selection criteria for premature concurrency. Similarly `yi` appears at two states, v_2 and v_3, forming a two state sequential cycle and may also be implemented as a toggle component. The synchronization layer circuitry for state v_2 consists of a single Muller c-element and the remaining two stable states require no additional hardware. Finally, the initial conditions are satisfied by abstract interpretation resulting in an asymmetric c-element for state v_3. The automatically synthesized circuit is shown in figure 7–8 below.

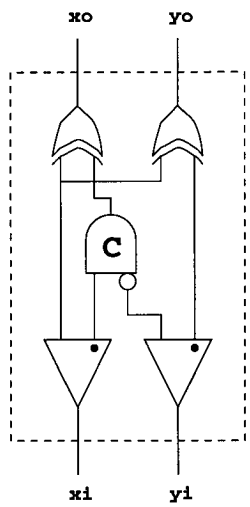


Figure 7–8: One place buffer implementation

The circuit correctly implements the one place buffer as specified by Martin in his paper. Interestingly, Martin argues in his paper that the above circuit cannot be implemented delay insensitively without reordering the communication actions at the circuit's interface. The above circuit implementation shows that a delay insensitive decomposition does exist for toggles and c-elements. Assuming Martin's low level implementation result is valid, this design forms a proof that the toggles themselves cannot be implemented in a truly delay insensitive way. This result, that there does not exist a delay insensitive gate circuit for a toggle, has independently been shown by Seger [105].

Chapter 8

Conclusions

8.1 Summary

The immediate goal of the research presented in this dissertation is to develop an automatic method to translate behavioural descriptions into asynchronous circuits. The potential size and complexity of modern digital systems continues to increase at a phenomenal rate due to advances in VLSI fabrication technology. At the same time, this continual improvement results in a rapid succession of technology generations, putting ever more pressure on circuit and product design times. These constraints are forcing systems engineers to consider novel design strategies and adopt new tools that increase design productivity and reduce design times. The approach to these problems advocated by this thesis is the use of automated compilation techniques to generate delay insensitive circuits. In addition to solving the complexity issues facing designers, asynchronous circuits, and particularly delay insensitive circuits, offer a large number of natural benefits to the technological problems facing by VLSI engineers.

Conventionally, asynchronous circuits have been notoriously difficult to design. The huge simplification achieved by discretizing time allows powerful mathematical formalisms, such as the theory of finite state automata, to be brought to bear on the analysis of system behaviour. This has allowed synchronous design techniques to build up, over the years, a huge number of automated tools and a large knowledge base to ease the design process. However, recent developments in the theory of concurrent systems coupled with the computational power of modern workstations promise to redress the balance. This thesis introduces a novel formal model of delay insensitive behaviour that drastically reduces the complexity of the design process by eliminating much of the interleaving caused by concurrent events.

Self-timed modules are easily composed to create larger structures and therefore form convenient building blocks for VLSI system design. Because a module's interface is defined purely by its function and not timing, any component (or network of components) may be replaced by a functionally equivalent one, without the need for global retiming analysis. Such a design style permits the reuse of well designed modules and allows incremental performance improvement, through either faster building blocks or improved fabrication technology. Considering that some current synchronous designs use as much as half of a chip's power budget and 40% of its area for clock distribution, the single fact that asynchronous circuits need no global clock makes such a design style attractive.

The advantage of an automated circuit compiler is that the complexity of asynchronous circuit decomposition is hidden from the circuit designer by providing the abstraction of a formal specification language. This abstraction allows circuit design to be viewed as a 'programming' problem, taking advantage of the achievements in computer science (and software engineering) to bridle complexity. This allows systems to be designed by system (application) specialists, away from the details of low-level circuits and to think of system behaviour in higher level (algorithmic) terms. Because the circuit specification is effectively a concurrent program, simulation and formal proof techniques may be used to verify that the program meets the system specification. Self-timed modules may be designed (and implemented) independently and expected to work together when integrated into a system. This permits a designer to rapidly consider different design alternatives.

Once a designer is satisfied with a circuit specification, it can be automatically translated into circuit-level implementation. This avoids the costly, time-consuming, tedious and often error-prone task of manually generating the circuit or converting the specification into a form suitable for existing tools. If the circuit compiler is implemented correctly the resulting circuit should be correct-by-construction and faithfully implement the behaviour of the original specification. Ideally, the correctness of the translation process could be proven. Unfortunately, no mathematical formalism currently exists in which to perform such a proof. Alternatively, because the behaviour of the resulting circuit may be expressed in terms of the composition of the primitive component behaviours it is possible to automatically verify each decomposition against the original specification. Indeed such a verification has been applied to many of the circuit implementations given in this thesis.

