THE UNIVERSITY OF CALGARY

PRACTICAL VERIFICATION AND SYNTHESIS OF LOW LATENCY ASYNCHRONOUS SYSTEMS

BY

KENNETH S. STEVENS

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

SEPTEMBER, 1994

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled: *"Practical Verification and Synthesis of Low Latency Asynchronous Systems"* submitted by Kenneth S. Stevens in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

____

Dr. G. Birtwistle, Supervisor & Chairman
Department of Computer Science

____

Dr. J. W. Haslett
Department of Electrical & Computer Engineering

____

Dr. J. Kendall
Dean of Science, University of Calgary

____

Dr. P. Kwok
Department of Computer Science

____

Dr. L. E. Turner
Department of Electrical & Computer Engineering

____

Dr. D. Edwards
Department of Computer Science
University of Manchester

Date ______________________

Abstract

A new theory and methodology for the *practical* verification and synthesis of asynchronous systems is developed to aid in the rapid and correct implementation of complex control structures. Specifications are based on a simple process algebra called CCS that is concise and easy to understand and use. A software prototype CAD tool called *Analyze* was written as part of this dissertation to allow the principles of this work to be tested and applied. Attention to complexity, efficient algorithms, and compositional methods has resulted in a tool that can be several orders of magnitude faster than currently available tools for comparable applications.

A new theory for loose specifications based on partial orders is developed for both trace and bisimulation semantics. Formal verification uses these partial orders as the foundation of conformance between a specification and its refinement. The definitions support freedom of design choices by identifying the necessary behaviors, the illegal behaviors, and behaviors that are irrelevant. Loose specifications and their refinements are written using CCS semantics.

Pure CCS has been modified so that all of the common asynchronous hazard models – delay-insensitive, quasi delay-insensitive, speed-independent, and burst-mode – can be supported by Analyze. The parallel composition semantics have been extended to allow conjunctive broadcast communication. These communication primitives are implemented in a mixed-mode fashion so that pure CCS evaluation or hardware component modeling can be accomplished. A meta transition rule called *computation interference* is also implemented to strengthen the correctness of verifications under labeled transition systems such as CCS.

Congruences hold for conformance verifications in Analyze so that hierarchical verification is supported. A hierarchical top-down directed synthesis procedure is developed. Process logics are refined for practical applications of labeled transition systems to circuits and systems, including a new definition for liveness and deadlock. The target implementation methodology of this work is a parallel set of communicating burst-mode controllers. Burst-mode, developed earlier by the author, is formalized so that Analyze can verify when a specification obeys all the burst-mode rules and can be automatically synthesized into an implementation.

Acknowledgments

As an undergraduate I was fortunate to learn about asynchronous circuits from Al Davis at the University of Utah. This has developed into a disease that I cannot shake, and over 13 years later I am still designing asynchronous circuits and systems. I am also indebted to Al Hayes and Kent Smith, the two advisors for my master's thesis, who immersed me in VLSI, asynchronous circuits, and embedded systems.

After my master's thesis I was fortunate to get a second chance to learn from Al Davis by working for him on the Mayfly project at Fairchild and Hewlett-Packard. He has unquestionably been my greatest mentor, both professionally and recreationally. The satisfaction I achieved in my research with him was easily paralleled by the thrill of climbing a vertical frozen waterfall as he taught me ice climbing in the Canmore "Junkyards". Bill Coates, Robin Hodgson, Ian Robinson, Shane Robison, and Bic Schediwy deserve my thanks for their help, support, and friendship during the Post Office project.

Graham Birtwistle gladly took me on as a PhD student at the University of Calgary in spite of my rather low opinion at the time of the applicability of formal methods to circuit design. My "devil's advocate" side had been considerably bruised by the long hours of hand VLSI design on the Post Office, so when I arrived at Calgary I was very open to new approaches. Graham nursed me through my first real exposure to greek letters other than those attached to the front of fraternity houses. Jo Ebergen, Rob van Glabbeek, Faron Moller, and Chris Tofts contributed invaluable "formal" insights. I would like to thank John Aldwinckle and Ying Liu, classmates at the University of Calgary, for many hours of insightful technical presentations and discussions, and for pulling me out of some tight jams. I would also like to thank my examiners, found on the Approval page, for their contributions and time.

Never have I worked with such a scholar and gentleman as Graham. Beyond the philosophical, technical, and organizational lessons he introduced me to a number of technical people whom I highly respect that have also become my friends. This is a typical exchange I copied from the whiteboard in Graham's lab, many of the entries coming from visitors he had brought in:

> *"Computer simulation is fun!"*
>
> *"Simulation is the intelectual tool of last resort."*
>
> *"At least us last resorters can spell intellectual."*
>
> *"As long as they have little drinks with umbrellas in them, you can send me to the last resort!"*

The atmosphere at Calgary was very good for my health and leisure as well as my intellect. Although Graham is twice my senior in years, I can barely keep up with him when he is on a mountain trail. I have even survived two of his races to the top of Mount Bourgeau and back during his Fall workshops. I would also like to thank Tom Fukushima as a great friend who took me to many of his favorite fishing holes, and Dave Spooner for keeping me in shape by dragging me off to the gym.

I would like to thank Hewlett-Packard for supporting this research through the generous scholarship funding of a Resident Fellowship. In particular I would like to thank Dick Lampman at HP Labs in Palo Alto, who is by far the best manager I have ever met. I miss his friendly visits as he applied "management by walking around" to perfection.

I would also like to thank my parents who let me and my family live in their cabin at beautiful Giles Flats in the Wasatch Mountains of Utah during the last few months of finishing my dissertation. The mountains were inspiring, but I think that perhaps my draw towards the fish in the creek squelched some of my literary inspiration.

Finally, God has blessed me with wonderful family – Dawn, Marina, Lincoln, and Lance. They have backed me in all my endeavors. None of this is worthwhile without them, and they will never know how much I love them.

Kenneth S. Stevens

k.stevens@ieee.org

September 1994

Giles Flats, Wasatch Mountains

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Circuit design has undergone a tremendous explosion of progress in the last decade. In the early 1980's fabrication technology with a feature size of five microns permitted hundreds or thousands of devices to be fabricated on an integrated circuit. Today, millions of devices can be placed on a single circuit of the same area as the feature size has shrunk to half a micron or less.

The cost of designing integrated circuits has exploded as well. Modern fabrication facilities cost millions of dollars, and the man hours required to design a large circuit can be staggering.

Full custom design cannot keep up with the exponential increase in circuit complexity, as design cost and time to market will also increase dramatically, removing the market advantage of these designs for manufacturers. The need for improved tools and technologies that can *rapidly* produce correct circuits rivaling full custom performance and area advantages is clear from market and technology trends. The need for a designer's workbench capable of synthesis and verification was personalized for me after spending thousands of hours of manual implementation on a large, full custom, high performance, parallel integrated circuit.

The application of simplifying abstractions that can be upheld in implementations is the best method for supporting the explosive growth in design complexity. Increasing the level of abstraction lets a designer concentrate on architectural concepts rather than the micromanagement of devices and low level implementation details. The abstractions also facilitate the design automation and formal reasoning about circuit properties. The *digital* assumption is the most widely used abstraction for gates and transistors typically used in VLSI fabrication. Digital design assumes that the devices exist in one of two states, on or off, at high or low voltages.

Although transistors are not necessarily digital, this assumption can be accurate for well designed processes and circuits – high gain devices interconnected in a low-load fashion. These "digital" devices can be connected in such a way as to design more complex *combinational* functions that are themselves digital. A combinational function is one that solely depends on the input set to determine the output function. This reasoning can accurately be employed for circuit synthesis.

1.1 Asynchronous Design

Many functions cannot be represented in a combinational fashion because they rely on the *history* of input sequences. Such logic is called *sequential* logic. How operations are *sequenced* provides the first and largest distinction between digital design styles. There are two fundamental methods of sequencing these circuits – synchronously or asynchronously.

Synchronous digital circuits assume that time is divided into global, distinct, discrete periods that are controlled by the metronomic tick of a global clock. Control and data signals are stored and passed in lockstep on fixed intervals as determined by the clock and its phases. Storage and sequentiality is typically introduced with clocked latches. All logic functions between the latches must be evaluated during the clock period or the circuit will fail.

Asynchronous circuits do not base their sequencing on regular time. Rather, an interface is defined such that function initiation and completion are explicitly signaled. These interfaces always embody *handshaking* to ensure that both the sender and receiver of the communication are ready. This interface protocol is commonly referred to as a *request/acknowledge* handshake. This formal handshake protocol simplifies the design and verification of asynchronous circuits by breaking them into hierarchical modules. The difficult task of creating large parallel systems is greatly simplified since no global analysis is required. Such systems are created by composing and interconnecting the formal interfaces of parallel modules, and verified by proving the interface protocols are upheld. The clean, formal interfaces of asynchronous logic come at the cost of increasing the difficulty of module design as the handshake signals must be free of all glitches and hazards. Hence the more difficult system design aspects are simplified by asynchronous circuits, and the easier challenge of module design becomes more complicated.

Asynchronous circuits are not slaves to a single unifying master (the clock) as are synchronous systems. Clocked, or synchronous, systems can be easily and reliably controlled inside an asynchronous formalism. Much of my early asynchronous design was based on stoppable clocks we termed "Chuck clocks" in honor of their inventor Chuck Seitz. The controlling asynchronous logic views the clocks as request or acknowledge handshakes, and the clocked domain can be fully synchronous. Such mixed mode designs are formally termed **self-timed**. Some timing analysis must be done to assure that the asynchronous control is always prepared to accept the clock handshake given the clock circuit's known frequency. Other delay models are introduced in Chapter 3.

The task of asynchronous controllers is to correctly accept and sequence handshaking signals. Datapath logic must also generate handshakes one of two ways: as true completion signals generated by side effect from the logic operations, or through delay elements that model the timing characteristics of the function. Completion signals are preferable as they accurately model the actual delays in the devices. At times true completion signals can be generated at very little additional complexity. For instance, ORing the two bit lines of precharged RAM cells will produce a handshake indicating the successful completion of the charging and the data valid operations. A good example of the use of delay approximations comes from Sutherland's 1988 Turing Award lecture on the micropipeline architectural style [Sut89]. A delay simulates the operational time for each pipeline stage and is used to control the following stage. Delay approximations permit the utilization of standard (synchronous) datapath components, but the designs will not be as robust. When true completion signals are not generated, the timing analysis can be done *locally*; the functional interfaces remain intact, supporting modular design and verification.

Data transmission is typically carried out using one of two methods: a **bundled data** protocol which requires assumptions on the transmission delays of the data and its associated handshake signals, or with data encoding techniques such as the **dual rail** protocol which encodes the completion signal with the data. Bundled data protocols contain parallel data and handshake paths, and the transmission delays of the two paths should be *equipotential*, or nearly identical. The data is then transmitted before the handshake signal, with the assumption that the handshake will be observed at the destination following the arrival of the data. This places a constraint on the drivers, layout, routing, and loading of the signals. Data encoding protocols use more than one wire per data bit to encode data validity as well as a digital value. This method is more robust and simplifies the layout and routing at the cost of doubling the number of wires required to transmit data.

The asynchronous design style results in the following advantageous circuit properties:

1. Control is localized, supporting modular hierarchical designs. Some global circuit issues such as the power carrying capacity of metal lines and timing analysis are simplified, while others such as clock skew are moot.

2. Power is actively expended only when a module is actively controlling or processing information. Well designed modules only consume leakage current when idle.

3. Performance can be improved as there is no need to wait for a clock edge to begin processing a transaction, and latency can be the device and ambient minimum. Performance estimates and run times for asynchronous circuits result in average delays for the data values rather than requiring worst case values for reliability.

4. Asynchronous inputs and interrupts are a natural aspect of asynchronous design, and will not result in synchronization failures.

5. The circuits are extremely robust because they can adapt to the ambient environment. Changes in temperature, voltage, and implementation parameters will not effect the correctness of the circuit's functions. For example, a hazard free asynchronous circuit designed in scalable rules can operate correctly in a

2 micron or 0.5 micron process, at a temperature of 30°C or immersed in liquid nitrogen, and at varying voltages. Modifying these parameters can effect the power consumption and performance dramatically.

6. The interfaces can be extremely robust, including the adaption to timed protocols [MCS94].

7. Fewer constraints may be required regarding the physical placement and routing of cells, simplifying implementation details.

8. Observability may be easier to achieve using the stuck-at fault model because a handshake signal that cannot make a transition results in deadlock. While faults are commonly observable by this method in practice, a more rigorous theory for fault coverage is required for the differing asynchronous design methodologies.

These advantages have positive ramifications for increasing the level of abstraction that is desperately needed for large high performance designs. Certain aspects of tool development – such as place and route software and timing analysis – can be simplified. However, several aspects of asynchronous design also present challenges.

1. Hazard free design is a difficult, complex process – particularly when using unbounded delay models. Synchronous synthesis and analytical tools are typically not applicable to asynchronous designs because algorithmic assumptions may require the side effects of a global clock, or they do not avoid or detect hazards. Asynchronous designs are also more difficult to create using ad hoc approaches.

2. Although it is difficult to make accurate comparisons, asynchronous designs are probably larger, and therefore more expensive to fabricate, than comparable synchronous designs. Clock lines and drivers are not present in asynchronous designs. However, most implementations require hazard removal techniques that add logic gates. The handshake signals also add wires between logic components. The circuit area of VLSI designs is more sensitive to the wiring requirements than the number of transistors, and it is difficult to project the tradeoff between the reduction in wire complexity due to clock removal versus the increase due to handshake lines. The additional area overhead supposedly varies from 5%–20% for custom designs, and up to 100% for designs synthesized using programming techniques. The overhead is much worse for data-intensive architectures when data encoding techniques (such as dual rail protocols) are used.

3. The handshake protocol is only effective over short distances, because the delay of transmitting a signal is usually proportional to the distance of the transmission. One would not use handshaking protocols for satellite transmissions! Although functionality is not compromised, there is a significant performance sensitivity to the placement of modules. There may also be a performance degradation for level sensitive return-to-zero (or four-cycle) protocols versus the transition based non-return-to-zero (or two-cycle) protocols when the additional handshake cycles cannot be hidden in the computation phase.

The greatest disadvantage comes not from any theoretical disability, but rather from the serious deficiency in tools tailored for asynchronous design practices and a lack of experience and case studies. This deficiency comes in all areas: simulation, synthesis, verification, and testing. The following sections discuss some of the significant progress recently achieved in asynchronous theory, architecture, and tools by a small core of researchers and engineers.

Research into asynchronous circuit design has been carried out sporadically since the early 1950's. Most of the modern theory is founded on the early work done by Huffman, Muller, and Unger [Huf57, Mil65, Ung69]. Although asynchronous systems such as the Illiac [Geo68] had proven asynchronous technology as a viable approach, interest waned in asynchronous design. Clocked systems were much easier to design since hazards did not need to be removed or controlled. The low integration and complexity of the devices allowed global analysis to be carried out efficiently.

Al Davis, Charles Molnar, Chuck Seitz, and Ivan Sutherland bucked the trend with their pioneering work in asynchronous systems [Dav77, CM72, MFR85, MC80, SMSM79]. Technological trends, including the ever increasing levels of integration of circuits, and the maturing of logic systems and formal methods have resulted in a new wave of interest and applicability in these novel circuits.

I studied asynchronous circuits under Al Davis in the early 1980's and have had the fortune of learning from Molnar, Seitz, and Sutherland. In the mid 1980's I joined a team designing a distributed memory multiprocessor called *Mayfly*. This gave me the opportunity to take my asynchronous circuit experience out of the academic world into industry with the development of a high speed CMOS VLSI communication coprocessor chip called the *Post Office*. The techniques at that time for hazard removal or control required a single input change constraint, large delays, or performance inhibiting flip flops [Ung69, CD73]. Since the performance of the Post Office was critical to the success of the Mayfly project, an alternative method of producing low latency control was needed. This led me to develop a new asynchronous control methodology called **burst-mode** which was successfully applied in all controllers of the Post Office.

During the development of the Post Office, interest in asynchronous designs was widely renewed in the academic community on three different fronts. Circuit designers found it ever more difficult to cope with the global constraints of synchronous design and had begun to seriously investigate asynchronous approaches. Software engineers began taking advantage of the modularity of asynchronous circuits to build program based asynchronous synthesis tools. Logicians and mathematicians discovered that asynchronous circuit design is a natural and useful application for their theories. The coming together of these three fields has resulted in a new renaissance of asynchronous design theory.

The next sections present some of the salient achievements of these three groups.

1.2 Circuit Design

One of the most challenging aspects of asynchronous circuit design is the removal of hazards from circuits. Since all hazards cannot be removed before layout with unbounded delay models, the ability to *control* the occurrence of the remaining hazards is also of paramount importance. The following general techniques have been applied to the direct layout of asynchronous circuits with attention to hazard removal and control.

1.2.1 Asynchronous Finite State Machines

The *asynchronous finite state machine* (**AFSM**) based methods take operational descriptions as inputs and produce circuit descriptions as outputs. These methods are typically targeted for CMOS design, creating gate or transistor level circuits. The primary goal of these systems is to produce circuits that are free from as many hazards as possible through compilation techniques, while also minimizing the latency and area of the implementation. This method is targeted for performance sensitive applications. While it is not possible to remove all sequential hazards using the unbounded delay model (see Chapter 3), these methods stand out as producing systems with the greatest performance with relatively few constraints required for hazard control.

They achieve the best performance and smallest size of all the design methods when moderately complex specifications are used. Unfortunately, most of the AFSM based tools use informal operational definitions such as state graphs, which limits the ability to reason about complex systems of AFSMs.

Most of these methods have adopted the burst-mode methodology to achieve higher performance with the smallest exposure to hazards. The most efficient circuits are typically created from descriptions containing from 5 to 32 burst-mode states. While these methods are typically used for control synthesis, they can also be applied to the creation of datapath logic.

The first burst-mode synthesis tool was developed by Bill Coates, Al Davis, and myself to aid in the design of the Post Office [CDS93a, CDS93b]. The tool was dubbed the "Most Excellent Asynchronous Tool" (**MEAT**) after being inspired by the movie “Bill and Ted’s Excellent Adventure”. MEAT was used in the development of 90% of the control modules in the Post Office. A quick introduction to the AFSM synthesis capabilities of MEAT can be found in Section 4.7.

The MEAT prototype did not remove all combinational hazards. A flaw was pointed out by Steve Nowick, who in the process proved that the burst-mode methodology permits the synthesis of totally hazard-free *combinational* logic [ND92]. Nowick also went on to produce a burst-mode synthesis system using local clocks [ND91a] as part of his Stanford dissertation. This interaction with Stanford also resulted in another burst-mode synthesis system [YD92].

A widely known method for formalizing and synthesizing AFSMs was developed by Chu based on *signal transition graphs* or **STG**s – which are a restricted form of Petri nets [Chu87]. This theory has recently been extended to support burst-mode specifications [Chu93].