8.2 Evaluation

The principal results of this thesis are the development of an abstract model of delay insensitive behaviour, and an efficient automated synthesis method for translating behavioural specifications into delay insensitive circuits based upon the above model. The stable state graph representation makes use of the constraints imposed upon a system by Udding's rules of delay insensitivity to avoid explicitly representing the interleaving of concurrent signals. The circuit synthesis method presented automatically decomposes arbitrary behavioural specifications into delay insensitive networks of primitive components. The quality of circuits generated by this procedure is shown in chapter 7 to be as good as, and often better than, many hand designed circuits. Indeed this synthesis method can be shown to be optimal for large classes of circuits, as mentioned in section 4.6.1.

As reviewed in chapter 1, a number of other researchers have suggested automatic compilation methods for asynchronous circuits [10,14,21,36,117]. All of these existing methods make use of syntax-directed translation to reduce the complexity of the synthesis task. This technique uses the syntactic structure of the circuit's behavioural specification (when described in a suitable formalism) to determine the decomposition of components.

The principal problem with syntax-directed translation is that the quality of the resulting circuit is dependent upon the way its behaviour is specified. This places a burden on the circuit designer to understand how the circuit compilation method works, and to specify the problem in such a way that the most efficient circuit is generated. Ideally, two functionally equivalent circuit specifications should result in the generation of the same (best) circuit implementation. Although, the respective authors argue the advantages of directing a circuit compiler towards the desired implementation, the history of compilers for software programming languages reveals that this leads to specifications that are tied to given implementation technologies and do not allow the full power of current algorithmic methods to be brought to bear. For example, the computational advantage of global state minimization during circuit synthesis greatly reduces the size and improves the quality of circuits generated from most behavioural specifications.

Another claim made by syntax-directed translation methods, such as Ebergen's, is that the size of the resulting circuit is linear in the size of the specification. This

can be shown a tight bound since some classes of circuits, such as N-toggles, require $\mathcal{O}(n)$ components (from a finite circuit basis), where n is the specification size. However, it is often possible to do much better. Unfortunately for syntax-directed translation styles both the worst case and the *best* case are linear in specification size. For this reason, non-syntax-directed approaches are required to efficiently implement the extremely large class of circuits which require less than $\mathcal{O}(n)$ components. To overcome this limitation, some syntax-directed circuit compilers apply peep-hole optimizations to improve the quality of resulting circuits. However for typical large circuit specifications, local transformations cannot correct the global inefficiencies in decomposition strategy.

One limitation with previous delay insensitive circuit synthesis methodologies has been the use of restrictive circuit models and specification formalisms. Most previous circuit compilers impose a predefined synthesis model, such as four-phase handshaking bundles for each communication action. This means that it is impossible to precisely define an arbitrary interface behaviour. This constraint means that it is impossible to specify the primitive components used to implement the circuit in the specification language itself. This makes it much harder to verify that a circuit meets its prescribed behaviour, as the compiler inputs and outputs are described in different formalisms. The one exception to this statement, Ebergen's design methodology, allows the specification of arbitrary interface behaviours provided that they may be represented as DI commands. Unfortunately, there are number of perfectly valid delay insensitive behaviours, such as the RCEL implemented in section 5.3.5, that cannot be expressed as such commands.

Prior to the work described in this thesis, the attempt to generate delay insensitive implementations of arbitrary interface specifications was based on the inefficient generic implementation strategy described in section 4.4 [5,97]. This thesis also details several improvements to this strategy that produce more efficient implementations that require less hardware.

8.3 Future Work

The main portion of this work consists of a large number of theorems or tactics on how to decompose delay insensitive behaviours into networks of primitive components. It would be nice to be able to formally prove these theorems and thereby validate a circuit compiler based upon this method. Unfortunately, no suitable formalism exists that can describe manipulations of classes of circuit behaviours. If such a formalism could be developed, the circuits generated by this method could be shown 'correct-by-construction'. At present, confidence that a circuit meets its specification is provided by verification. An intermediate approach would be to integrate the proposed circuit decomposition method into a higher-order logic hardware theorem prover. Although it is not currently possible to prove (or disprove) the generic theorems described in this thesis, it is possible to make use of them to direct a proof that each specific application of them to a given circuit is valid (semantics preserving).