Another interesting synthesis system was developed by Luciano Lavagno at Berkeley [LKSV91]. This synthesis uses timing analysis to assure that when hazards exist in the circuit, sufficient delays are added to ensure that they are controlled, so that they will not occur in the physical design. Beerel and Ming have also developed analysis and synthesis techniques based on bounded delay methods [BM92].

1.2.2 Architectural Methodologies

Some architectural methodologies are based on design styles which can remove hazards from control logic [Hay81, Hay83, Hol82]. Most of these methods are of restricted applicability, or do not have the performance advantages of the AFSM approach.

Sutherland's micropipeline methodology is the most efficient and successful asynchronous architectural methodology [Sut89]. The micropipeline design style is based on transition logic where the asynchronous rendezvous, or **C-element**, is used to control interaction between hierarchically compositional pipeline stages. There are no direct synthesis tools based on this technique despite the simplicity of the control stages. The most impressive successful application of the micropipelined design style is the asynchronous version of the ARM microprocessor developed by Steve Furber's group at Manchester University called AMULET [FDG+93, Pav94]. The synchronous ARM was the most widely produced RISC processor in the world in the 1980's.

1.2.3 Macro Module Based Design

Programming language methods are not targeted for circuit synthesis on the gate and transistor level. Rather, as in programming language compilers, the program instructions are compiled into a set of predefined primitive operations [BS89, vBBK+94, Ebe88, Mar91]. These primitives are a set of asynchronous macro module components such as C-elements, TOGGLEs, MERGE elements, and so forth, that are typically associated with the semantics of the language constructs. The physical design of these macro cells may require an expert asynchronous circuit designer (or one of the AFSM based tools of Section 1.2.1). The area and efficiency of the resultant circuits are very dependent on the primitives chosen. Higher level primitives typically result in larger, slower circuits while lower level primitives normally result in smaller and faster circuits. A significant advantage of this approach over the other two is the relative ease of porting the system to other technologies.

1.3 Silicon Compilation

Software engineers have made remarkable asynchronous circuit synthesis tools using variants of Hoare's CSP programming language for specifications [AG92, Bru91, Bru93a, GA93, Mar91, vBKR+91, vB92b]. These techniques compile down to primitive agents as described in Section 1.2.3, and interesting academic grade implementations have been built [Bru93b, MBL+89]. Brunvand's approach is fully automated, whereas Martin's approach is a directed synthesis system and requires further specification as lower level logic termed *production rules*. The major drawback of these methods is that the synthesis steps are deemed "correct by construction" so no formal verification is carried out between synthesized implementations and specifications. Deadlock and other properties of a faulty specification may be faithfully implemented, and the source of such errors will be difficult to discover without verification formalisms.

1.4 Formal Methods

Circuit simulation is exponential in time on the input set and device delay variations. Formal proof methods can greatly improve on these results because verification can be carried out hierarchically, values can be abstracted into functions, and regular designs can utilize inductive techniques. The increased complexity of VLSI circuits has produced the demand for logicians to create practical proof systems that can be applied to complex systems.

Several logic systems have been used to verify hardware, including the Boyer-Moore Theorem Prover [Hun86] and higher order logic (**HOL**) [Sys89, GM93, Gra92, Mel88, Coh88]. The complex fine-grain logic models have been successful in accurately verifying data path and leaf cells, but are cumbersome for coarser block-level verifications. Hardware designers and engineers usually consider such formal tools of marginal use or even an impediment to the design process because of the time, effort, and theorem proving expertise that is required to utilize such methods.

Success in automating circuit proof systems has been achieved in the asynchronous circuit domain. Ebergen, Udding, and Josephs successfully use trace theory for the formal design and verification of a class of asynchronous circuits that use the *delay-insensitive* hazard model [Ebe91, Udd84, JU90] (see Section 3.2 for a description of the various hazard models of asynchronous circuits). Dill uses a variant of trace theory to verify circuits using the *speed-independent* hazard model [Dil89]. His tool was invaluable for the verification of the AFSMs in the Post Office. Process algebras, such as CCS and Circal, have recently been applied to the verification of asynchronous circuits [Bai94, Liu92, MM91, Mol91].

While simulation systems such as VHDL and CSP-based programming languages have been successfully applied to the synthesis of synchronous and asynchronous circuits, automated synthesis has not been achieved with systems capable of formal verification.

1.5 Automated Formal Asynchronous Design

Figure 1.1 partitions the asynchronous design problem space into three columns corresponding to circuit designers, software engineers, and logicians, listing some of the top achievements for each group. The Post Office and the asynchronous ARM

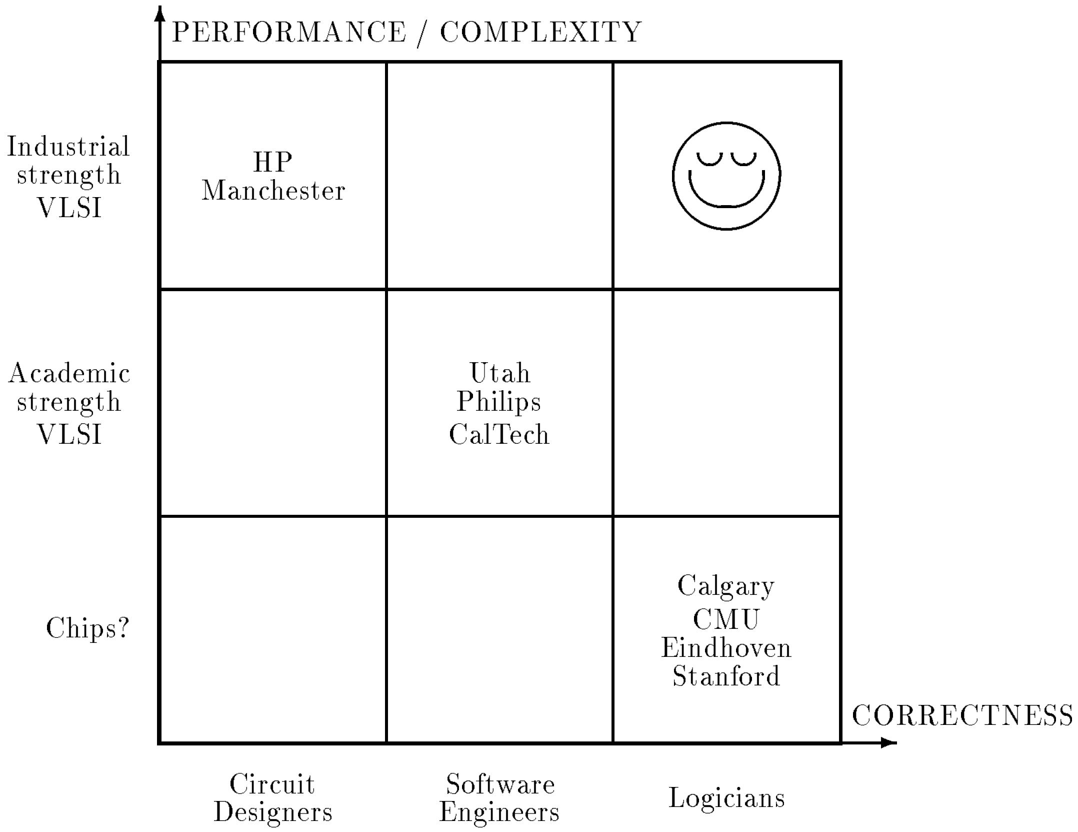


Figure 1.1: Asynchronous Technology Spectrum

at Manchester are engineering feats achieved with almost no tool support by superb circuit designers. On the other end of the scale, mathematical verification has moved from the laboratory into practical use as automated proof systems are being used by engineers for the verification of small circuits and systems. In between is a set of excellent software and systems people who have produced a set of tools capable of automating the arduous task of design synthesis.

The goal of this thesis is to merge the best results from circuit designers, software engineers, and formal mathematicians to work towards a designer's workbench capable of synthesis and automatic verification of asynchronous circuits in such a way that large, complex, parallel ICs can be rapidly prototyped and fabricated.

1.6 The CCS Process Algebra

In the foreword to Milner's book on CCS, C.A.R. Hoare states:

> *"Concurrency remains one of the major challenges facing Computer Science, both in theory and practice. The wide variation in structure and architecture of concurrent machines is now as great as in the early days of sequential machines … Such variation gives rise to confusion and fear of innovation.*
>
> *Fortunately, progress in theoretical Computer Science brings understanding in place of confusion, and confidence in place of fear. A good theory reveals the essential unities in computing practice, and also classifies the important variations. Such a theory was propounded by Robin Milner ten years ago in his Calculus of Communicating Systems."*

Hoare's communicating sequential processes (**CSP**) and Milner's calculus of communicating systems (**CCS**) are *process algebras*, or mathematical systems, that can model and analyze concurrency. This work has applied CCS to the highly concurrent testbed of VLSI circuits, resulting in some practical refinements. CCS is a theoretically satisfactory as well as a practical foundation for the work in this thesis.

CCS relies on the notion of persistent parts, or *agents*, that act independently of each other yet also synchronize. The independence of the actions of agents allows them to proceed *concurrently*, and the synchronization of agents occurs with *communication*. The atomic actions of a system can be represented by a set of symbols called **labels**. These actions can be partitioned into two sets with a "complementation" operation represented by an overbar, extended such that for action α, $\overline{\overline{\alpha}} = \alpha$. Assume that $P, Q, \ldots$ represents processes while $a, b, \ldots$ (and $\alpha, \beta, \ldots$) represent the actions of a system. The occurrence of an action is represented as

$$P \xrightarrow{a} P' \tag{1.1}$$

meaning that as process P performs the action a it simultaneously evolves into the process P'. These actions are represented as **transition** relations over the processes, and if they are all known the behavior of the system of processes is defined.

Communication is defined as a primitive, atomic interaction between processes. The interaction occurs between a label and its complement. This interaction is further defined as *handshake communication* which removes the notion of active (performers or producers) and passive (media or consumers) pairs. If both the label and its complement are offered, the communication can occur; otherwise one of the processes may have to wait. Figure 1.2 shows how processes P and Q can communicate using labels b and c. When a handshake occurs, both P and Q evolve together through an atomic event τ into P' and Q'.

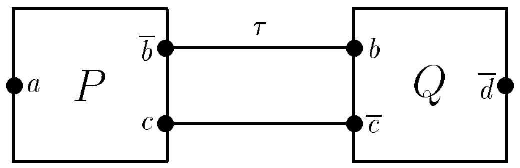


Figure 1.2: CCS Communication Interaction

The transitional semantics of such a language is termed a **labeled transition system**. Since processes are sequential, concurrency arises when independent processes are composed in parallel. When modeling asynchronous systems each component, be it a state machine, register, RAM cell, ALU, or wire, can be modeled as a sequential process that may communicate with other processes. However, most processes can also be decomposed into a smaller set of communicating sequential processes. For example, a certain ALU process could equally be modeled as a parallel set of adder processes. The level of detail of interest will dictate the detail and hierarchy of the description.

1.6.1 Syntax and Semantics of CCS

CCS is a very simple language in both syntax and semantics. The syntax of a CCS process is described by the following BNF description.

$$
\begin{array}{rcll}
\mathrm{P} & ::= & \mathrm{Nil} & \\
& | & \mathrm{Q} & \textit{constant} \\
& | & \alpha.\mathrm{P} & \textit{prefix} \\
& | & \mathrm{P}_1 + \mathrm{P}_2 + \cdots + \mathrm{P}_\mathrm{n} & \textit{summation} \\
& | & \mathrm{P}_1 \mid \mathrm{P}_2 \mid \cdots \mid \mathrm{P}_\mathrm{n} & \textit{composition} \\
& | & \mathrm{P} \setminus \mathrm{L} & \textit{restriction} \\
& | & \mathrm{P}[f] & \textit{relabeling}
\end{array}
$$

The set of actions that an agent can perform is called its **sort**. The special agent Nil can perform no actions, therefore it is the deadlocked or stopped process and its sort is the empty set of actions. CCS is given semantics by induction over the above structure for agent expressions. The semantics are in terms of the labeled transition system, defined as

$$(S, T, \{\xrightarrow{\alpha} : \alpha \in T\}) \tag{1.2}$$

where S is a set of states (or processes), T is a set of transition labels, and the transition relation $\xrightarrow{\alpha} \subseteq S \times S$ for each $\alpha \in T$. The transitional semantics are defined by inference from the transition rules of Figure 1.3 where each rule will have zero or more hypotheses and a conclusion.

$$
\begin{array}{llll}
\textbf{Act} & \dfrac{}{\alpha.E \xrightarrow{\alpha} E'} & \textbf{Con} & \dfrac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad (A \stackrel{\text{def}}{=} P) \\[2ex]
\textbf{Sum}_1 & \dfrac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'} & \textbf{Sum}_2 & \dfrac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'} \\[2ex]
\textbf{Com}_1 & \dfrac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F} & \textbf{Com}_2 & \dfrac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'} \\[2ex]
 & \textbf{Com}_3 \; \dfrac{E \xrightarrow{\alpha} E' \quad F \xrightarrow{\overline{\alpha}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'} & & \\[2ex]
\textbf{Res} & \dfrac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L} \; (\alpha, \overline{\alpha} \notin L) & \textbf{Rel} & \dfrac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}
\end{array}
$$

Figure 1.3: CCS Transition Rules

The Act rule, syntactically using the '.' operator called **prefixing**, is the building block for sequential operations. For example, the agent $\overline{req}.ack.$Nil can do a $\overline{req}$ action, followed by an *ack* action, and nothing more. The constant rule Con defines references to processes, so we can now create recursive, nonterminating agents. For example, a TOGGLE can now be defined as:

$$\text{TOGGLE} \stackrel{\text{def}}{=} a.\overline{b}.a.\overline{c}.\text{TOGGLE} \tag{1.3}$$

This definition says that after the first a input the TOGGLE will produce a $\overline{b}$ output. After the second a transition a $\overline{c}$ transition is produced. The behavior then repeats.

Nondeterministic choice is modeled by the **summation** operator '+' using transitional rules Sum_1 and Sum_2. A process with summation can behave nondeterministically like any of the summands. The C-element can now be defined by:

$$\text{C-element} \stackrel{\text{def}}{=} a.b.\overline{c}.\text{C-element} + b.a.\overline{c}.\text{C-element} \tag{1.4}$$

The C-element can behave like either of the two parts; if the a action is taken first, then it evolves into the agent $b.\overline{c}.$C-element.

Applying the **relabeling** function f to agent E results in an agent that behaves like E where the labels have been changed according to the function f as expressed by rule Rel. The relabeling function is syntactically expressed as $[new/old]$ where all occurrences of the label *old* have been replaced by the label *new*. Relabeling is typically applied to library templates (such as AND gates) where the default labels must be instantiated to the names being used for the specific circuit interconnections.

Concurrent operation is modeled with the **composition** operator '|' using the Com transition rules. Signals can transition independently, expressed by rules Com_1 and Com_2, and signals with the same label can synchronize in an atomic communication (rule Com_3). **Restriction**, syntactically represented as a set L, is used to prevent the restricted agent from actions in the set L, as defined by the side condition to rule Res. This internalizes labels that can communicate in a parallel composition by allowing the internal synchronized action τ to proceed uninhibited, and prohibiting the independent actions of the labels from rules Com_1 and Com_2. *All* communication actions should be restricted in hardware descriptions using CCS.

1.7 Thesis Structure

Chapter 2 uses the fully asynchronous Post Office chip as motivation for higher levels of abstraction and tools supporting asynchronous design and synthesis. The *Mayfly* distributed memory multiprocessor developed at HP is briefly introduced, along with the role of the Post Office in that system. Some advantages and disadvantages of the asynchronous implementation of the Post Office are discussed. Some of the contributions that grew out of this experience of designing an industrial asynchronous chip are included.

Chapter 3 introduces hazards and how they impact asynchronous designs. Asynchronous delay and hazard models are described. The most common combinational and sequential hazards are described with examples, including problems with certain common circuit constructs. Since *no* synthesis system can be hazard free, techniques for hazard removal and controlling unremoved hazards are discussed.

Chapter 4 formalizes burst mode, which I developed in the early stages of the Post Office implementation, in terms of specification and implementation requirements. This permits the automatic verification of terminal burst-mode specifications for the synthesis system presented later in this thesis.

Chapter 5 formalizes the CCS labeled transition system and defines several useful properties including trace equivalence, bisimulation, determinacy, and confluence. The weaknesses in trace semantics are pointed out, motivating the need for stronger bisimulation semantics. A new partial order called **conformance** is introduced. Conformance is aimed at hardware verification and is formally applied to trace and bisimulation semantics formally, and illustrated with several examples.

Chapter 6 introduces Hennessey-Milner process logic and the Modal-μ calculus. Temporal logics are applied to property testing of asynchronous systems, including a new definition for liveness and deadlock. Other invariant properties that are necessary for complete verification are formalized. A comparison between using process logics and conformance is made for circuit verifications.

Chapter 7 unifies this work in terms of a prototype software tool capable of automatic verification and directed synthesis called **Analyze**. The problems in the CCS notation that prevent CCS from modeling hardware are discussed, including solutions that extend CCS in such a way that retains its advantages of specification clarity. New transitional semantics are presented for these changes. The automated functions of Analyze are discussed. Minimization is an important step of efficient verification, and a new minimization algorithm is presented for branching time bisimulations. Computation interference is then formalized. The steps necessary to formally verify a valid burst-mode specification are shown. The high level top-down synthesis process is then described, including the support supplied by Analyze.

Chapter 8 contains hindsights gained from the development and limited application of the Analyze tool, and areas of further research are discussed.

1.8 Contributions

The major contributions of this dissertation include the following:

1. A software prototype CAD tool called *Analyze* was developed and described in this dissertation. It is the first tool to utilize multiple equivalences appropriate for hardware. It also includes all of the common hazard models of asynchronous circuit analysis and verification. The tool is designed using compositional methods and is one or more orders of magnitude faster than the Concurrency Workbench for comparable problems.

2. A new theory for loose specifications based on partial orders is developed. Partial orders sufficient for verification of asynchronous hardware systems are formalized using both trace and bisimulation semantics. Formal verification uses these partial orders as the foundation of conformance between a specification and its refinement (possibly as an implementation).

3. Weaknesses in the CCS labeled transition system have been formally fixed with new transition rules and a meta evaluation rule based on computation interference principles which allows direct representation and analysis of asynchronous hardware modules as CCS processes. Analyze implements these changes in a mixed-mode fashion, allowing standard CCS transition rules as well as the new conjunctive parallel composition operator.

4. The burst-mode model remains an important foundation of this work even though its invention preceded the work in this thesis. The specification and

implementation requirements are formalized and the requirements for the verification of terminal specifications is laid out. Completely automating the validation of a burst-mode specification is not always possible because of the constraints on the environment.

5. A high level synthesis procedure, supported by the Analyze tool, is developed. These steps can be used to test different approaches and can, with supporting module layout and place and route software, rapidly produce verified industrial strength low latency asynchronous systems.