An alternative course of investigation, is to consider the use of the above circuit synthesis strategy as a back-end and/or module generator to a conventional high-level synthesis system. Such systems take a high-level algorithmic description and automatically generate a synchronous digital data-path circuit that meets that specification. The input high-level description is expressed either in a conventional programming language such as Pascal, or in a hardware description language such as VHDL. The use of high-level hardware description languages should abstract the designer from a particular implementation technology, and allow such a synthesis system to decide upon an appropriate, possibly asynchronous, design style. Indeed, the composition properties of self-timed modules makes them ideal for the register transfer level components, such as ALUs, registers and multiplexors, used by high-level synthesis systems. In such a 'silicon compiler', the presented circuit synthesis methodology provides the means to synthesize the complex asynchronous controller circuits required by such data-path designs.

Finally, application of delay insensitive decomposition methods may be applied to fields outside that of VLSI circuit design. One potential use of the methodology presented in this thesis is in the design of distributed systems and parallel programs. Each communication action (voltage transition in circuit terms) can be considered to be a packet (or message) transmitted over a computer network. The

utility of program design styles that are tolerant of network communications delays between communicating machines is obviously desirable. Rather than completely decompose the specification into trivial primitive operations the decomposition is halted at an intermediate level. Because all internal communication actions are delay insensitive, the assignment of different modules to different processors may be arbitrary. Indeed this provides a mechanism for dynamic load balancing.

Glossary

Asynchronous An asynchronous circuit is an 'unclocked' circuit, i.e. a circuit which does not rely on global synchronization by an external clock signal. Asynchrony implies the absence of any timing bounds on the operation of a circuit (whose duration may be subject to many uncontrolled factors).

Delay Insensitive A circuit is delay insensitive if its correct operation is independent of any assumptions about the delays of the individual components or wires in the circuit except that those delay be finite. c.f. *speed independent.*

Equipotential An equipotential region is a portion of a circuit within which propagation delays in wires are considered to be negligible. The smaller the area of the region, the more validity this assumption has in practice.

Isochronic A fork or branch of a wire in a circuit is considered to be isochronic if the difference in propagation delays between branches is negligible. This is obviously the case if all branches of the fork are contained in an *equipotential* region. A *delay insensitive* circuit in which all forks are isochronic is effectively only *speed independent.*

Self-Timed Self-timed circuits are those which are characterized by the use of asynchronous initiation and completion (or request/acknowledge) signals. This is sometimes referred to as "self-synchronization" by the Russians. The class of self-timed circuits includes all *delay insensitive*, *speed independent* and *self-clocked* circuits.

Self-Clocked Self-clocked circuits are *self-timed* designs that are implemented using a hidden internal clock within an *equipotential region.* Although internally they are composed of clocked synchronous elements, self-clocked circuits retain an external asynchronous interface.

Speed Independent A circuit is said to be speed independent if its correct operation is independent of the delays in the individual components of the circuit except that those delay be finite. It is assumed there is no propagation delay associated with the wires of the circuit. c.f. *delay insensitive.* A speed independent circuit is often called a "Muller circuit" or "aperiodic" in translation from Russian.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Douglas B. Armstrong, Arthur D. Friedman, and Premachandran R. Menon. Design of asynchronous circuits assuming unbounded gate delays. *IEEE Transactions on Computers*, 18(12):1110–1120, December 1969.

[3] J. C. M. Baeten. *Applications of Process Algebra.* Cambridge University Press, 1990.

[4] Clifford Barney. Logic designers toss out the clock. *Electronics*, pages 42–45, December 1985.

[5] Igor Benko and Jo C. Ebergen. Delay-insensitive solutions to the committee problem. In *Advanced Research in Asynchronous Circuits and Systems*, pages 228–237, Salt Lake City, Utah, November 1994.

[6] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. Technical Report 842, Institut National de Recherche en Informatique et en Automatique (INRIA), May 1988.

[7] Hans Bisseling, Hank Eemers, Michiel Kamps, and Ad Peeters. Designing delay-insensitive circuits. Technical Report VOC, Eindhoven University of Technology, September 1990. Final Report of the Software Technology Postgraduate Programme.

[8] David L. Black. On the existence of delay-insensitive fair arbiters: Trace theory and its limitations. *Distributed Computing*, 1:205–225, 1986.

[9] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th Design Automation Conference*, pages 40–45. ACM/IEEE, June 1990.

[10] Geoffrey M. Brown. Towards truly delay-insensitive circuit realizations of process algebras. In *Designing Correct Circuits*, September 1990.

[11] E. K. Brunvand and M. Starkey. An integrated environment for the design and simulation of self timed systems. In A. Halaas and P. B. Denyer, editors, *VLSI 91*, page 4a.2, August 1991.

[12] Erik Brunvand. Parts-R-Us a chip apart(s)... Technical Report CMU-CS-87-119, Carnegie Mellon University, May 1987.

[13] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, September 1991. CMU-CS-91-198.

[14] Erik Brunvand and Robert F. Sproull. Translating concurrent communicating programs into delay-insensitive circuits. Technical Report CMU-CS-89-126, Carnegie Mellon University, 1989.

[15] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical Report CMU-CS-92-160, Carnegie Mellon University, July 1992.

[16] J. A. Brzozowski and J. C. Ebergen. Recent developments in the design of asynchronous circuits. Technical Report CS-89-18, University of Waterloo, Computer Science Department, May 1989.

[17] J. A. Brzozowski and J. C. Ebergen. On the delay-sensitivity of gate networks. *IEEE Transactions on Computers*, 41(11):1318–1327, November 1992.

[18] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. Technical Report CMU-CS-91-195, Carnegie Mellon University, October 1991.

[19] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.

[20] Steven M. Burns. Automated compilation of concurrent programs into self-timed circuits. Master's thesis, California Institute of Technology, 1988. Caltech-CS-TR-88-2.

[21] Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In *Advanced Research in VLSI: Proceedings of the 5th MIT Conference*, pages 35–50, 1988.

[22] Steven M. Burns and Alain J. Martin. Performance analysis and optimization of asynchronous circuits. In *Advanced Research in VLSI 1991: Proceedings of the University of California, Santa Cruz, Conference*, pages 71–86, 1991.

[23] Luca Cardelli. *An Algebraic Approach to Hardware Description and Verification.* PhD thesis, Department of Computer Science, University of Edinburgh, 1982.

[24] M. K. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[25] Thomas J. Chaney and Charles E. Molnar. Anomalous behaviour of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, 22(4):421–422, April 1973.

[26] Tam-Anh Chu. On the models for designing VLSI asynchronous digital systems. *INTEGRATION, the VLSI journal*, 4(2):99–113, June 1986.

[27] W. A. Clark and C. E. Molnar. Macromodular computer systems. In R. Stacy and B. Waxman, editors, *Computers in Biomedical Research*, chapter 3, pages 45–85. Academic Press, 1974. Vol. IV.

[28] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs Workshop*, pages 52–71. Springer–Verlag, May 1981. LNCS 131.

[29] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 117–126, 1983.

[30] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. Technical Report LFCS-89-83, Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, August 1989.

[31] Al Davis, Bill Coates, and Ken Stevens. Automatic synthesis of fast compact asynchronous control circuits. In *Working Conference on Asynchronous Design Methodologies*, Manchester, England, March 1993.

[32] Jack B. Dennis. Modular asynchronous control structures for a high performance processor. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 55–80. ACM, New York, 1970.

[33] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1985.

[34] David L. Dill. *Trace Theory for Automatic Hierachical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. The MIT Press, 1989.

[35] David L. Dill and Edmund M. Clarke. Automatic verification of asynchronous circuits using temporal logic. In Henry Fuchs, editor, *1985 Chapel Hill Conference on VLSI*, pages 127–143. Computer Science Press, 1985.

[36] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits.* PhD thesis, Eindhoven University of Technology, 1987.

[37] Jo C. Ebergen. Arbiters: An exercise in specifying and decomposing asynchronously communicating components. Technical Report CS-90-29, University of Waterloo, Computer Science Department, 1990.

[38] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[39] Jo C. Ebergen and Sylvain Gingras. A verifier for network decompositions of command-based specifications. In T. Mudge, V. Milutinovic, and L. Hunter, editors, *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, volume 1, pages 310–318. IEEE Computer Society Press, January 1993.