6. New definitions for liveness and deadlock for parallel processes is formalized. Other logic macros are defined that simplify the process of total verification. These logics can be applied to a previously available tool called the Concurrency Workbench.

Chapter 2

Motivation for Analyze

In the late 1970s and early 1980s asynchronous circuits and systems, such as the DDM machines [Dav77], were built out of small scale integrated components on wire wrap boards. State machines were built using gates, EPROMs, muxes, and other devices where the handshake signals were all accessible. Switch and light panels that could intercept handshake signals and logic probes usually sufficed as test jigs. The added complexity and inaccessibility of signals in integrated LSI circuits ([Ste84, Hay83]) increased the difficulty of testing and designing asynchronous systems, but their low level of integration coupled with the modularity of asynchronous protocols made implementations feasible. Even so, the primary goal of all these circuits was to demonstrate operational feasibility and supply academic proofs of concept; circuit performance was not an issue.

However, performance *was* critical to the success of the full-custom CMOS VLSI *Post Office* chip begun in 1987 [SRD86, CDS93b]. The complete chip is the largest and most complex fully asynchronous integrated circuit in published work. It consists of approximately 300,000 transistors and over 95 different finite state machine controllers with an external bandwidth of 300 megabytes per second.

I was responsible for the architectural design and implementation of this chip. The sheer level of integration available coupled with performance requirements created problems which could not be hidden by the modularity of asynchronous interfaces. Many lessons were learned as design techniques were rationalized, mechanized, and formalized. Hindsight proved to be a valuable tutor in many areas, leading to the more advanced and integrated tool Analyze developed herein.

This chapter uses experiences from the design of the Post Office, including small design vignettes, to demonstrate the need for the improved methodologies and tools presented in this thesis.

2.1 Overview of Mayfly and the Post Office

The Mayfly architecture is a general purpose parallel processor, often called a distributed ensemble architecture [DCH+89, Dav92]. Multiple processing elements (or *PEs*) cooperate to solve single complex problems which have been broken into smaller parallel computations. There is no globally shared memory. Task spawning and communication between processes on different PEs are carried out via message passing. The Post Office chip is the communication coprocessor which supports this internode message passing.

Al Davis	General architecture, processor board, & context cache
Bill Coates	I-Cache, Post Office interface board, & PO RAM cells
Robin Hodgson	Runtime software & debugging
Richard Schediwy	Data cache
Ken Stevens	Post Office design & implementation

Table 2.1: Mayfly Design Responsibilities

The Mayfly team consisted of five people with responsibilities as shown in Table 2.1. The top level architecture and programming principles were developed by Al Davis at Fairchild/Schlumberger and Hewlett-Packard in the mid to late 1980s. The programming language and compiler is a parallel variant Scheme (a Lisp language) developed at the University of Utah under the direction of Bob Kessler. The Mayfly hardware was built at Hewlett-Packard. Figure 2.1 shows the major components of a single Mayfly PE. A Mayfly PE consists of two Hewlett-Packard Precision Architecture (or *HP-PA*) RISC processors. One processor is responsible for executing user code (the *EP* or evaluation processor), while the other processor is responsible for all system overhead (the *MP* or maintenance processor). MP tasks include setting up the run list and packetizing messages for delivery. These two processors execute in parallel.

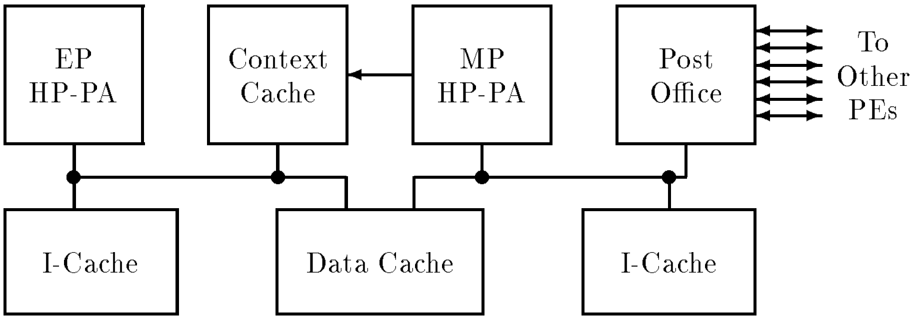


Figure 2.1: Mayfly Processing Element Block Diagram.

The circuits used in the Mayfly design consisted of custom HP-PA processor chips, programmable logic devices, glue logic, memory components, and the full custom Post Office chip. Al Davis designed and built the processor motherboard and novel context cache. Parallel data structure access is facilitated by a 4-page dual ported data cache, built by Richard Schediwy. Bill Coates designed and built the instruction cache and Post Office interface board.

The Post Office architecture and communication topology were designed by myself [Ste86]. The design was taken from concept to a complete VLSI implementation between 1985 and 1991. The Post Office handles all physical communication aspects of message passing in the Mayfly processor. It includes subsystems for handling adaptive routing, buffering, transmissions and retransmission, congestion and deadlock avoidance. First silicon was complete in February 1991. The final version was completed at the University of Calgary and fabricated in November 1992.

The topology (shown in Figure 2.2) and architectural design were created during 1985 and 1986. Helios [Kra85], a distributed simulation tool which ran on networked Symbolics Lisp machines, was used for register transfer level simulations. The Post Office was implemented as a single VLSI integrated circuit, and was laid out entirely by hand using the Electric system [Rub87]. I designed and implemented the entire chip, including the pads, with the exception of the RAM cells and driver circuitry which were laid out by Bill Coates. Simulations of the layout used COSMOS [Car], a switch-level simulator. I tested all the fabricated chip fragments on an IMS tester. The complete chip was tested in a single Mayfly node. Robin Hodgson wrote device drivers and runtime system software. He tested the Mayfly and first and final silicon of the Post Office extensively, although I did much of the initial testing of the first silicon.

The Mayfly interconnection network is a hexagonal mesh wrapped as a twisted torus resulting in the provably minimal diameter [Ste86]. This creates what is known as a processing *surface*. Surfaces have hexagonal boundaries themselves and can be interconnected by abutment in a hexagonal mesh to form a two-level "recursive" topology. The Post Office is therefore a seven ported device. It physically connects to six other adjacent processing nodes in the surface via six 8-bit bidirectional external ports. There is also an internal 32-bit word-wide PE port through which the Post

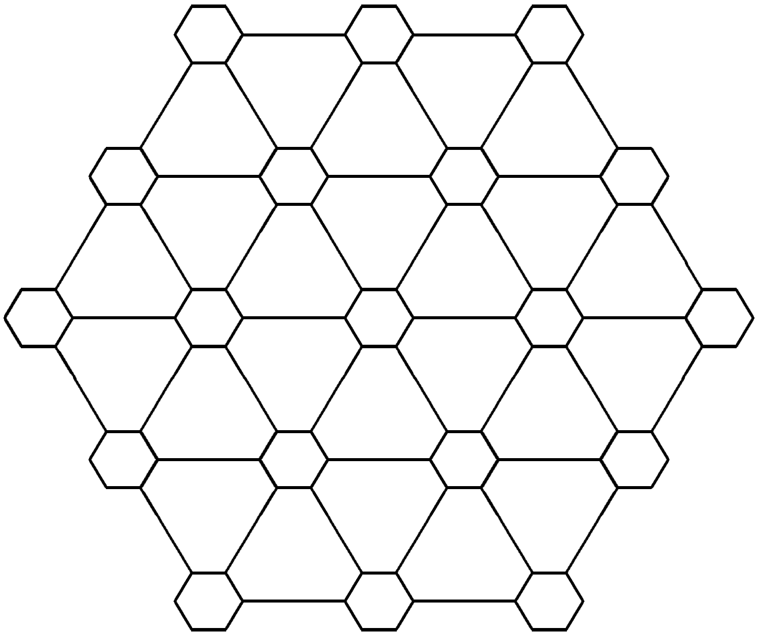


Figure 2.2: Mayfly Interconnection Topology

Office can access the processor cache and local memories for message retrieval and delivery. The Post Office design permits simultaneous transmission on all seven ports. Measured performance in a seven node Mayfly prototype indicates that all six external ports can sustain transfers at an average rate of 50 megahertz, for an aggregate network delivery bandwidth of 300 megabytes per second.

Performance is critical to the Post Office since communication latency is key to the distributed memory Mayfly system. Bandwidth utilization of the links between the Post Office chips must be optimized to achieve a performance that scales with the architecture. Hence a packet switched system was chosen. Virtual cut-through [KK79] was employed because it allows packets to be forwarded to the next destination as soon as the header is received, resulting in a "pipelined" delivery across many chips. Packets which cannot be forwarded immediately are buffered centrally in the

Post Office chip in order to free the external link for other packet traffic. When the destination ports are free, the packet will be forwarded through one of those links. Cut-through is not used in the source and destination Post Office chips; the packets are placed directly into the buffer pool. Although this increases message latency, it insulates packets from any delays that may be encountered in the software protocols that load and unload packets from the Post Office and the Mayfly PE. This results in better utilization on the communication links, and permits the implementation of the external ports to use the smaller dynamic logic (as opposed to the static logic required for the PE interface and the central buffer pool and logic).

The Post Office effort was challenging for several reasons:

1. It was a pioneering coprocessor for distributed ensemble routing architectures. Its design preceded CalTech's wormhole routing [DS87] and multi-queue architectures [TF92].

2. The design is massively parallel, with a complex control structure.

3. The Post Office is an asynchronous chip placed into a synchronous environment.

4. It is the most complex fully asynchronous single integrated circuit in published work.

5. It was built in a commercial environment where performance was an important aspect.

2.2 Asynchrony in Mayfly

2.2.1 Features

The Post Office is an island of asynchrony in an otherwise fully synchronous architecture. The processors in the Mayfly are synchronous HP-PA machines using synchronous bus protocols. The decision to design the Post Office as a fully asynchronous part was based on a number of factors, including the robustness of asynchronous interfaces, scalability, and the desire to build a practical, complex device which is superior to synchronous techniques. The scalability of the Mayfly architecture is probably the single most important argument in favor of an asynchronous Post Office design. The physical extent of the Mayfly architecture is formally unbounded, and the size of an implementation is only limited by the size of the address word. The current Post Office chip supports instantiations of up to 519,841 PEs. The ability to arbitrarily scale the architecture poses serious technical problems if a global clock is necessary to synchronize operations. Clock skew can be a problem in itself for synchronous design as technology progresses [Bak90]. For extensible systems such as the Mayfly where the PE count is unbounded, synchronizing all of the nodes with a single clock becomes intractable.

The robustness of functional, asynchronous interfaces removes the problems of clock skew and simplifies link arbitration and transfer synchronization. Mayfly processors are composed by simply plugging the Post Office links together (subject to topological constraints). Each PE in the multiprocessor contains a local crystal and a clock generator that runs at its own clock speed. Processor speeds for communication between PEs are irrelevant due to the asynchronous interface. One PE in the

HP prototype running at an internal clock speed of 16 MHz communicates perfectly well with another running at 64 MHz via the Post Office chips.

Additionally, the arrival times of packets from the external ports and the PE can be completely random. In a synchronous system, these arrivals would have to be normalized to the system clock, resulting in slower delivery. Receptive asynchronous systems begin processing packets as soon as data arrives [NDDH93]. This robustness of interfaces is also being investigated for commercial applications in noisy environments [MCS94].

The low power nature of asynchronous architectures was one further advantage demonstrated in the Post Office. Asynchronous circuits contain fine grain, dynamic power management due to the handshake protocols. Each idle Mayfly PE requires 30 amperes of current at 5 volts. By way of contrast, the Post Office, which is the only asynchronous part in the system, uses only 2 milliamps when idle.

2.2.2 Problems

In a clocked system, care must be taken to assure that the clock signal has a short rise and fall time, that noise is minimized (ringing, overshoot and undershoot, etc.), and that the clock is driven to the power and ground rails. The same restrictions exist on the handshake signals in asynchronous systems. Although there are many more handshake signals than clocks, handshakes are generally localized between pairs of controllers, on the same integrated circuit, with low capacitive loading. For example, the clock and drivers inside the synchronous alpha chip are global and highly loaded taking up 30% of the chip area and nearly 60% of the power, resulting in considerable technical problems [Com92]. When asynchronous handshake signals are not local, such as between processing elements in the Mayfly, care must be taken to assure that failures do not occur due to violations of the assumption that signals are "digital".

In the Mayfly prototype, each processor had a separate clock generator and power supply. Mayfly PEs are easily composable through the Post Office because the functional interface scales well and is not subject to failures due to synchronization, arbitration, or device speed or clock speed variations. However, communication was susceptible to failures when (i) voltages varied significantly between nodes, (ii) when crosstalk was a significant problem, (iii) when the impedance of the drivers and receivers were not matched such that ringing occurred, and (iv) due to current variations that cause power supply noise. These faults are all due to physical properties that violate the digital assumption, and they can be cumulative. As an example, the *request* handshake signal, driven from the power supply of one Post Office PE, may be received by another Post Office chip with a different power supply. If there is a significant difference in voltages, switching thresholds may not be reached, and very small amounts of noise on the line could cause the receiver to perceive unintended changes in the binary value on the line.

Noise perceived as a switch in logic levels on handshake signals in asynchronous systems and in clocked systems can both result in failures, but they may be more severe in asynchronous machines. Illegal voltage changes to a state machine can result in the circuit deadlocking or switching to an improper state. This generally will result in illegal outputs which can ripple the effect if the outputs are also control lines. Noise can result in similar effects in the controllers of a synchronous system.

The final version of the Mayfly prototype solved noise and voltage problems between different processing elements. The most significant design concept was to use devices with a high noise margin [WE85], such as Schmitt triggers, on all handshake lines between ICs (or anywhere noise or slow rise time might cause problems). This technique increases the immunity of the chips to noise and voltage variations at the cost of a minor degradation in performance. Other techniques that reduced voltage variations and noise included wiring a common ground to all the processors, using impedance matched shielded cable for intra-PE communications, assuring that the power supplies drive a similar voltage, and lowering the resistance on the power and ground supply lines.

Asynchronous systems can easily control synchronous systems, and locally clocked subsystems are common to asynchronous designs [ND91b]. However, due to the unyielding global time domain, many difficulties can arise when the synchronous system is the master of an asynchronous subsystem.

The HP-PA processors in the Mayfly node use a synchronous bus protocol. This presented a major challenge because the Post Office design had to ensure that the interface to the local CPU would not cause any synchronization failures. Two major failure scenarios exist; one of processing interrupts and one of mapping variations in processing speeds of the asynchronous part to the global clock. There is no guarantee that interrupts or status communication between the Post Office and MP processor will be safely aligned with the clock. Spice simulation and performance analysis of the silicon shows that the worst-case delays of the asynchronous Post Office protocols are significantly less than the synchronous transfer requirements for both reads and writes to the PE registers. In practice, none of the communication faults that have arisen in the Mayfly prototype have been attributed to synchronization failure even though the potential exists.

2.3 Post Office Implementation

2.3.1 Datapath Components

The Post Office contains two major logic classes – datapath logic and controllers. The datapath logic includes the ALUs for routing calculations, counters, adders, registers, RAM, and so forth. The controllers are all burst mode AFSMs. These controllers communicate with each other and cooperate to control the datapath logic using request/acknowledge handshake signals. The AFSMs typically have all hazards removed under the burst-mode hazard model using unbounded delays, and are described further in Chapter 4.

The primary responsibility of the Post Office is to transport data from one location to another. Data paths vary in width from five to 128 bits. The performance oriented design style uses the bundled data protocol to reduce the area and control circuitry and achieve greater performance. The bundled protocol assumes that all the data and associated handshake signals are in an *equipotential region* where signal propagation delay is similar for all lines [MC80]. Standard request/acknowledge handshaking is used with the bundled data protocols. The data signals must be valid before the request is driven.

Most Post Office datapath logic senses the completion of an operation and then asserts the acknowledge line, signaling completion to the requester. For example, the RAM blocks are the size of a packet (1152 bits) organized in 128×9 arrays. Write completion is easily detected by sensing when the word line has been driven. The acknowledge indicating data validity and precharge status for reading a RAM block is accomplished by sensing the voltages on a single bit line pair.

Some datapath circuits and communication lines are also controlled in a *clocked* manner with stoppable clocks. The clocks are generated by a burst mode AFSM concurrently with asynchronous handshake signals with sufficient delay for the clocked datapath circuitry to complete its operations. The minimum delay of these handshake signals is greater than the maximum delay required by the clocked logic.

The external port interfaces contain two “clocked” counters that track the number of transfers. A state machine is signaled when the correct number of transfers has completed to load or unload the RAM and register blocks. The counters are implemented with eight-transistor two-phase dynamic shift register stages. They are clocked at the RAM's access speed (using the bit line or word line completion signals as the clock). The ports also contain routing ALUs and latches. The control circuitry “clocks” these ALU circuitry at the same speed as the external asynchronous handshake transfers across the port.

2.3.2 Arbitration

All control circuitry in the Post Office is implemented with custom AFSMs with the exception of the arbitration logic which requires analog mutually exclusive behavior not easily built into state machines using current technology. Arbitration logic must be used whenever concurrent access to a shared resources is possible.

Standard arbitration serializes access in a nondeterministic fashion. The winner of the arbitration utilizes the resource while the loser waits until the resource is freed. An example of this type of arbitration is the serialized assignment of bus mastership. A second type of arbitration was required in the Post Office, whereby if the resource is allocated to another user, the loser will proceed with other tasks rather than wait for the busy resource to be freed. *Nacking* or *nonblocking* arbitration is required in the Post Office when multiple packets want to utilize the same external port for packet delivery. When the port is busy, the packet should not wait for it to become idle; it should be forwarded out a different port or placed in the central buffer pool.

The Post Office contains 13 nacking arbiters. I found this to be an intriguing circuit with no previous published reference, and posted it as a design and implementation exercise to our peers who are designing asynchronous circuits and tools [ND89, JU90]. The circuit in the Post Office consists of a SEQUENCER (built out of mutual exclusion elements and a few NANDs) and a small state machine.

2.3.3 Features

The Post Office corroborates the benefits of the modularity and composability of asynchronous circuits, particularly considering that this 300,000 transistor full custom circuit was designed and implemented by a single individual. The 300 megabytes per second transfer rate is comparable with communication networks by today's industrial standards. The low standby power of the device is an artifact of the chip's asynchronous design.

The burst-mode constraint and its associated design tool, MEAT, reduced the design time of Post Office AFSMs tenfold. When coupled with Dill's verifier and the complex gate generator, most of the implemented AFSMs were hazard free, increasing the confidence in the correctness and robustness of the design. COSMOS was used to simulate modules as well as the entire chip from pad to pad after I made minor modifications that allowed it to accommodate an asynchronous regime.

2.3.4 Difficulties and Design Flaws

The lack of integrated design tools comparable in quality and scope to synchronous tools became apparent in the Post Office project, as a majority of the implementation effort was spent on tasks that can be automated. Problems with a "synchronous" architecture style based on busses also became evident. This section discusses some of the lessons that were learned in hindsight after embarking on a large asynchronous VLSI implementation project.