[40] Jo C. Ebergen and Ad M. G. Peeters. The modulo-N counter: Design and analysis of delay-insensitive circuits. Technical Report CS-91-25, University of Waterloo, Faculty of Mathematics, June 1991.

[41] Jo C. Ebergen and Ad M. G. Peeters. Modulo-N counters: Design and analysis of delay-insensitive circuits. In *2nd Workshop on Designing Correct Circuits*, pages 27–46, Lyngby, January 1992.

[42] Reinhard Enders, Thomas Filkorn, and Dirk Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, 6:155–164, 1993.

[43] A. D. Friedman and P. R. Menon. Synthesis of asynchronous circuits with multiple-input changes. *IEEE Transactions on Computers*, 17(6):559–566, June 1968.

[44] Arthur D. Friedman and Premachandran R. Menon. *Theory & Design of Switching Circuits.* Digital Systems Design Series. Computer Science Press, 1975.

[45] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In *VLSI 93*, September 1993. (submitted).

[46] Dov Gabbay, Amir Pneuli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, 1980.

[47] J. D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In *Working Conference on Asynchronous Design Methodologies*, Manchester, England, March 1993.

[48] M. A. Gavrilov. *The Theory of Relay-Switching Circuits. Analysis and Synthesis of Relay-Switching Circuit Structure.* USSR Academy of Sciences Press, Moscow and Leningrad, 1950. (in Russian).

[49] Rix Groenboom, Mark B. Josephs, Paul G. Lucassen, and Jan Tijmen Udding. Normal form in a delay-insensitive algebra. In *Working Conference on Asynchronous Design Methodologies*, Manchester, England, March 1993.

[50] Scott Hauck. Asynchronous design methodologies: An overview. Technical Report TR93-05-07, Department of Computer Science and Engineering, Univeristy of Washington, 1993.

[51] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[52] Alan B. Hayes. Stored state asynchronous sequential circuits. *IEEE Transactions on Computers*, 30(8):596–600, August 1981.

[53] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[54] C. A. R. Hoare. A model for communicating sequential processes. Technical Report PRG-22, Programming Research Group, Oxford University Computing Laboratory, 1981.

[55] Lee A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, 31(12):1133–1141, December 1982.

[56] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257:161–190, 1954.

[57] David A. Huffman. The design and use of hazard-free switching networks. *Journal of the ACM*, 4(1):47–62, January 1957.

[58] Mark B. Josephs and Jan Tijmen Udding. An algebra for delay-insensitive circuits. Technical Report WUCS-89-54, Washington University, St. Louis, 1989.

[59] Mark B. Josephs and Jan Tijmen Udding. Delay-insensitive circuits: an algebraic approach to their design. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension*, pages 342–366. Springer–Verlag, August 1990. LNCS 458.

[60] Mark B. Josephs and Jan Tijmen Udding. The design of a delay-insensitive stack. In Geraint Jones and Mary Sheeran, editors, *Designing Correct Circuits*. Springer-Verlag, September 1990.

[61] Mark B. Josephs and Jan Tijmen Udding. Designing a delay-insensitive stack. Technical Report CS9004, Department of Computer Science, Groningen University, 1990.

[62] Mark B. Josephs and Jan Tijmen Udding. Implementing a stack as a delay-insensitive circuit. In *Working Conference on Asynchronous Design Methodologies*, Manchester, England, March 1993.

[63] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes and three problems of equivalence. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 228–240, August 1983.

[64] Robert M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, 23(1):21–33, January 1974.

[65] Joep Kessels and Martin Rem. Designing systolic, distributed buffers with bounded response time. *Distributed Computing*, 4:37–43, 1990.

[66] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[67] Kim G. Larsen and Bent Thomsen. A modal process logic. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, pages 203–210, Edinburgh, 1988.

[68] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *28th Design Automation Conference*, pages 302–308. ACM/IEEE, June 1991.

[69] Luciano Lavagno, Kurt Keutzer, and Alberto L. Sangiovanni-Vincentelli. Synthesis of verifiably hazard-free asynchronous control circuits. In *Advanced Research in VLSI 1991: Proceedings of the University of California, Santa Cruz, Conference*, pages 87–102, 1991.

[70] S. C. Leung and H. F. Li. On the realizability and synthesis of delay-insensitive behaviors. *IEEE Transactions on Computer-Aided Design*, 14(7):833–848, July 1995.

[71] Kuan-Jen Lin and Chen-Shang Lin. Automatic synthesis of asynchronous circuits. In *28th Design Automation Conference*, pages 296–301. ACM/IEEE, June 1991.

[72] P. F. Lister and A. M. Alhelwani. Design methodology for self-timed VLSI systems. *IEE Proceedings, Pt. E*, 132(2):25–32, January 1985.

[73] C. N. Liu. A state variable assignment method for asynchronous sequential switching circuits. *Journal of the ACM*, 10, 1963.

[74] Yonatan Malachi and Susan S. Owicki. Temporal specifications of self-timed systems. In H. T. Kung, Bob Sproull, and Guy Steele, editors, *VLSI Systems and Computations*, pages 203–212, 1981.

[75] Alain J. Martin. The probe: An addition to communication primitives. *Information Processing Letters*, 20:125–130, April 1985.

[76] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1:226–234, 1986.

[77] Alain J. Martin. The design of a delay-insensitive microprocessor: An example of circuit synthesis by program transformation. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 244–259. Springer–Verlag, 1989. LNCS 408.

[78] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming*. Addison-Wesley, 1989.

[79] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In W. J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*. MIT Press, 1990.

[80] Alain J. Martin. Synthesis of asynchronous VLSI circuits. Course Notes, VLSI 91, Edinburgh, August 1991.

[81] David May. Occam 2 language definition. In Dick Pountain, editor, *A Tutorial Introduction to OCCAM Programming*. INMOS, March 1987.

[82] Anthony J. McAuley. Dynamic asynchronous logic for high-speed CMOS systems. *IEEE Journal of Solid-State Circuits*, 27(3):382–388, March 1992.

[83] E. J. McCluskey. Fundamental mode and pulse mode sequential circuits. In *Information Processing 62*, pages 725–730. IFIP, 1962.

[84] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[85] Michael Mendler and Terry Stroup. Newtonian arbiters cannot be proven correct. In *2nd Workshop on Designing Correct Circuits*, pages 47–66, Lyngby, January 1992.

[86] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer Aided Design*, 8(11):1185–1205, November 1989.

[87] George J. Milne. Circal and the representation of communication, concurrency and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985.

[88] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[90] David Misunas. Petri nets and speed independent design. *Communications of the ACM*, 16(8):474–481, August 1973.

[91] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on VLSI*, pages 67–86. Computer Science Press, 1985.

[92] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1957.

[93] Steven M. Nowick and David L. Dill. Practicality of state-machine verification of speed-independent circuits. In *International Conference on Computer Aided Design*, pages 266–269. ACM/IEEE, November 1989. ICCAD-89.

[94] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[95] D. Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Proceedings of the 5th GI-conference*, pages 167–183. Springer–Verlag, March 1981. LNCS 104.

[96] S. Patil. Circuit implementation of petri nets. Technical Report Computation Structures Group Memo 73, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1972.

[97] P. Patra and D. S. Fussell. Efficient building blocks for delay-insensitive circuits. In *Advanced Research in Asynchronous Circuits and Systems*, pages 196–205, Salt Lake City, Utah, November 1994.

[98] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI, FN 19, Aarhus University, Denmark, 1981.

[99] Martin Rem. Concurrent computation and VLSI circuits. In M. Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming*, pages 399–437. Springer-Verlag, 1984.

[100] Marly Roncken and Ronald Saejis. Linear test times for delay-insensitive circuits: A compilation strategy. In *Working Conference on Asynchronous Design Methodologies*, Manchester, England, March 1993.

[101] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, 37(9):1005–1018, September 1988.

[102] L. Ya. Rosenblum. Petri nets. *Izvestiya Akademii Nauk SSSR*, 5:12–40, 1983. (In Russian).

[103] L. Ya. Rosenblum and A. V. Yakovlev. Signal graphs: From self-timed to timed ones. In *Proceedings of the International Workshop on Timed Petri Nets, Torino, Italy, July 1985*, pages 199–207. IEEE Computer Society Press, 1985.

[104] Roger Sayle. On the synthesis of modulo-n counters. In *Proceedings of the ACiD-WG/EXACT Workshop on Asynchronous Controllers and Interfacing*, IMEC, Leuven, Belgium, September 1992.

[105] C.-J. Seger. On the existence of speed-independent circuits. Technical Report CS-87-63, University of Waterloo, Computer Science Department, November 1987.