Layout. Completion of the Post Office implementation was greatly delayed by hand layout. Automated layout of burst-mode state machines is being developed by Bill Coates and the Stetson project at Stanford University [MCS94].

Hierarchical verification. I was unable to verify the correct implementation of multiple burst-mode state machine modules using Dill's verifier. This was partially due to the lack of tool support for burst-mode and the specification style.

Simulation was the only viable technique to check for correct implementation of module interfaces and system behavior. Simulation is too weak a technique for asynchronous circuit certification as the results are only as good as the timing model and fault coverage of the generated vectors. Pathological failures are difficult if not impossible to discover using the unit delay simulation model employed.

Formal verification is an improvement over simulation as it can test for invariant properties which must hold for the circuit to function properly. The most important invariants are conformance to the specification, safety, deadlock and liveness [Liu92]. Many of these verification tests can be checked automatically and are independent of any particular circuit implementation.

Two serious flaws in the Post Office first silicon were missed by COSMOS simulations but would have been detected by the verification tool Analyze. A dynamic 0 hazard existed on the *ack* line of the external ports at the end of a packet transfer. This was due to a race in the enable and tristate logic to the bidirectional *ack* control circuitry. Fortunately the receiving logic was robustly designed so that it was not adversely affected by the spurious wobbling of the *ack* line at the end of the cycle. A second problem resulted in part of the chip interfaces becoming deadlocked while other parallel interfaces continued to operate uninhibited. This was not observed in the initial simulations, yet occurred regularly in the first silicon. This error required an expensive refabrication of the chip because it disabled the chip from operating correctly once the deadlock had occurred. The source of the failure was extremely difficult to discover using simulation techniques; it was only found after several *months* of work.

Lack of tools. The lack of design and analysis tools resulted in a bottom-up implementation of the Post Office. Although the register transfer level Helios simulations were top-down, there was no way to annotate the Helios model or use it to direct or check the implementation process. The register transfer model did not specify all interface signals, and efficient VLSI designs also required some modifications to the initial design. This resulted in some incompatibilities in the interfaces between several modules which added months of work to the design. Due to design inertia the incompatibilities were usually "fixed" by adding a "wart" module to the design which mapped between the interface differences of the modules because this was simpler than going back and entirely redesigning one or more modules. Design time, area, and performance suffered from this bottom-up design style. The top down synthesis driven nature of Analyze can be the foundation for an effective synthesis system that documents and verifies changes in a design.

Effect of modifications. Slight modifications to architecture and design choices can greatly alter area and performance of a chip. The original Post Office design called for packets half the size of the final implementation. The ballooning packet size resulted in an incompatible floor plan, and a large, slow spine bus. Fabrication requirements forced the larger spine bus to be segmented into three communicating busses. Data for the external ports was extracted from the center of the chip rather than the edge of the active portion. These flaws added 20-40% overall delay to the circuit.

Testability. No builtin self-test or scan path analysis method has been developed for burst-mode state machines. The only method of determining internal state of the chip under failed conditions was to image voltage changes with a scanning electron microscope (or **SEM**). A consistent failure was easily reproduced in the first silicon with a short test vector, but could not be reproduced by the simulator. I was able to get an image of wire excitations with a SEM by repeatedly resetting the chip and exercising it with the vectors which caused the failure, thus uncovering the fault. Two gates in a module were not physically connected in an AFSM even though the layout tool claimed otherwise. This problem was fixed in the layout tool, and in the circuit by sputter etching a connection to allow further testing of the first silicon.

Although SEM testing can be used in restricted instances, it is not a general testing scheme. For example, the deadlock occurrence was dependent on event timings, and was irreproducible under SEM constraints.

Each large module was fabricated and tested as a separate fragment before composing them to form the complete circuit. The external port interfaces, although they passed all simulation tests as stand alone devices, did not function properly when composed together to form the entire chip. This flaw occured due to dynamic charge storage lossage on an internal node. Dynamic logic was used throughout the Post Office chip for increased performance and decreased area when static charge storage was not necessary. When a device remained idle, it was held in the reset state until its operation was again required. The external port controllers contained some modulo-n counters which determined when an entire packet had been delivered [EP92]. These were designed out of dynamic shift registers where one of the bits contained a high voltage and the rest a low voltage. The emergence of the high voltage from the end of the shift register indicated completion. (The shift registers were cascaded to multiply the depth for large counts). The reset port of a counter had been mistakenly connected to the global reset rather than the port idle reset signal. When the external ports remain idle for large periods of time, the high voltage dissipates. This resulted in the counter never indicating completion. The problem was discovered through layout inspection, and repaired in the circuit with micro-surgery by cutting the global reset line with a laser and connecting the port idle reset to the counter reset with sputter etching.

Unfortunately the application of process logics and formal methods cannot uncover dynamic logic flaws. Nor can they be used to weed out circuits with fabrication faults, which is a second important application of scan path analysis. Builtin self-test of burst-mode asynchronous systems, an important aspect of commercial designs, is a topic in need of future research.

2.4 Summary

Some of the environment, architecture, implementation, features, and flaws of the Post Office, a 300,000 transistor full-custom chip that serves as a message coprocessor for a distributed multiprocessor, have been described. The Post Office was fabricated through the MOSIS service on an HP 1.2 micron CMOS process and has an area of 11×8.3 mm. The control portion consists of 95 different asynchronous finite state machines, most of which operate concurrently and occupy 19% of the chip area. Datapath circuitry accounts for 45%, pads cover 11%, wire routing occupies 22% of the chip area, and the remaining 3% of the space is unused on the rectangular 84 pin die.

There are seven complete ALUs for routing calculations. The part scales up to a distributed processor containing a maximum of 519,841 PEs (limited by the size of the address word). Each chip has a measured throughput of 300 Megabytes per second external network bandwidth, plus a local CPU transfer bandwidth of 150 Megabytes per second. Figure 2.3 is a photo of the final silicon.

Testing of fabricated chip fragments was done entirely by myself. Hodgson did most of the testing of the final silicon once it was part of a Mayfly processing element. A majority of the tool development (the complex gate tool, MEAT, and COSMOS modifications) was done by myself during the project as well.

The design effort of the Post Office influenced fresh work in a number of areas and has contributed significantly to researchers in the asynchronous design community. Some of these contributions include:

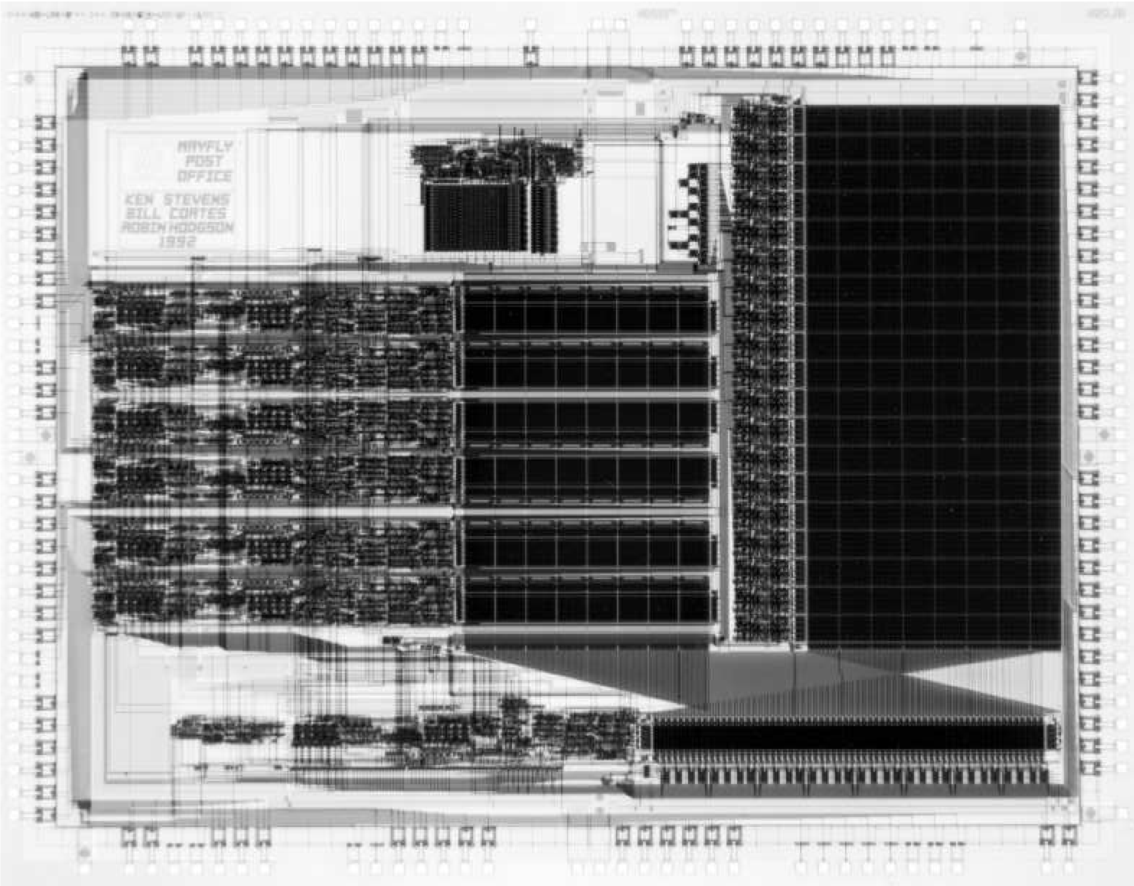


Figure 2.3: Photo Micrograph of the Post Office

1. I developed a new formalism for state machine design and synthesis called burst-mode to cope with parallelism and hazard avoidance in Post Office state machines. This is a significant contribution as it has become the first formalism widely applied for the synthesis of multiple input and output change state machines. Burst-mode also uses a representation that is natural to engineers. Because of these advantages, burst-mode specifications and synthesis is gaining widespread popularity in the research and industrial asynchronous design communities. See under (4,5) below.

2. The lack of tool support is a major impediment to the acceptance and use of an asynchronous design style. Burst-mode synthesis was automated with the MEAT CAD tool. I initiated development on the tool and wrote or redesigned for efficiency a majority of its code. This tool was one of the pioneering asynchronous synthesis systems.

3. Design vignettes of the Post Office are available to the asynchronous community for tool benchmarking and design challenges, including a set of Post Office state machine specifications [Chu93, LKSV91, SMD93], novel CMOS device implementations, and design problems such as the nonblocking arbiter [JU90].

4. Other design and synthesis projects have been spawned as a direct result of the Post Office work. This includes research done at the HP science center at Stanford University. Several dissertations emanate from there [ND91b, YD92, SMD93].

5. Improved algorithms and methods for hazard-free design have been developed as a result of this project [ND92].

During the implementation phases of the Post Office project it became evident that automated synthesis tools are a necessary and viable alternative to hand layout for low latency asynchronous circuits. Once completed, MEAT produced circuit designs comparable in area and performance to the hand designs. Dill's verifier further aided in the AFSM design as it contributed to the removal of all hazards under burst-mode in a majority of the leaf cells. The utility of MEAT and the verifier resulted in a larger portion of the effort to be directed toward the layout and simulation of the chip, two additional areas that can be supported by software tools.

The need for a stronger means of assuring correct system behavior became apparent after the first silicon was fabricated and the deadlock was discovered. Simulation techniques proved ineffective and inefficient in discovering the cause of the deadlock, motivating a stronger formalism for validating system behavior. Although formal methods cannot detect all the failures (such as the dynamic logic error) in the Post Office, the need to make stronger assertions about the properties of a large parallel circuit is a great motivator for the end goal of this thesis: to produce an asynchronous workbench capable of the practical synthesis and verification of asynchronous circuits.

Chapter 3

Hazards

"but it has been delayed until I am indifferent and cannot enjoy it"

Life of Johnson, volume 11, page 262

Boswell 1755

Delays are inherent in signal generation and transmission. A **hazard** exists in a circuit if its output behavior depends on both the internal stray delays of the circuit and on its logic components. A hazard **occurs** when delays in the circuit cause an unplanned output transition. When hazards are present in a circuit, the spurious signal transitions they engender wreak havoc with the chip control logic (and can surface as deadlock). It is thus of paramount importance that asynchronous circuits be hazard free. Since every transition in an asynchronous system counts, spurious hazards become a failure point. Unfortunately creating circuits that are immune to hazards is one of the most difficult and misunderstood aspects of asynchronous design.

Hazards fall into two categories: those that can be removed pre-layout by modifying the logic, and those that can only be **controlled** post-layout by examining and engineering delays in the physical layout to ensure that the hazard will not occur. *No synthesis methodology can create hazard free systems completely independently of the physical implementation parameters.*

The analysis of the source of hazards and methods of hazard removal is complex and not well understood by the design community as a whole. This chapter discusses the various delay and hazard models that are in standard use, categorizes the various types of hazard that can arise in asynchronous circuits, and where possible, shows how to circumvent them. Analyze points out hazards to the designer for removal or control (see Chapter 7). The inability to use syntactic constraints or synthesis techniques to produce circuits free of all hazards necessitates tools for identifying hazards that must be controlled during the layout phase. Analyze is the first tool that can apply multiple hazard models, uses the best methods for spotting them with multiple equivalences, points out more hazard types than other verifiers, and supports hierarchical application so that hazard removal can be deferred to a different environment.

3.1 Delay Models

All logic devices and interconnections introduce stray delays of some magnitude. The magnitude of the delay can be modeled as taking on any value ranging from zero to some upper bound. The **unbounded delay** model assigns an arbitrarily large value for the upper bound. Any canonical circuit description that is hazard free using the unbounded delay model can be implemented hazard free in other technologies, because physical circuits do not exhibit unbounded delays. Engineering delay models, called **bounded delay** models, assign discrete upper bounds on delays. The magnitude of the delays are based on an engineering analysis of the parameters of a target technology and its variations.

Well placed realistic delay assumptions can result in much smaller, simpler, and faster circuits. However, there is a danger that hazard analysis using bounded models can overlook some hazards because of variations in device performance, circuit placement and stray delays, and the ambience. This occurs when deviations in the physical implementation stray outside the parameters of the analysis. Bounded delay systems must be coupled with the physical technology mapping and device layout process to ensure their constraints are upheld. This can pose significant difficulty, particularly when detailed layout and delay information is unavailable. Overstating the delay assumptions can result in larger, slower circuits, which is also a danger of the unbounded delay model.

Circuits designed using the bounded delay model may be less robust than those where all of the hazards have been removed in the pre-layout steps and can only be completely validated after the circuit has been implemented and shown to adhere to the delay assumptions. Unbounded delay hazard analysis creates circuits that are more robust. However, it may not be possible to synthesize the desired functions free of all hazards in the unbounded model.

A reasonable approach is to make asynchronous circuits as robust as possible by removing as many hazards in an implementation independent fashion before the layout steps. Once all hazards have been removed, the implementation stage must control the hazards by using a bounded delay assumption. This model is used throughout the thesis, and unless noted all hazard analysis assumes the technology independent unbounded delay model.

3.2 Hazard Models

A number of hazard models have been designed to meet varying needs, ranging from elegant mathematical models to those that are designed for implementation simplicity. Since the goal of hazard analysis is to create working circuits, one must be consistent and careful in the use of hazard models and in judging their effect on the final implementation.

Each hazard model can be used with the unbounded or bounded delay model. These models assign delay values to *devices* and *interconnections*. The interconnections are typically aluminum, polysilicon, or diffusion wires, but may be optical, infrared, or other communication channels. Accurate hazard modeling requires that the devices be the smallest canonical function units in the technology, such as transistors, AND and OR gates. Grouping smaller components together as macro devices can hide hazards internal to the devices, creating an inconsistencies between the physical circuit and the analysis model.

The **delay-insensitive** (or **DI**) model considers delays on the devices as well as the interconnections. Multiple paths in the interconnection are considered to be independent.

The **isochronous fork** assumption[1] states that the difference in stray delays on a given set of electrically connected wires is insignificant. Hence a signal driven on an isochronous wire set propagates across the interconnection in such a way that it reaches its destination devices "simultaneously"[2].

[1]Isochronous means "uniform in time", or "having equal duration".

[2]The probability of two independent events occurring simultaneously is negligible. Their temporal separation could always be measured by a finer instrument. However, for this discussion, two events are considered simultaneous if their order cannot be distinguished by the logic devices.

The **equipotential region** assumption requires that a *set* of independent wires has indistinguishable stray delays. This model is similar to the isochronous fork assumption. If the stray delays and drivers of the independent wires are approximately equal, and they are driven the same time, then the receivers will not be able to distinguish a difference in the arrival times of the signals.

The **Quasi delay-insensitive** (or **QDI**) model is a DI model where some of the forked interconnections must be isochronous for the circuit to be hazard free. Even when the unbounded delay model is used, these circuits are implementation dependent because a forked signal from a fixed circuit structure in one environment may be DI whereas it can require the isochronous assumption forcing QDI modeling in another environment.

The **speed-independent** (or **SI**) model assigns all stray delays to the devices, assuming that interconnections have negligible delays.

The **burst-mode** model assigns all stray delay to the devices (like the SI model) and additionally requires each module to be stable after each output burst and before the subsequent inputs arrive. A module is stable if no outputs or internal signals are able to make a transition.

The isochronous fork assumption can remove from consideration hazards associated with stray delay in the interconnections. The speed independent model is similar to the delay-insensitive model where all interconnections are modeled as isochronous forks. The equipotential region assumption is typically used to remove from consideration hazards that would otherwise be present in the bundled data protocol. The burst-mode model allows multiple output change modules to be verified.

The delay-insensitive model is the most robust and mathematically elegant model for asynchronous design, but is a mathematical dream as truly DI implementations are typically unrealizable [BE92]. However, when delay assumptions can be localized and contained internal to modules whose layout is controlled, building blocks that are burst-mode, SI, or asynchronous can be supplied which can be assembled according to DI constraints.

Utilizing a speed-independent or burst-mode model throughout the system results in better circuit density and performance as is shown through a typical Post Office circuit in [CDS93b]. There a tenfold reduction in area and twofold reduction in propagation delay in the example was achieved by convolving a group of macro module components into a single burst-mode circuit. For area and performance reasons burst-mode implementations are used throughout this thesis.

3.3 Circuit Hazards

Hazards occur as an interaction between a circuit and its environment. Although hazards are created by devices and delays internal to a circuit, the way the environment interacts with a module can play a major role in the ability to design well behaved circuits. The interaction of the environment and a circuit is so basic that all but one class of hazards is defined using environmental constraints.

Circuits that operate in **Fundamental mode** constrain the environment so that it must hold the inputs into a circuit stable sufficiently long to allow the changes to propagate through the logic, produce the desired outputs, and stabilize internally before a new input set is applied. Asynchronous methods utilize handshake protocols to indicate to the environment when the circuits are in a receptive state and new inputs can be supplied. Detecting receptiveness is straightforward using single input change (**SIC**) and single output change (**SOC**) handshaking. Multiple input change (**MIC**) and multiple output change (*MOC*) handshaking cannot as readily detect when the input or output change has completed, or allow the environment to restrain its response until the circuit is receptive. For example, an early response to a partially complete multiple output change will violate the fundamental mode assumption if new input signals are presented to the circuit before it has completed the output change and stabilized.