[106] Charles L. Seitz. System timing. In Carver Mead and Lynn Conway, editors, *Introduction to VLSI Systems*, chapter 7, pages 218–262. Addison-Wesley, 1980.

[107] N. A. Starodubtsev. *Autonomous, Antitone, Sequential Circuits.* I. Definitions and Interpretation. *Engineering Cybernetics*, 19(4):111–116, 1982. II. Cyclograms and Their Properties. *Engineering Cybernetics*, 19(5):74–79, 1982. III. Minimization. *Engineering Cybernetics*, 19(6):63–67, 1982. IV. Estimates of the Complexity. *Engineering Cybernetics*, 20(1):92–98, 1983.

[108] N. A. Starodubtsev. *Synthesis of Contol Circuits for Parallel Computer Systems.* PhD thesis, The USSR Academy of Sciences, Leningrad, USSR, 1984. (Russian).

[109] J. Staunstrup and M. R. Greenstreet. Designing delay insensitive circuits using using 'synchronized transitions'. In J. Staunstrup, editor, *Formal VLSI Specification and Synthesis. VLSI Design Methods*, pages 209–226. North-Holland/Elsevier, 1990.

[110] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[111] J. H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, 15:551–560, August 1966.

[112] Jan Tijmen Udding. *Classification and Composition of Delay-Insensitive Circuits.* PhD thesis, Eindhoven University of Technology, September 1984.

[113] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1:197–204, 1986.

[114] Jan Tijmen Udding. Algebraic verification of a modulo-n counter. In *Proceedings of the ACiD-WG/EXACT Workshop on Asynchronous Controllers and Interfacing*, IMEC, Leuven, Belgium, September 1992.

[115] Stephen H. Unger. *Asynchronous Sequential Switching Circuits.* John Wiley & Sons, 1969.

[116] Stephen H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions on Computers*, 20(12):1437–1444, December 1971.

[117] C. H. (Kees) van Berkel. Beware the isochronic fork. Nat. Lab. Unclassified Report UR 003/91, Philips Research Laboratories, Eindhoven, The Netherlands, January 1991.

[118] Kees van Berkel. An asynchronous systolic modulo-n counter and its CMOS realization. In *Proceedings of the ACiD-WG/EXACT Workshop on Asynchronous Controllers and Interfacing*, IMEC, Leuven, Belgium, September 1992.

[119] Kees van Berkel. *Handshake Circuits: An intermediary between communicating processes and VLSI.* PhD thesis, Eindhoven University of Technology, 1992.

[120] Kees van Berkel. VLSI programming of a modulo-n counter with constant response time and constant power. In *Working Conference on Asynchronous Design Methodologies*, Manchester, England, March 1993.

[121] Jan L. A. van de Snepscheut. Deriving circuits from programs. In *3rd CALTECH Conference on Very Large Scale Integration*, pages 241–256, 1983.

[122] Jan L. A. van de Snepscheut. *Trace Theory and VLSI design.* Verlag–Springer, 1985. LNCS 200.

[123] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *International Conference on Computer Aided Design*, pages 262–265. ACM/IEEE, Novemeber 1990. ICCAD-90.

[124] V. Varshavsky, M. Kishinevsky, V. Marakhovsky, V. Peschansky, L. Rosenblum, A. Taubin, and B. Tsirlin. *Self-Timed Control of Concurrent Processes.* Kluwer Academic Publishers, 1990. (Russian Edition Nauka, Moscow, 1986).

[125] V. Varshavsky, L. Rosenblum, V. Marakhovsky, A. Astanovsky, V. Peschansky, and N. Starodubtsev. *Aperiodic Automata.* Nauka, Moscow, 1976. (in Russian).

[126] V. I. Varshavsky and L. Y. Rosenblum. Dead-beat automata and asynchronous parallel process control. In *Proceedings of the First IFAC-IFIP Symposium, SOCOCO-76*, pages 161–164, Tallin, USSR, 1976.

[127] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design, A Systems Perspective.* Addison-Wesley, 1985.

[128] Alexandre Yakovlev. Designing self-timed systems. *VLSI Systems Design*, 6(9):70–90, September 1985.

[129] Meng-Lin Yu and P. A. Subrahmanyam. Hazard-free asynchronous circuit synthesis. In *Working Conference on Asynchronous Design Methodologies*, Manchester, England, March 1993.