Hazards can result and operation is ambiguous when inputs are supplied to a circuit when it is in an unreceptive state. For this reason all but one of the hazards are defined under fundamental mode constraint. When handshaking protocols are insufficient to assure receptiveness and circuit stability and this results in hazards, the layout must be examined to verify that the physical devices have sufficient time to stabilize before new inputs arrive.

There are two broad classes of circuits which may be designed – **combinational** and **sequential**. A logic function is combinational if its outputs can be determined solely from the input values supplied. Outputs from sequential logic are a function of the current inputs as well as the history of previously applied input signals. Sequential logic requires *memory* to record the input history because the outputs cannot be calculated from the inputs alone. The following section defines the occurrence of a hazard in a circuit. Section 3.3.2 deals with hazards in combinational logic, and Section 3.3.3 discusses hazards in sequential logic.

3.3.1 Hazard Occurrences

The occurrence of a hazard can be measured at the output of a logic device. Figure 3.1 shows the output waveforms of the two classes of hazards, which are classified solely by the effect of the hazard, not by its cause.

A **static hazard** occurs when the steady-state output function remains the same when a new set of inputs is received, but a momentary change occurs in the circuit output due to internal delays. Static hazards are classified as either static 0 hazards as shown in Figure 3.1(a) where the steady state is a logical 0, and static 1 hazards as shown in Figure 3.1(b). A static hazard which doesn't drive a signal the full voltage amplitude is called a **runt pulse**.

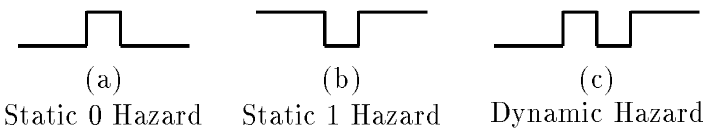


Figure 3.1: Hazards Waveforms in Combinational Logic

The second class of hazard, called a **dynamic hazard**, is shown in Figure 3.1(c). Dynamic hazards occur when a new set of input values produce a change on the output, and the output oscillates before settling to the steady state value.

3.3.2 Hazards in Combinational Logic

Combinational hazards in a circuit are transitory if no new inputs are applied until the circuit stabilizes. Once the circuit has stabilized it will produce the correct output for the current inputs.

Logic Hazards

A **logic hazard** exists in a combinational circuit if, under the fundamental mode assumption, the delays in the circuit result in a hazard. Logic hazards are the most commonly discussed cause of combinational hazards. Such hazards are the result of the particular logic network that is used to implement the output function. Changing the circuit implementation can remove logic hazards – as well as introduce them.

Unger showed that under a single input change constraint it is always possible to synthesize all transitions in a combinational circuit to be free of logic hazards [Ung69]. This is accomplished by adding redundant covers, making the implementation larger. Table 3.1 contains the definition of a contrived specification that demonstrates logic hazard removal techniques for SIC implementations. Figure 3.2(a) contains the Karnaugh map generated from the specification and Figure 3.2(b) shows two implementations. Both circuits contain the ac and $b\overline{c}$ implicants, and one circuit adds the ab AND gate by including the dotted interconnections.

$$
\begin{aligned}
E1 &\stackrel{\text{def}}{=} b.\overline{z}.E1_1 \\
E1_1 &\stackrel{\text{def}}{=} c.\overline{z}.a.\overline{z}.E1_3 + a.E1_2 \\
E1_2 &\stackrel{\text{def}}{=} c.E1_3 \\
E1_3 &\stackrel{\text{def}}{=} b.c.\overline{z}.a.E1
\end{aligned}
$$

Table 3.1: A SIC Circuit Specification

As can be seen by examining the Karnaugh map in Figure 3.2, there are three prime implicants, $b\overline{c}$, ac, and ab [McC86]. The two essential prime implicants, ac and $b\overline{c}$, are circled in the K-map. A logic hazard exists in the circuit designed using only the essential prime implicants. Assume the implementation is in state $E1_2$ (abc:110) and input c arrives. At this point the implicant $b\overline{c}$ is keeping the output

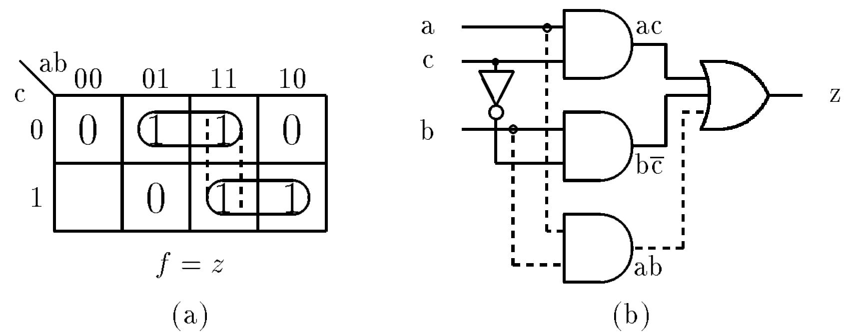


Figure 3.2: SIC Covering Example

high. The arrival of c turns that implicant off, and the implicant ac on. If $b\overline{c}$ turns off faster than ac turns on, then a static 1 hazard may occur on the output.

Unger showed that including *all* prime implicants into SIC combinational logic is sufficient to remove all its logic hazards. The third prime implicant, ab, is shown in the K-map with the dashed box, and connected to the circuit with the dashed lines. During the c transition from state $E1_2$ the implicant ab keeps the output z asserted.

Removing logic hazards from combinational logic in multiple input change circuits is more complex. Including all prime implicants removes all static hazards for outputs that must remain stable during a properly designed MIC transition, as is the case in the SIC covering example of Figure 3.2. However, other methods must be used to assure that no "intervening" implicants are temporarily asserted during a transition when a signal does not remain stable. Such intervening implicants create dynamic hazards. For example, from state abc:101 in Figure 3.2 when both signals a and b change in a MIC transition to state abc:011 a dynamic hazard exists. The ac implicant will unassert during the transition, whereas the ab implicant can temporarily assert, depending on input trajectories and delays. The hazard occurs when the ac implicant and the OR gate become unasserted, and then the ab implicant temporarily asserts long enough to drive the OR gate. Refer to [ND92] for more details on designing logic hazard free combinational logic.

Lesson 1 *Combinational logic can be synthesized free of logic hazards under the input and output constraints of burst-mode.*

Function Hazards

A **function hazard** exists in a MIC circuit if and only if an output changes more than once along a minimum length path of an input transition. Function hazards are caused by the specified functional behavior of a circuit. Unlike logic hazards, function hazards *cannot* be removed by changing the circuit design. This type of hazard does not exist for SIC implementations, and can only arise in a MIC transition which passes through a "cube" of multiple states in a Karnaugh map. Unger showed that with MIC combinational logic, every function with more than one prime implicant may contain function hazards that cannot be circumvented through logic design alone.

Even native logic devices can contain function hazards. Figure 3.3 shows the function specification for a NAND gate. If there is a multiple input change from ab:10 to 01 as shown by the arrow in the Karnaugh map, then a static 1 function hazard exists. If the intermediate state ab:11 is reached, the output may temporarily become a 0, causing the hazard.

The remedies for function hazards include constraints or modifications to the environment or the circuit behavior, and include:

1. Require that the environment provides the inputs simultaneously so that a direct jump to the destination state is achieved.

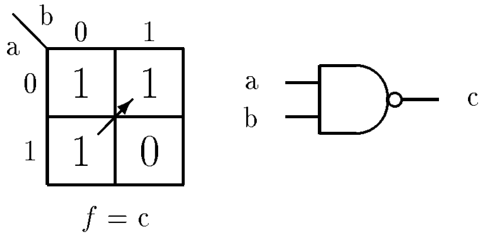


Figure 3.3: Functional Hazard in a NAND Gate

2. Enforce SIC signal ordering by the environment in such a way that a hazard free minimum path is taken for the transition.

3. Redesign the specification so that it will work in all environments by removing the function hazard.

The first two solutions attempt to control the environment so that the hazard will not occur in practice. Neither is a good solution, and the first is generally unrealizable. The hazard is avoided in the second solution if a always lowers before b asserts as the transition will proceed along the path ab:10 $\rightarrow$ 00 $\rightarrow$ 01. If the environment provides the signals in either order in parallel, an expensive SIC sequencing unit must be designed to constrain the order of the signals from the environment. This may introduce other hazards as well.

MIC specifications can be designed without function hazards. More restrictive implementation rules can guarantee that the ab:10 $\rightarrow$ 01 transition is hazard free. Burst-mode is such a system because it restrict specifications in such a way that function hazards cannot arise. As long as the outputs are identical in the cube covering the transition from the start state up to (but not necessarily including) the end state, a function hazard will not exist. The ab:10 $\rightarrow$ 01 transition cube requires the states ab $\{00, 10, 11\}$ evaluate to 1 in order to avoid function hazards, clearly an impossibility for a NAND gate implementation.

Lesson 2 *Function hazards can be avoided in MIC circuits by implementation constraints.*

3.3.3 Hazards in Sequential Automata

Sequential logic is formed by adding memory to combinational logic. Most practical circuit implementations contain sequential logic. The state machines discussed in this thesis are low latency Huffman machines unless otherwise noted, whose construction is shown in Figure 3.4. The sequentiality – and memory – in these circuits is created by feeding the state outputs back in as inputs to the combinational logic. These are the asynchronous finite state machines (or **AFSM**s) produced by MEAT. The hazards discussed in this section are present in Huffman machines as well as all other classes of AFSMs using other different memory means such as latches or C-elements.

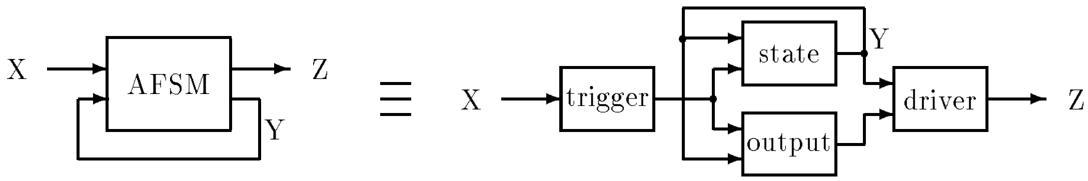


Figure 3.4: Huffman and MEAT State Machines in the Post Office

The hazard discussion of this section assumes that all sequential logic is derived from combinational logic free of hazards by techniques discussed in the previous section. However, *combinational circuits free of logic and function hazards used as the building blocks of state machines do not guarantee a hazard free sequential circuit!*

Almost all state machines contain **sequential hazards**, which are generally caused by two signals arriving at a decision element from two types of paths: one from inputs or combinational logic only, and one from a storage signal (a state variable in a Huffman machine).

The first three classes of sequential hazards that will be discussed in this section assume deterministic combinational logic operating in fundamental mode. The final type of hazard, the delay hazard, does not assume fundamental mode and can occur in both combinational and sequential circuits.

Essential Hazards

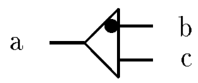
Essential hazards can be present in a sequential circuit if and only if from a starting state, when an input is changed three times, the final stable state is different from the stable state reached after the input is changed only once. Essential hazards can lead the AFSM into an erroneous state. This occurs when the input causes a change in a state variable that leads to the desired stable state. If this state change is perceived by a second state variable *before* the input then the second state variable logic may react to the new state and old input value and switch into an erroneous state.

Essential hazards are similar to function hazards in that they are part of the definition of a function and there is no known automated technique for removing essential hazards, including logic redesign or state variable reassignment.

The TOGGLE element is a classic example of a circuit with essential hazards. Table 3.2 shows the CCS definition, flow table, and logic symbol of a TOGGLE. From any stable state, no trio of transitions will return you to the same stable state. There is a possible essential hazard for each transition. Assume, for instance, that

State	a = 0	a = 1
0	0,-	1,-
1	2,b	1,b
2	2,bc	3,bc
3	0,c	3,c

$$T \stackrel{\mathrm{def}}{=} a.\overline{b}.a.\overline{c}.T$$



Possible Encoding

state	y_1	y_0
0	0	0
1	0	1
2	1	1
3	1	0

$y_1 = b$
$y_0 = a$

Table 3.2: The TOGGLE Element

from state 0–0 of the flow table the input a has makes the transition $0 \rightarrow 1$, which should move the AFSM into state 0–1. This is an unstable state, so the state is changed by moving to row 1. The output $\overline{b}$ will also change, moving the AFSM into the stable state 1–1. The hazard occurs when the state change is processed before the input change in an implementation, sending the state machine through the state sequence 0–0 $\rightarrow$ 1–0 $\rightarrow$ 2–0 $\rightarrow$ 2–1 $\rightarrow$ 3–1, arriving at an erroneous state.

Transient Hazards

Transient hazards exist if from a given starting state, when the input is changed the second time the starting state is reached but a static hazard is possible on any of the outputs. Transient hazards occur in a circuit when a state variable change is perceived by the output logic *before* the input change is perceived. These hazards are similar to combinational logic hazards because they are transient. Transient hazards are also similar to essential hazards because they are part of the function definition and cannot be removed by logic design; the difference being that the hazard is produced in the output logic rather than the state logic. This hazard is extremely common in MIC AFSMs.

D-trio Hazards

D-trio hazards may occur if from a given starting state, when the input is changed three times, the final state is the same as the internal state after one change in the input set, but the second state is different from the initial state. If any outputs change in those three states, a d-trio hazard can occur.

3.3.4 Delay Hazards

The techniques for defining and synthesizing circuits free of combinational and sequential hazards assumes the fundamental mode stability requirement. However, other than the burst-mode model, none of the hazard models of Section 3.2 assume the fundamental mode of operation. This inconsistency between hazard definitions and analysis can result in hazards that slip through synthesis techniques. Delay hazards arise due to this inconsistency and are extremely common in sequential circuits.

A **Delay hazard** can be present when more than one implicant enables a function output in any circuit state. The hazard occurs when multiple implicants are to assert and hold the output high, but only a subset of these implicants stabilize, and the subsequent inputs unasserts the stable implicants before the unstable implicants have stabilized.

For example, the SIC circuit of Figure 3.2 from Page 55 has delay hazards. We will look at the following delay hazard pointed out by Analyze (in the SI analysis mode) using the circuit implemented with all prime implicants.

```
;;; Implementation doesn't conform to specification!
;;; Implementation generates an illegal output!
```

```
;;;     Illegal Signal: 'z  valid spec signals: (b)
;;;   Signal trace: (b 'bc* 'z a c 'c* 'bc* 'z)
```

The implementation behaves correctly until it moves into state $E1_2$ in Table 3.1. At this point the ab implicant is unstable but does not assert. When the c input arrives, the ac implicant also becomes unstable. If the inverter and the $b\overline{c}$ implicant each make a transition before the ab and ac implicants, then the hazardous output that is pointed out above occurs.

This example points out that delay hazards can occur even when using the most restrictive logic class (combinational logic) and environmental constraints (SIC and SOC). It will also occur in sequential logic and with other environmental constraints, and can be exacerbated by the addition of coverings used to remove logic hazards!

Lesson 3 *Synthesis systems do not remove all sequential hazards.*

3.3.5 Example of Hazards in Sequential C-element

The Muller C-element is a standard building block used by many asynchronous systems. This section points out hazards and problems that may occur even in this simple component. A CCS specification of the C-element is shown in Equation 3.1.

$$\text{C-element} \quad \stackrel{\text{def}}{=} \quad a.b.\overline{c}.\text{C-element} + b.a.\overline{c}.\text{C-element} \qquad (3.1)$$

The C-element specification conforms to all syntactic requirements for any asynchronous synthesis system, and can be translated into the state graph and Karnaugh map shown in Figure 3.5. The standard C-element implementation is shown in Figure 3.6 and is synthesized from the K-map coverings.

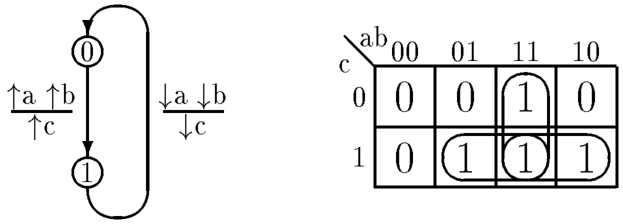


Figure 3.5: C-element State Graph and K-map

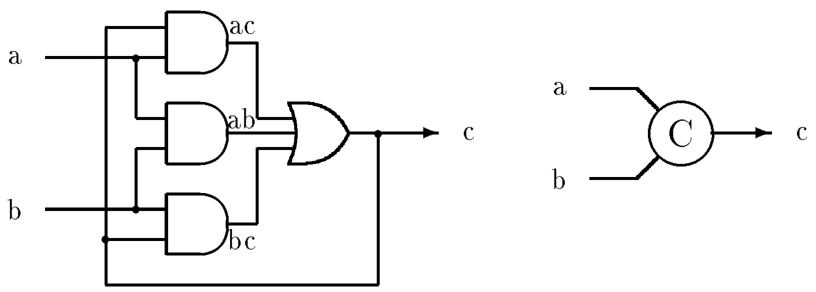


Figure 3.6: C-element AND-OR Implementation and Logic Symbol

This simple circuit obeys all semi-modular and burst-mode constraints. The logic used in the C-element is free of all combinational and sequential hazards including logic, function, essential, d-trio, or transient hazards. Assuming that this results in a hazard free sequential circuit is incorrect. Analyze pointed out eight instances of computation interference in the circuit caused by delay hazards, resulting in static 1 hazards on the output. Table 3.3 is a transcript of one of the error traces of Analyze and is used as an example. Inputs a and b became asserted, setting implicant ab. This asserted the output $\overline{c}$, which in turn enabled the other two implicants, and the C-element arrived in state abc:111 in the K-map of Figure 3.5. However, only the bc implicant asserted before the b input became unasserted. After arriving in state abc:101, the implicants ab and bc became unasserted, allowing the output $\overline{c}$ to transition, resulting in the occurrence of the static 1 hazard. The other seven delay hazard errors are similar, and occur after arriving in state abc:111 without all three implicants becoming asserted before an input changes.

```
;;; ERROR! Computation interference encountered!
;;; Signal 'c in agent C-ELEMENT*
;;; Trace:  (b a 'ab 'c 'bc b 'ab 'bc 'c)
```

Table 3.3: One of Eight SI Delay Hazard Errors in the C-element

Although the C-element contains eight errors using the SI hazard model, it is verified in Analyze as correctly implementing the specification under the burst-mode hazard model. The fundamental mode assumption ensures that the three terms asserting the output are stable before the next input set arrives. This result is logical because the hazard definitions, with the exception of the delay hazard, assume fundamental mode, and the C-element doesn't have any of these hazards in its design.

3.3.6 Other Potential Faults in the C-element

This section examines other potentially serious problems with the C-element design of Figure 3.6, which are also common to many asynchronous synthesis systems and designs.

Figure 3.6 shows that the output $\overline{c}$ is fed directly back into the circuit as a state variable. Faster circuit response can be achieved by using state variables as direct outputs. However, this creates an isochronous fork that conjoins the external environment and the internals of the state machine, destroying the modularity of an AFSM as internal state signals pass directly on to the spatially unconstrained environment. This is an undesirable location for an isochronous fork for the following two reasons:

1. *The fundamental mode assumption can easily be violated.* The environment can act on the output concurrently with the internal logic of the AFSM receiving the state change. A quick response from the environment can violate the stability requirement. Infinitely fast environment response is achieved by feeding the forked output directly back into the state machine. This *always* violates the fundamental mode assumption and nearly always results in circuit failure. Analyze can detect when a direct output to input connection results in circuit failure, but post layout timing analysis must still be carried out.

2. *Global circuit analysis is required.* As can be seen from Figure 3.6, the forked signal $\overline{c}$ is passed directly to the environment as well as internal to the circuit as a state variable. The modularity of the circuit is compromised as the load on the state variable depends on the external circuits it drives and their placement. Timing analysis of the C-element *and its environment* is required to assure fundamental mode is not violated by the fork. Kees van Berkel showed that even when the output of a C-element passes through logic, and the load on the output is greater than the feedback delaying the signal externally, it still can fail as a result of this fork [Rob61, vB92a].

These problems can be controlled and localized by buffering forked outputs that feed back as state variables. That is one function of the driver box in Figure 3.4. This solution usually comes at the cost of a slightly larger and slower circuit, but can improve performance when the outputs are heavily loaded (but this depends on the circuit environment). Localizing the fork using the above technique is essential if the circuit is to be used as a building block in a macro module library.

Burst-mode implementations permit the rendezvous operation of the C-element to be convolved into AFSMs. This can result in better aggregate performance, as well as the removal of the isochronous fork on the signal outputs. For these reasons C-elements were rarely used the Post Office designs.

3.4 Specification Complexity and Hazards

The difficulty of implementing a circuit free of hazards increases rapidly with the complexity of the behavior. As implementations grow over 32 minimized burst-mode states, the additional coverings required to remove logic hazards create delay hazards. The likelihood of delay hazards increases with the complexity of a design. Conversely, Very small specifications – those consisting of less than 5 minimized states – are usually undesirable because of the logarithmic scaling of the binary state representation.

The difficulty of building large hazard-free implementations arises mainly due to the difficulty of removing hazards from the feedback signals. As the number of state variables and logic devices that must share the feedback signals increases, so is the probability of creating hazard-free covers greatly reduced due to the difficulty of covering all cubes and assuring there are no interfering covers [ND92].

Some synthesis systems attempt to reduce the number of dedicated state variables by feeding the outputs back into the circuit as state variables [Chu87, Chu93]. This results in isochronous forks, with their attendant analytical problems, at nearly all outputs! This trick can decrease the number of dedicated state variables (since the outputs need to be produced in any case), but the *total* number of feedback signals (state variables) usually increases dramatically. The probability of sequential hazards (essential, transient, and d-trio), as well as delay hazards, may increase by adding to the total number of state variables, particularly when the excitation pattern of many of the state variables is inflexible.

AFSMs in the Post Office were all characterized by burst-mode state machines. Given a correctly constructed burst-mode description and system behavior, MEAT and other synthesis tools can generate physical device descriptions suitable for integrated circuit fabrication. A graphical burst-mode description that will easily fit on a piece of paper is typically very easy for engineers to specify and understand (its appearance is similar to a Mealy state machine), and is usually of the correct implementation complexity for automatic implementation (from 5 to 32 minimized states).

3.5 Hazard Summary

General implementation independent techniques for hazard removal in MIC combinational circuits have only recently been developed. There is no method of syntactically restricting specifications or designing them in such a way as to remove all sequential hazards using the implementation independent unbounded delay assumption. For example, simple asynchronous building blocks such as the TOGGLE cannot be synthesized without hazards using current technology, and the C-element can only be designed hazard free with great effort[3]. That few combinational asynchronous circuits can be built without hazards has been the topic of some recent papers [Mar90]. Brzozowski and Ebergen discussed the delay sensitivity of asynchronous implementations, and proved that it is theoretically impossible to design many simple circuits without hazards using logic gates [BE92]. This results in the following rule of thumb of asynchronous design:

Observation 1 *Hazard free sequential circuit synthesis is not always possible, and hazard free systems are extremely rare.*

Applying the Speed-independent hazard model with unbounded delays reveals that transient, essential, and delay hazards abound in sequential asynchronous circuits that are built from locally hazard free combinational logic. Table 3.4 lists the hazards discussed herein and the current ability to guarantee that synthesized circuits are free of such hazards.

Guaranteed Hazard Free Synthesis		
Hazard	**Type**	**Circuit Class**
logic hazard	combinational	most circuits
function hazard	combinational	most circuits
essential hazard	sequential	semi-modular circuits
transient hazard	sequential	semi-modular circuits
d-trio hazard	sequential	semi-modular circuits
delay hazard		no circuits

Table 3.4: Hazard Free Circuit Classes

Automated techniques are available that can synthesize most combinational circuits free of all logic and function hazards. There may be some simple constraints

[3]A hazard free DI implementation of the C-element exists, but is a mystery to me how it was coined!

that assure, for example, that a function hazard is not contained in the specification. There is only a limited class of circuits that is guaranteed to contain no sequential hazards. These circuits, called semi-modular circuits, were described by Muller [Mil65]. Such circuits are *confluent* in every state (see Section 5.3.2). There is no class of circuit that can be synthesized with modern techniques that is guaranteed to be free of delay hazards.

A lot of confusion has resulted both within and outside the asynchronous community regarding claims of so-called hazard free synthesis systems. Although these systems may remove certain *classes* of hazards based on the fundamental mode assumption, such as logic hazards in combinational logic, no system is capable of general hazard free circuit synthesis under the implementation independent unbounded delay models. Because bounded delay models require implementation dependent information, they are difficult to evaluate without the circuit layout and parameters. More rigorous and honest reporting must be employed regarding the assumptions used, the weaknesses, and constraints of synthesis systems because it is unlikely that there can be a single hazard model that can be applied from high level descriptions on down to circuit implementations. For example, the DI model, although mathematically elegant and useful for coarse high level evaluation, cannot be applied to physical implementations. Other hazard analysis, removal, and control methods must be used at the physical circuit level. The way in which any particular synthesis methodology treats this dichotomy of hazard modeling must be made evident.

The following actions are necessary to design safe burst-mode circuits, which like any other class of circuits cannot be guaranteed to be synthesized free of all hazards.

Action 1 *Synthesis systems can be created which remove combinational hazards and most sequential hazards.*

- Designing sub-circuits free of all combinational hazards *may* or may not increase the likelihood of a final hazard free implementation.
- Many sequential circuits can be synthesized hazard free, but not all hazards are removed by synthesis constraints.
- Function hazards can be avoided with implementation and specification constraints. This step must be a pre-synthesis procedure because modifying the circuit structure or state assignments cannot remove such hazards.

Burst-mode was developed to assure that designs are free of function hazards and to aid in the development of compact, low latency hazard free AFSMs.

Action 2 *Hazard analysis is required following circuit synthesis to point out where delay hazards and unremovable sequential hazards exist.*

- It may not be possible to specify certain behaviors without sequential hazards (such as the TOGGLE).
- Some or all potential sequential hazards (such as essential hazards) may not be present in a particular design.
- No synthesis tool creates hazard free sequential circuits.

Essential and transient hazards will not be present in an implementation where there is no buffering or inversions of the input signals. However, this logic constraint is impossible in practice and some of the unremovable hazards due to the behavior of the specification will occur. Delay hazards are always possible even after removing all other hazards.

Action 3 *Hazards may be removed using techniques unique to the implementation media.*

Hazards that were not removed in the specification and synthesis stages can be flagged for special treatment and removal. Section 3.7 discusses two techniques used in the Post Office design for removing hazards from synthesized logic.

Action 4 *Remaining unremoved hazards must be controlled in the circuit layout.*

When hazard removal fails, hazards can be *controlled* at the time of circuit layout because hazards are created by stray delays. Implementation technology and layout may require additional investigation to assure that the requirements of the assumptions made by the hazard models used in verifying an implementation have not been violated. For example, the isochronous fork assumption has resulted in real circuit failures [Mar89]. Flagging potential hazards in a circuit for special layout consideration allows one to:

1. attempt to create a layout where the delays will not result in a hazard, and

2. analyze the layout to assure this is the case.

The design methodology of the Post Office successfully took this approach to controlling hazards. Hazards in the AFSMs were all localized to the state machine itself where hazards are easily controlled. Model assumptions such as the isochronous fork and fundamental mode were also restricted to local areas whenever possible.

3.6 Controlling Hazards

All hazards are not removed during circuit synthesis under the unbounded delay model using any of the hazard models presented in Section 3.2. Further, all these hazard models, with the exception of the DI model, contain abstractions which remove certain hazards from consideration. Care must be taken in the layout to assure that the hazards that remain or have been removed from consideration do not occur in the circuit.

Since hazards are caused by stray delays, engineering techniques exist which can organize the delays of a circuit to preclude the *occurrence* of the hazards. Controlling of hazards is implementation dependent and if the circuit is implemented in another technology or with different parameters the hazard may not be controlled using the same techniques. Logic or electrical circuit diagrams are not proof that a hazard is controlled – the physical properties of the devices and layout must be examined. Technology mapping that does not add additional hazards only prevents the increase of layout restrictions. Technology mapping does not control the potential problems that already exist and must be passed information regarding these hazards for reliable implementations to be created. Even so, the layout restrictions for asynchronous circuits are much looser than for synchronous circuits. Implementations can also be made smaller and faster, as will be shown in Section 3.7.2, if some knowledge of the actual delays of the physical device are considered since no physical devices used in circuit fabrication actually demonstrate unbounded delay.

Many synthesis systems control hazards by adding delay to the output of the state logic [Ung69], a solution which is not satisfactory for low latency circuits. A better solution (used in the Post Office) is to use careful layout and circuit design coupled with an inequality timing analysis to assure that the hazards will not occur given the delays in the circuit. Unger presented an inequality on page 179 of [Ung69] which assures that hazards caused by delays hidden with the fundamental mode assumption are controlled. A similar inequality was used in the Post Office and is described in [Ste92].

3.7 Hazard Removal

This section presents two methods used in the Post Office to remove hazards once state machines have been synthesized with MEAT. These techniques have some weaknesses since they may not remove all hazards, have not been automated, and will not work for all classes of logic or circuits. Further study is necessary to generalize, strengthen, and automate these techniques.

3.7.1 Signal Reordering

Figure 3.7 shows the MEAT implementation of the Post Office state machine SBuf-Send-Ctl. Refer to Section 4.7.1 for an explanation of this circuit and its specification. This circuit contains a transient hazard because the *Req-Send* output combinational logic can process a change in the *Y0* state variables *before* noting the change in the *Begin-Send* input signal if the inverter is slower than the OR gate, causing a static 0 hazard. The hazard path is indicated by the dotted line, and occurs in state 2 of the specification (in Figure 4.4) as *Begin-Send* becomes asserted.

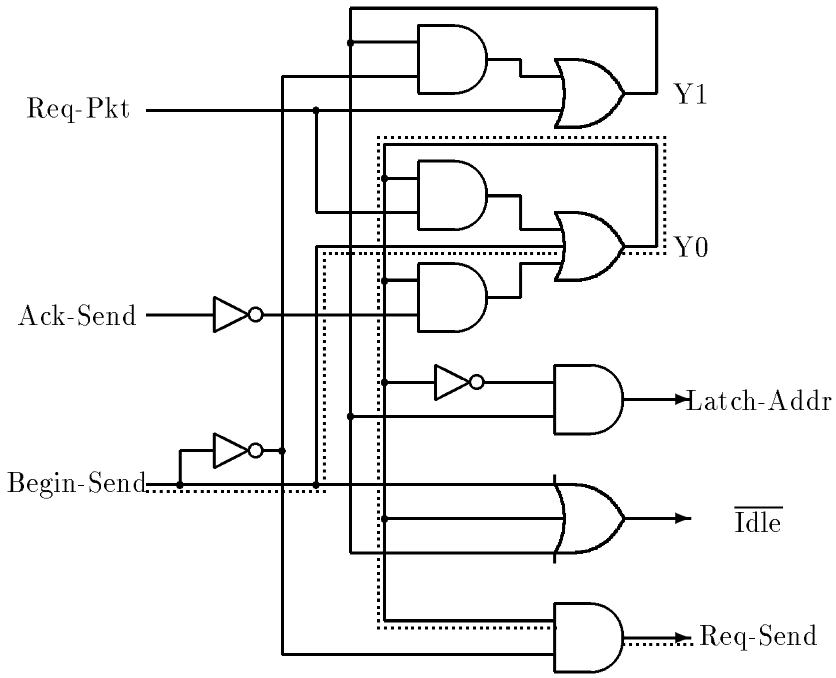


Figure 3.7: SBuf-Send-Ctl Circuit with a Transient Hazard

The first hazard removal technique reorders the sequence that signal transitions are evaluated by adding inverters to a signal path. The transition on $\overline{\textit{Begin-Send}}$ must arrive at the Req-Send logic before the Y0 state logic to remove the hazard. "Double inverting" *Begin-Send* to the Y0 state logic forces the output logic and state blocks to evaluate input transitions in a fixed order using burst-mode or SI analysis so that the *Begin-Send* is accepted by the Req-Send logic prior to the *Y0* logic.

Figure 3.8 shows SBuf-Send-Ctl with the hazard removed by adding an inverter to double buffer the *Begin-Send* signal. Removing the hazard in this manner comes at the cost of additional inverters and a slightly larger circuit. Output latency is not usually increased by this method of hazard removal if the hazard is static, as is the case in this circuit.

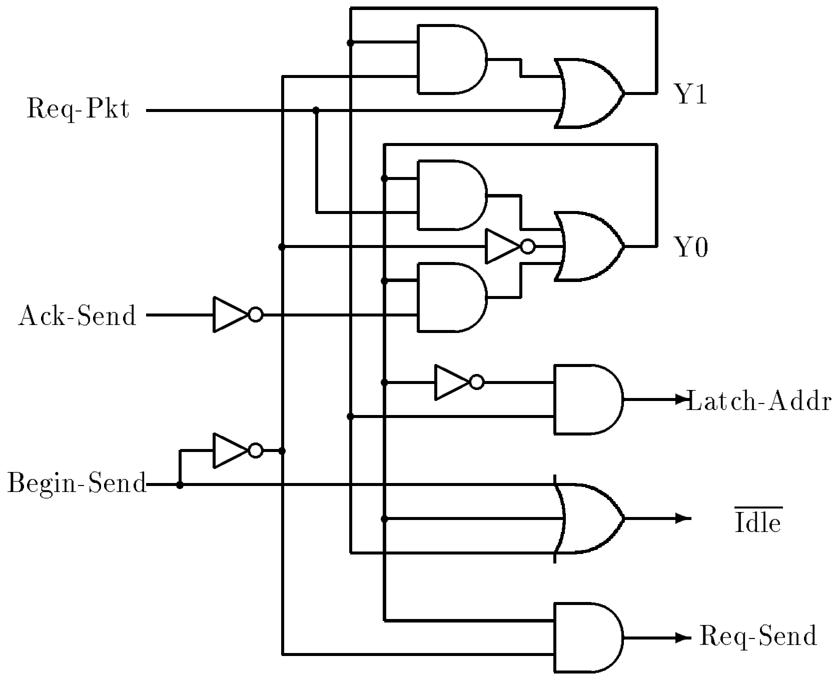


Figure 3.8: Burst-mode Hazard Free SBuf-Send-Ctl Logic

3.7.2 Complex Transistor Gates

Races between gates can be removed by combining the gates into a single complex functional unit designed with transistors. Such transistor structures are referred to as **complex gates**. This is the second hazard removal technique used in the Post Office. The rules for creating these complex gates are part of a tool I developed that interfaces with Electric [Rub87] and produces transistor schematics. Applying the AND-OR implementation of the C-element shown in Figure 3.6 to the complex gate tool produces the single complex gate in Figure 3.9.

This implementation is free of all hazards using SI analysis. The delay hazard in the circuit of Figure 3.6, caused by unequal delay of the three AND gates, is removed by convolving the independent gates into the single complex device.

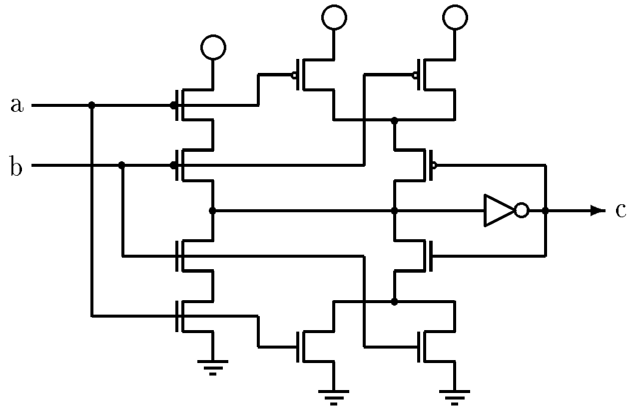
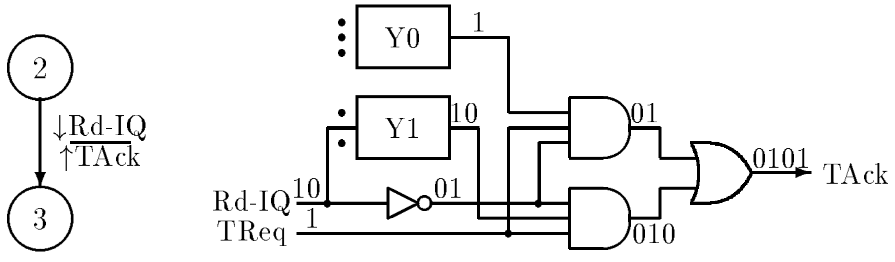


Figure 3.9: Complex Gate CMOS Transistor Implementation of C-element

Another example of complex gate hazard removal is demonstrated with the PE-Send-Ifc AFSM. This state machine controls the external handshake lines on the PE port when an outbound packet is being loaded into the Post Office registers. This state machine is fairly complex, and is not shown here in its entirety. A piece of the burst-mode specification is shown in Figure 3.10(a). The MEAT implementation for the *TAck* signal requires seven AND terms. Only the two terms active in this transition are shown in Figure 3.10(b) using the canonical AND-OR gate implementation.



$$\text{TAck} = \text{Y0}\times\overline{\text{Rd-IQ}}\times\text{TReq} + \text{Y1}\times\overline{\text{Rd-IQ}}\times\text{TReq}$$

SG Segment (a) | AND/OR Implementation with Hazard (b)

Figure 3.10: PE-Send-Ifc Hazard

The dynamic 1 hazard present in this implementation is shown by the signal values on the wires in the figure. This hazard exists because a term (in the bottom AND gate) can become temporarily asserted during the input and state change burst. The top AND gate becomes asserted and remains stable. If the top AND gate is significantly slower than the bottom AND gate, and the Y1 logic is slower than the inverter, then the bottom AND gate can turn on then off before the top AND gate ever fires, producing the hazard as the output can bounce 0–1–0–1 before stabilizing.

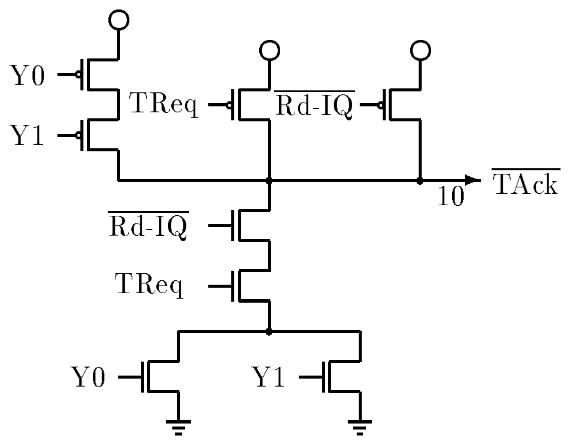


Figure 3.11: PE-Send-Ifc Hazard Removal with Complex Gate

Combining the two AND gates into a single complex gate removes the hazard and the final circuit is smaller and faster. Figure 3.11 shows the complex gate used in the Post Office chip. This gate was created by the MEAT back end complex gate generator. As can be seen, the complex gate removes the hazard because the $\overline{TAck}$ signal cannot be pulled low until both the *Y1* and $\overline{Rd\text{-}IQ}$ signals have changed.

Note that this hazard exists in the sum-of-products form because the combinational logic is not entirely hazard free for MIC logic. The bottom AND gate becomes asserted because it is an "intervening" gate that turns on at an intermediate part of the MIC transition. This hazard may be removed with careful synthesis and additional logic and states. This example shows that extra logic required to remove hazards in the sum-of-products form may not be necessary if the function will be implemented as a complex gate.

The example of Figure 3.2 confirms that at times the additional coverings are necessary to create a hazard free complex gate implementation. Using only the essential prime implicants ab and $b\overline{c}$ produces a complex gate of Figure 3.12(a) that does not remove all of the hazards in the complete circuit as the inverter delay creates static hazards on the output. No hazards are introduced with complex gates, and the circuit of Figure 3.12(b) is a hazard free implementation.

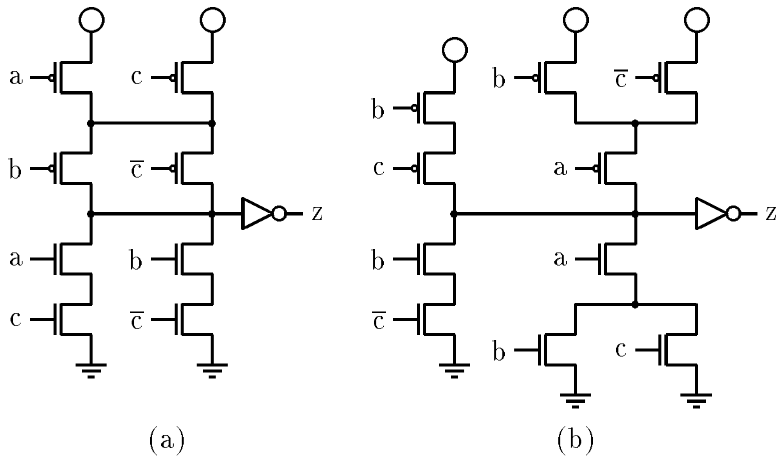


Figure 3.12: Hazard Free SIC Circuits as Complex Gate

3.8 Summary

This section presented a description of delay models and hazard models commonly used to describe and analyze asynchronous circuits. The definition and causes of common hazards and their avoidance was also presented with examples. Most hazards are defined based on a fundamental mode requirement, yet most hazard analysis models do not use the fundamental mode stability requirement.

Hazards cannot be removed from all circuits in a technology independent manner. Specification and synthesis procedures can be used to automatically synthesize combinational circuits free of all but delay hazards. Sequential circuits cannot be automatically synthesized free of hazards under the unbounded delay model. For some circuit specifications, hazard free implementations have been proven impossible, and for others hazard free synthesis cannot be accomplished with current technology. Many instances arise due to the divergence between hazard definitions that assume stability and analysis methods which do not. Hazards are sensitive to circuit structures, and AFSMs of moderate complexity with fewer implicants and state feedback signals typically ease hazard removal while allowing low latency.

Analysis tools are necessary to identify unremoved hazards by a posteriori evaluation. Two methods for removing hazards following circuit synthesis were presented. When the hazards cannot be removed with these and other techniques, delays in the physical layout must be organized so that the occurrence of the hazards will be prevented. Assumptions of the hazard model must also be verified. Delay inequalities, like the one used with the CMOS AFSMs in the Post Office implementation, can ensure that the remaining hazards have been controlled.

Delay-insensitive macro module synthesis systems use many components, such as the C-element, that contain hidden isochronous forks and hazards. However, when the hazards and forks are localized to AFSMs then compact, low latency burst-mode designs can be synthesized which are as reliable and robust as DI circuits.

Chapter 4

Burst-mode and AFSM Circuit Synthesis

Key to any hardware design is the correct construction of physical devices. A major challenge in creating the final circuit is the process of unifying the physical behavior of components with a suitably abstract concept of the desired external operations. Burst-mode simplifies this process by restricting specifications in a way that makes them easier to build correctly as it tends to more closely match the designer's mental model of the hardware. One of the most significant contributions of burst-mode is the ability to design hazard free multiple input change combinational logic.

This chapter begins by discussing burst-mode specifications in Section 4.2 followed by the implementation and specification requirements. The specification requirements are formalized in terms of CCS. This formalism allows the verification of a CCS agent description as a valid burst-mode specification which is suitable for implementation by MEAT or other tools. If behavioral descriptions can be validated then the synthesis process can formally prove correctness from a high level specification on down to the specification of each state machine. These specifications can then be passed directly to MEAT or an analogous tool to generate masks which are ultimately fabricated. This chapter concludes with a design example from the Post Office.

4.1 Burst-mode

Before building the Post Office I had designed many small asynchronous systems, and one moderately sized asynchronous integrated circuit [Ste84]. However, the complexity, low latency requirements, and inherent parallelism of the Post Office made most of my previously used asynchronous design styles impractical. While single input change (or **SIC**) techniques were well developed, they were not directly applicable to Post Office control due to the amount of parallelism present. When several inputs to a SIC AFSM may change simultaneously, they must be filtered or combined with input conditioning which makes the design more difficult, and area and performance suffer. Some MIC techniques overly restricted the arrival time of signals. The stored state model I used previously for integrated circuits was a fairly unrestricted multiple input change (or **MIC**) model, but its implementations were also very large and the response time was slow [Hay81]. Other MIC methods required inertial delays (delays that can filter out small duty cycle transitions) or delays on the feedback lines which were also unsuitable for performance oriented designs.

My solution for implementing low latency state machines designed for parallel process forking and synchronization was to invent the burst-mode design style [Ste92, CDS93a]. Performance was further improved by transforming sum-of-products descriptions into complex gate CMOS implementations. Burst-mode permits a restricted form of MIC signaling which supports hazard free sequential logic and simplifies the implementation of hazard-free combinational logic in asynchronous finite state machines. It also results in small, intuitive specifications.

There existed a serious lack of design tools at the start of the Post Office project. Our tool set consisted of only a hand layout editor and design rule checker, and register transfer level architectural simulator. The need for some synthesis tools became more evident as the project wore on. I could spend a week or more designing, laying out, and checking a single burst-mode state machine. This time would double if errors were found, and the hand process was highly error prone.

I produced a tool that would take sum-of-products function specifications and produce a CMOS complex gate implementation. The tool produces a COSMOS ntk format file, or a schematic in Electric which can be printed.

Following the success of the complex gate tool, I approached Al Davis with the concept of writing a burst-mode synthesis tool as there were no tools available at the time which fitted our needs. Davis, Coates and myself then embarked on the MEAT tool to automate the synthesis of low latency AFSMs.

This prototype tool greatly decreased design time and opened up other design issues which had previously been hidden by the complexity of hand AFSM synthesis. David Dill had just completed his dissertation at this time and we were fortunate enough to get one of his students, Steve Nowick, to modify his trace structure verifier to model burst-mode AFSM verification. We could now synthesize and verify AFSM modules! Nowick also discovered a problem with our synthesis approach which could generate hazards under certain circumstances. This discovery and the algorithm to avoid the hazard were key contributions to his thesis.

4.2 CCS Burst-mode Specifications

Easily specifying and exploiting parallel operations is highly desirable for asynchronous state machine controllers because concurrency is "free" in hardware if the components must exist for behavioral reasons. There is no cost other than complexity of control and increased power consumption for operating transistors in parallel. Invoking operations in parallel and synchronizing multiple process completions should be as natural as sequencing for these systems. The distinguishing feature of burst-mode is its ability to control parallel activity while constraining it to ease implementation complexities and testability.

Rule 1 *Input bursts and output bursts may not overlap*

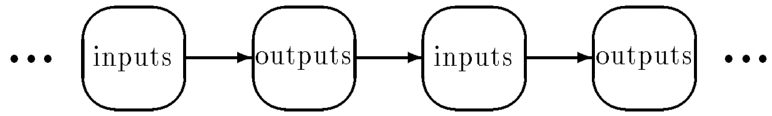


Figure 4.1: Burst-mode Conceptual Model

The most salient feature of burst-mode specifications is the constraint that inputs and outputs are separated into distinct stages of parallel activity as shown in Figure 4.1. When this cannot be accomplished, the specification must be decomposed into a multiplicity of communicating AFSMs. The standard syntax for CCS and the Concurrency Workbench does not allow a convenient burst-mode syntax. All orderings of signal interleavings must be expressed explicitly. For large bursts, this can be very tedious and error prone. The following definitions describe an extended notational convenience that will be used for burst-mode behavior. It is used throughout the remainder of the text and by the software analysis and verification tools developed as part of this thesis.

Definition 1 *For $\alpha_1, \cdots, \alpha_n \in \mathcal{A}, n \geq 1$, where α_i are all distinct, the* ***input burst*** *$(\alpha_1, \cdots, \alpha_n).P$ is a set of events defined recursively as follows:*

$$
\begin{aligned}
().P &\stackrel{\text{def}}{=} ERROR \\
(\alpha_1).P &\stackrel{\text{def}}{=} \alpha_1.P \\
(\alpha_1, \cdots, \alpha_n).P &\stackrel{\text{def}}{=} \textstyle\sum_{1 \leq i \leq n} \alpha_i.(\alpha_1, \cdots, \alpha_{i-1}, \alpha_{i+1}, \cdots, \alpha_n).P \qquad (n > 1)
\end{aligned}
$$

Definition 2 *For $\overline{\alpha_1}, \cdots, \overline{\alpha_n} \in \overline{\mathcal{A}}, n \geq 0$, where α_i are all distinct, the* ***output burst*** *$(\overline{\alpha_1}, \cdots, \overline{\alpha_n}).P$ is $\subseteq$ the set of events defined recursively as follows:*

$$
\begin{aligned}
().P &\stackrel{\text{def}}{=} P \\
(\overline{\alpha_1}, \cdots, \overline{\alpha_n}).P &\stackrel{\text{def}}{=} \textstyle\sum_{1 \leq i \leq n} \overline{\alpha_i}.(\overline{\alpha_1}, \cdots, \overline{\alpha_{i-1}}, \overline{\alpha_{i+1}}, \cdots, \overline{\alpha_n}).P \qquad (n > 0)
\end{aligned}
$$

A notational extension is applied to CCS in this thesis (by Definition 10) where the set of names $\mathcal{A}$ is defined as inputs and the set of conames $\overline{\mathcal{A}}$ is defined as actively driven outputs. Input bursts will only contain input signals, and output bursts will only contain output signals. The input burst is further restricted to be a nonempty set of transitions. A second important concept of burst-mode is that the order and time of arrival of events in a burst are unconstrained.

The C-element or rendezvous is a good example of a simple multiple input change burst-mode state machine. The two inputs, a and b arrive in a burst. The order and time of arrival of these two signals is unconstrained. After the inputs have arrived, the output $\overline{c}$ will be driven, and the circuit will then accept another input burst. The extended burst-mode notation described in Definitions 1 and 2 is compared against CCS syntax in Table 4.1 for the simple C-element.

C-element Specifications			
Burst-mode:	C-elt	$\stackrel{\text{def}}{=}$	$(a, b).\overline{c}.$C-elt
"Standard" CCS:	C-elt	$\stackrel{\text{def}}{=}$	$a.b.\overline{c}.$C-elt $+$ $b.a.\overline{c}.$C-elt
Barrier synchronization:	C-elt	$\stackrel{\text{def}}{=}$	$(\mathrm{A}\vert\mathrm{B}\vert\mathrm{S})\backslash\{p, g\}$
	A	$\stackrel{\text{def}}{=}$	$g.a.\overline{p}.$A
	B	$\stackrel{\text{def}}{=}$	$g.b.\overline{p}.$B
	S	$\stackrel{\text{def}}{=}$	$\overline{g}.\overline{g}.p.p.$C
	C	$\stackrel{\text{def}}{=}$	$\overline{c}.$S

Table 4.1: Different Burst Specification Styles

Standard CCS syntax requires that all signal interleavings be explicitly stated, including all the parallel choice space. This results in a specification with $n!$ signal traces, where n is the number of signals in the burst. Bursts quickly become extremely difficult to specify correctly and hard understand in the pure CCS notation.

A general solution using standard CCS notation and "barrier synchronization" doesn't require enumerating all of the signal interleavings but requires $n+2$ agents. Each of the signals in the burst are placed in a separate agent and the signal is bounded by synchronization signals which enable their transition, g, and signal the transition has fired, $\overline{p}$. This is clearer than enumerating the interleavings, but such descriptions are difficult for systems to evaluate compositionally due to the local nature of the parallel compositions of interdependent processes it requires. Barrier synchronization also results in an unsatisfactory circuit definition because it relies on CCS handshake synchronization that results in computation interference (as with signal g in Table 4.1) or multiple outputs driving the same signal (as with signal $\overline{p}$). See Section 7.2 more details on correct circuit constructions in CCS.

Lesson 4 *Segregating inputs and outputs allows clear, concise control of parallelism from a sequential agent.*

Lesson 5 *The behaviors of most native elements used for asynchronous design such as the C-element, MERGE, TOGGLE, etc. segregate inputs and outputs.*

4.3 Fundamental Mode Requirement

Multiple output change circuits lose the verification simplicity and some of the robustness of delay-insensitive and speed-independent circuits. Figure 4.2 is an example of such a circuit.

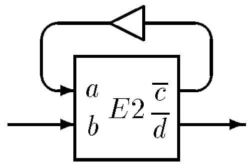


Figure 4.2: Burst-mode AFSM with Output Burst

Box $E2$ of Figure 4.2 has two inputs a and b, and two outputs $\overline{c}$ and $\overline{d}$. Assume that the non-inverting buffer has the obvious behavior where a transition on the input will be followed by a transition on the output. Also, assume that $E2$ has the following behavior with an output burst:

$$E2 \stackrel{\text{def}}{=} b.(\overline{c}, \overline{d}).a.\overline{d}.E2$$

The environment should provide a transition on signal b, at which point the input burst is complete. The output burst will then be enabled to fire. Assuming an unbounded gate delay model, $E2$ can produce a transition on signal $\overline{c}$ at which point the buffer can then produce a transition on signal a. This results in *computation interference* because $E2$ has not completed its output burst and is not in a receptive state for the a transition.

Thus MOC burst-mode circuits are not in themselves either speed-independent or delay-insensitive modules. However, if the stability function of fundamental mode holds, as is typically the case, then all outputs ($\overline{c}$ and $\overline{d}$) in the burst will fire before the next input arrives. Burst-mode assumes fundamental mode as an engineering abstraction that is relatively easy to uphold and is more consistent with hazard definitions.

AFSM $E2$ may be placed in a different environment from Figure 4.2. If a transition on signal a will not be generated until after both signals $\overline{c}$ and $\overline{d}$ have been driven and accepted by the environment there will be no computation interference. Therefore it is possible to place burst-mode machines in an environment where they can operate in a delay-insensitive or speed-independent fashion.

The window of vulnerability to computation interference is very small in Huffman machines. For example, in the Post Office design, none of the state machines required extra delay to ensure that fundamental mode delay assumption hold. Some methods, such as the 3-D method [YD92], have been developed which reduce the window to where it is practically nonexistent if the complexity of output generation is similar for all signals.

4.4 Burst-mode Specifications

Burst-mode specifications in the Post Office design used a graphical representation which was a variant of Mealy state graphs. This format has the advantage that it is familiar to hardware designers and is a simple way to encapsulate concurrency, communication, and synchronization. Further, these specifications can easily be mapped to a textual format for synthesis tool input (such as MEAT or Stetson).

Although there are explicit rules for the correct construction of a burst-mode specification, the above mentioned tools do not enforce or check many of the constraints. Ensuring that the behavior, properties, and most burst-mode specification rules are correct is left as an exercise for the designer. This can result in physical implementations that will not operate predictably.

The remaining sections of this chapter will discuss the burst-mode requirements, and formalize them so that burst-mode AFSM specifications can be proven correct in a larger synthesis system. The mechanism for this is developed later in this thesis. The correct specifications can then be passed to an AFSM synthesis and layout system.

4.5 Burst-mode Implementation Rules

Burst-mode transitions can be defined in terms of flow table specifications. A flow table is a two dimensional array structure which captures the internal and external states of a circuit [Ung69]. The rows of the table correspond to the internal state of the circuit, and the columns to the state of the inputs. Table entries are ordered pairs containing the next state and current output information. When the next state in an entry corresponds to the current state, the flow table is in a **stable** state; otherwise the current state is unstable and an internal *state transition* will occur. A simple way of understanding the flow table is to note that horizontal movement within a row represents changes in the values of input signals, while vertical movement within a column represents a state transition.

Definition 3 *Let* $P = \{0, 1\}$. *Each state machine contains an input set* $\mathcal{I}$ *of* I_m *variables where the value of* $I_m \in P$. *Each state machine also contains a set of output signals* $\mathcal{O}$ *of* O_n *variables where the value of* $O_n \in P$.

Definition 4 *An* input burst IB *for the transition* $\xrightarrow[OB]{IB}$ *consists of the nonempty set of input signals* $\forall\ I \in \mathcal{I}$ *which change value during the transition.*

Definition 5 *An* output burst OB *for the transition* $\xrightarrow[OB]{IB}$ *consists of the set of output signals* $\forall\ O \in \mathcal{O}$ *which change value during the transition.*

Definition 6 *A burst-mode state machine BSM,* $(S, S_0, \mathcal{I}, I_0, \mathcal{O}, O_0, \{\xrightarrow[OB]{IB} : IB \subseteq \mathcal{I}\ \wedge OB \subseteq \mathcal{O}\})$, *consists of*

- *a set* S *of* states, *where* S_0 *is the initial state.*
- *a set* $\mathcal{I}$ *of* inputs, *where* I_0 *is the initial values of the inputs.*
- *a set* $\mathcal{O}$ *of* outputs, *where* O_0 *is the initial value of the outputs.*
- *a* transition relation $\xrightarrow[OB]{IB} \subseteq S \times S$ *potentially for each value of IB and OB where IB is the* input burst *and OB is the* output burst.

Rule 2 *Each input burst must contain at least one signal transition. An output burst may be empty.*

The signals in the input burst may not be empty as at least one input must change for a transition to occur. The values of the inputs and outputs are significant, as these values are mapped to voltages in an implementation. The typical method for representing transitions in a burst-mode description include up or down arrows for the value transition of an input or output. For example, signal a changing from high to low is represented by $\downarrow a$. This can be textually represented as `a~` (and $\uparrow a$ represented as `a`).

Definition 7 *The* entry point *of a transition corresponds to the location in a flow table with the initial row (state) and input (column) values before any inputs in the transition have occured.*

Definition 8 *The* exit point *of a transition corresponds to the location in a flow table within the initial row (state) of the transition. The column value consists of the final state of the transition where all inputs in the burst $I \in IB$ have their new values.*

Rule 3 *Given the set of burst-mode transitions $T = \{\xrightarrow[OB]{IB} : IB \subseteq \mathcal{I} \ \wedge OB \subseteq \mathcal{O}\}$, $\forall\ T_i \in T$ the exit point of T_i must have the equivalent column location (input values) of an entry point T_j. Further, the destination state of T_i must be equivalent to the starting state of T_j and $T_i \neq T_j$.*

Rule 4 *The entry point E_i must be stable. Further, all flow table states in the cube covering entry point to but not necessarily including the exit point of the transition must be stable and will contain the same output values and state markings as the entry point.*

Rule 5 *The exit point will contain the destination state marking. The marking of the exit point will contain either the same output marking as the entry point or the marking of the new output values following the completed output burst. If the implementation uses a single transition time implementation technique, then all signals in the input burst can be don't cares in the exit point.*

Burst-mode is defined using closed flow tables. State transitions are defined with input and output signal transitions that are segregated into independent bursts of activity. Requirements for filling in flow table values are described which eliminate function hazards, and do not allow the state change or output burst to proceed until the input burst is complete. The output burst can change concurrently with the state change or may be delayed until the state change is complete. The choice is left to the designer, and results in a slight tradeoff between performance, area, and possibly the ability to remove hazards.

Each input burst results in a particular path through the flow table and AFSM state space, starting at the stable entry where the burst begins. At least one input change is required to generate a transition as there is no clock. The circuit remains in stable states, and state changes in the flow table can only move horizontally until all inputs in the input burst have been accepted. At this point a state change and output generation may occur.

The exit point of a transition in a *primitive* flow table always moves to a new row of the flow table forcing a state change. Minimization of the flow tables can result in merging of rows (or states in the AFSM), and may result in transitions which remain in the same state (or row of the flow table). These rules apply to minimized as well as primitive flow tables and may further restrict the way a state machine is minimized.

Theorem 1 *Function hazards in combinational logic are not possible in burst-mode design.*

Proof A function hazard can only exist for a transition $A \rightarrow B$ iff there exists a minimum length path between A and B where the output function value changes more than once. According to Rule 4 and 5, all outputs and state variables can only change following a completed input burst, or in state B for outputs. Hence there is no possible minimal path that can contain multiple function changes. $\square$

Exact, minimal, hazard free sum-of-products circuit implementations can be found for incompletely specified boolean functions generated from burst-mode specifications. Function hazards will not exist in the combinational output or state logic produced by MIC burst-mode specifications. All variants of static and dynamic hazards can be removed from the combinational logic necessary to build burst-mode state machines.

Rule 6 *The* ***burst-mode stability*** *requirement does not allow a new input burst to arrive until the output burst has completed and all circuit elements have stabilized.*

As shown in Section 4.3, this stability requirement is necessary for MOC circuits to avoid hazards and computation interference.

4.6 Burst-mode Specification Rules

This section describes the rules for correct construction of burst-mode state machines. A state graph interface was used in the Post Office, but the specifications were not checked for correct usage and construction. The formalizations that follow allow this checking to be automated with CCS specifications.

Rule 7 *All inputs and outputs must strictly alternate between rising and falling transitions for any valid path of input bursts in the state machine.*

The necessity to unambiguously mark transitions from the state of signals in the input set causes transitioning inputs and outputs to change an even number of times when there are loops in the state graph. Transition levels and voltage initialization values are necessary for burst-mode because it describes hardware implementations. Signal a becoming asserted from a low voltage to a high voltage is represented with $\uparrow a$ in graphical specifications used in the Post Office. Some mapping from a more abstract model, such as CCS or one based solely on transitions, must be carried out before a circuit is implemented. In general, the transformations required can double the size of the specification (such as for the C-element description of Equation 3.1), but it is generally an easy transformation.

Rule 8 $\forall\, T_i, T_j \in T$ *if the entry point of transition* T_i *is equivalent to the entry point of transition* T_j *then* $IB_i \not\subseteq IB_j$ *and* $IB_j \not\subseteq IB_i$

No transition burst is a sub-transition of others from same starting state, and unspecified signal transitions are illegal.

Rule 9 *$\forall\ T_i, T_j \in T$, if the entry points of the two transitions are equivalent then there must be at least one pair of inputs $i \in IB_i$ and $j \in IB_j$ such that $i \neq j$ and the environment will not provide both i and j. Otherwise, the state machine BSM must operate in single input change mode.*

When multiple edges exit a single state, there must be at least one pair of mutually exclusive signals for all pair of edges exiting the state [Mil65]. If there is no pair of mutually exclusive signals for all pair of edges then the state machine can only operate in single input change mode. This constrains the behavior of the environment.

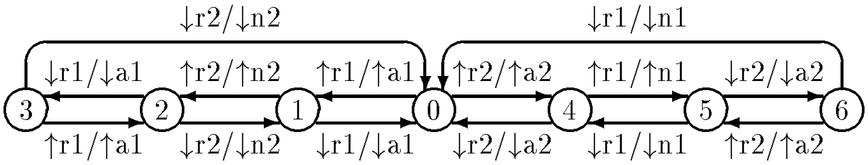


Figure 4.3: Nacking Arbiter SIC State Machine Specification.

The state machine for the nonblocking (nacking) arbiter of Figure 4.3 used in the Post Office is an example of a state machine that must operate in SIC mode because the environment permits both *r1* and *r2* to transition concurrently. These signals are passed through a SEQUENCER which converts MIC signals into SIC signals.

Nondeterministic behavior is not possible in a burst-mode state machine. However, nondeterministic behavior can be achieved when mutual exclusion elements (**ME**s) are used to condition inputs to a state machine, such as by using the SEQUENCER to condition the *r1* and *r2* signals to the nonblocking arbiter. MEs are analog devices, and are the only "library" device that may be required to implement burst-mode control functions. They are easily fabricated in most VLSI technologies, requiring 12 transistors in CMOS.

4.7 Post Office Design Process Example

The implementation of control circuitry consisted of the following steps once the MEAT toolset was in place. The behavior and algorithms of the Post Office had been simulated at the register transfer level. The control behavior was converted into burst-mode state machine specifications. Each state machine specification was fed into the MEAT tool. Individual AFSM implementations were verified with a version of Dill's verifier which had been converted to the burst-mode model. If hazards were found, they were removed if possible via complex gates and inverter restructuring, and the circuit was reverified. The final verified hazard-free burst-mode sum-of-products form was fed into the complex gate tool which would generate schematics. Each state machine was then laid out by hand from the schematics and simulated with COSMOS.

Datapath circuitry, such as latches, shift registers, and ALUs were designed and laid out in a similar fashion to synchronous components. They were simulated with SPICE and COSMOS. These were then composed with the controlling state machines, and the large blocks were simulated with COSMOS.

The datapath and burst-mode AFSM blocks were interconnected to form larger asynchronous modules. No verification was possible at this point with our toolset for two reasons. There was no intermediate form that could compare the top-down register transfer level design simulation with the bottom-up physical implementation. Further, even systems of state machines could not be verified with Dill's verifier. This was due to both the bottom-up design style and differences in the burst-mode model and his verification tool.

The stock version of COSMOS could not be used to simulate the entire Post Office chip. The event queue was designed such that new events could not be injected between *cycles*. A COSMOS cycle is not complete until the circuit has stabilized – there are no more pending events in the event queue. This behavior models a clocked system, where the circuit must stabilize between each clock phase. Asynchronous event injection between COSMOS cycles was necessary in certain situations in the Post Office. For example, state machines continuously attempt to forward centrally buffered packets out available ports. If the destination ports are busy the packets cannot be forwarded; the state machines loop continuously attempting to forward the packets. I modified COSMOS to allow asynchronous events, at which point it could be used to simulate the entire Post Office chip, including the pads. This was a critical aspect of the design process because it was the only method available for validating the implementation. Fortunately COSMOS was efficient enough to permit the pad to pad simulation of the entire circuit.

4.7.1 Asynchronous State Machine Design Example

The state machine from the Post Office chip called *SBuf-Send-Ctl* will be used as a design example. This state machine initiates the forwarding of a packet that has been placed in the central buffer pool out an idle port. This is one of the burst-mode examples I made publicly available which have been used as synthesis benchmarks by tool designers [Chu93]. Reported implementations of this state machine generated from other toolsets can be found in [LKSV90, ND91b]. When I designed this circuit, the first step was to create the burst-mode specification of the state machine graphically, which can be seen in Figure 4.4.

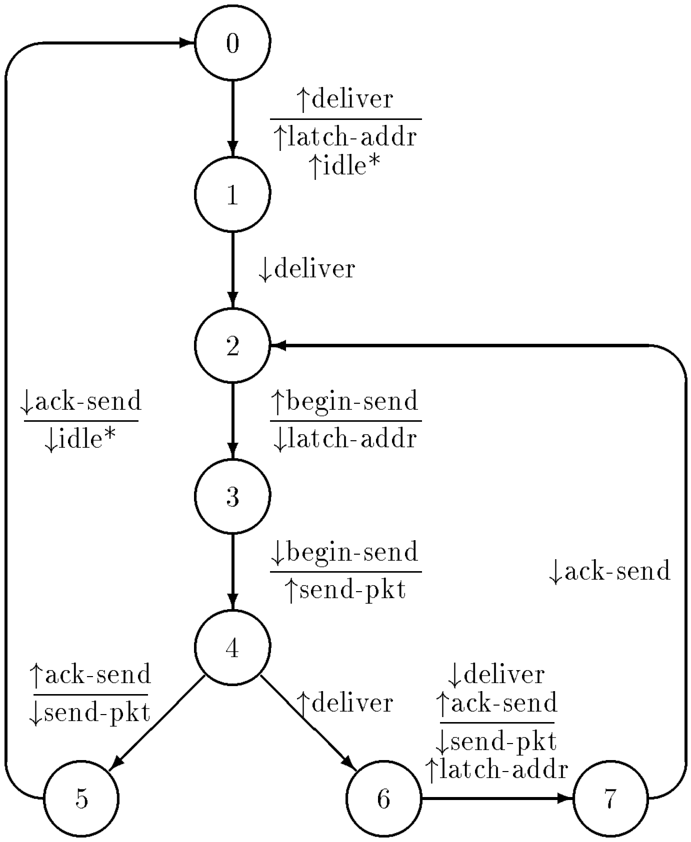


Figure 4.4: SBuf-Send-Ctl Burst-mode Specification

The graphical representation of the specification was then converted into a textual description suitable for input to MEAT. The :fsm directive names the state machine, and the :in and :out directives declare the names of the input and output signals of the state machine. The remainder of the text describes the behavioral specification. Transitions are specified as a four-tuple following the :state directive. The current state appears first, followed by the input burst in parenthesis. The next state is entered followed by the output burst in parenthesis. The conjunction of signal transitions in a burst is represented with the '*' symbol, while disjunctive choice is represented by the '+' symbol. Active high transitions on signal a (shown as $\uparrow a$ in Figure 4.4) are textually entered as `a`, whereas low transitions on a, ($\downarrow a$) are textually entered as `a~`. The textual conversion is as follows:

```
:fsm SBuf-Send-Ctl
:in (Rej-Pkt Begin-Send Ack-Send)
:out (Latch-Addr IdleBAR Req-Send)
:state 0 (Rej-Pkt)
        1 (IdleBAR * Latch-Addr)
:state 1 (Rej-Pkt~)
        2 ()
:state 2 (Begin-Send)
        3 (Latch-Addr~)
:state 3 (Begin-Send~)
        4 (Req-Send)
:state 4 (Ack-Send)
        5 (Req-Send~)
:state 5 (Ack-Send~)
        0 (IdleBAR~)
:state 4 (Rej-Pkt)
        6 ()
:state 6 (Rej-Pkt~ * Ack-Send)
        7 (Req-Send~ * Latch-Addr)
:state 7 (Ack-Send~)
        2 ()
```

MEAT is executed to process the specification contained in the file. The sum-of-products sequential logic required to produce the outputs is generated, along with necessary state variables used for feedback. MEAT produced an implementation for SBuf-Send-Ctl that required two state variables, *Y0* and *Y1*. An edited transcript of the MEAT session follows. (The user was required to enter the maximal compatibles in the version of MEAT used to design the Post Office. See [Ste92, CDS93a] for more details on MEAT.)

```
> (meat "sbuf-send-ctl.data")


Max Compatibles: ((0 5) (1 2 7) (3 4) (6))
> Enter State set: '((0 5) (1 2 7) (3 4) (6))


SOP for "Y1":
  18: REJ-PKT + Y1*BEGIN-SEND~
SOP for "Y0":
  28: BEGIN-SEND + Y0*ACK-SEND~ + Y0*REJ-PKT
SOP for LATCH-ADDR:
  12: Y1*Y0
SOP for IDLEBAR:
  30: BEGIN-SEND + Y0 + Y1
SOP for REQ-SEND:
  12: Y0*BEGIN-SEND~
HEURISTIC TOTAL FOR THIS ASSIGNMENT: 100
```

The combinational logic generated from the above transcript is free of hazards. However, hazard free implementations cannot be guaranteed with sequential logic. Hazards due to the feedbacks in sequential logic were removed whenever possible in the Post Office design. The Post Office design style localizes unremovable hazards internally to the AFSMs. The unremovable hazards can be verified and analyzed using the physical properties and variations of the devices and layout, rather than an asynchronous analysis using unbounded delays. This can result in faster, smaller circuits with functionally correct asynchronous interfaces.

The MEAT generated circuits in the Post Office, including the one in this design example, were all verified to determine if hazards existed in the implementation, and if they could be removed by design tricks. Following is the transcript of the verification of SBuf-Send-Ctl using Dill's verifier ported by Nowick for burst-mode AFSMs. The verifier reads the specification and then calls MEAT to generate the implementation using the `verifier-read-fsm` command. The definition of the specification is placed in the global variable `*spec*`, and the implementation in `*impl*`.

Dill's verifier assumes that each combinational function, including signal inversion, utilizes distinct devices. Hence, in the example, a separate inverter is created for the $\overline{\textit{begin-send}}$ signal to the Y1 and Req-Send logic. This multiplicity of physical instances of the "same" signal nearly always results in hazards in burst-mode speed-independent analysis. Merging all inverters with the same source signal together, and fanning the output of the single device to the destination logic blocks typically removes these hazards and creates a smaller faster circuit. In this example, the two Begin-Send inverters are merged, and their output fanned out to both logic blocks. The merge and fanout operations are executed in the verifier by issuing the `merge-gates` function below. The `verify-module` function is then called to analyze the circuit for hazards:

```
> (verifier-read-fsm "sbuf-send-ctl.data")


Max Compatibles: ((0 5) (1 2 7) (3 4) (6))
> Enter State set: '((0 5) (1 2 7) (3 4) (6))


> (setq *impl* (merge-gates '(1 11) *impl*))
> (verify-module *impl* *spec*)
10 20 30 40 50
Error: Implementation produces illegal output.
```

The verifier points out an implementation error, a transient hazard [Ung69]. Two transformations were used in the Post Office project to remove hazards. The removal of this hazard with signal reordering is described in Section 3.7.1 by the addition of an inverter using the verifier's `connect-inverter` function. The circuit was then verified free of hazards as shown with the following transcript.

```
> (setq *impl* (connect-inverter 1 7 *impl*))
> (verify-module *impl* *spec*)
10 20 30 40 50 60 70 79 states.
T
```

The implementation has now been verified as hazard free. The next step was to lay out the circuit in an efficient manner. All Post Office state machines used

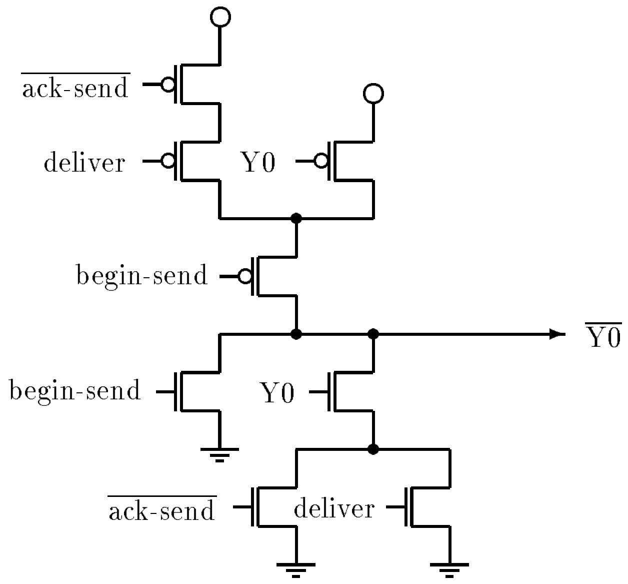


Figure 4.5: Complex Gate Schematic for SBuf-Send-Ctl Y0

complex gates to reduce the area and possibly increase performance of the circuit. The complex gate tool evaluates the equations for each state variable and output, and a schematic for each complex gate is generated. Figure 4.5 shows the complex gate generated by this tool for the Y0 state variable logic. The circuit was then laid out using in Electric.

The final layout of each cell, subsystem, and the entire chip were checked by simulation. The SBuf-Send-Ctl layout was extracted from Electric as a COSMOS 'ntk' file. COSMOS test vectors were hand-generated from the burst-mode specification and the layout was simulated in COSMOS. This state machine was then interconnected as part of a larger Post Office subsystem and simulated by COSMOS, and ultimately as the complete chip.

Since no behavioral model or interface description existed for blocks larger than single state machines, the simulation vectors were tested and developed concurrently with the circuit design and layout. The vectors did a poor job of fault covering and behavioral testing of the larger function blocks. In retrospect, a better effort in this area could have helped with the design modeling and testing of the implementation.

The layout of SBuf-Send-Ctl used in the Post Office chip is shown in Figure 4.6.

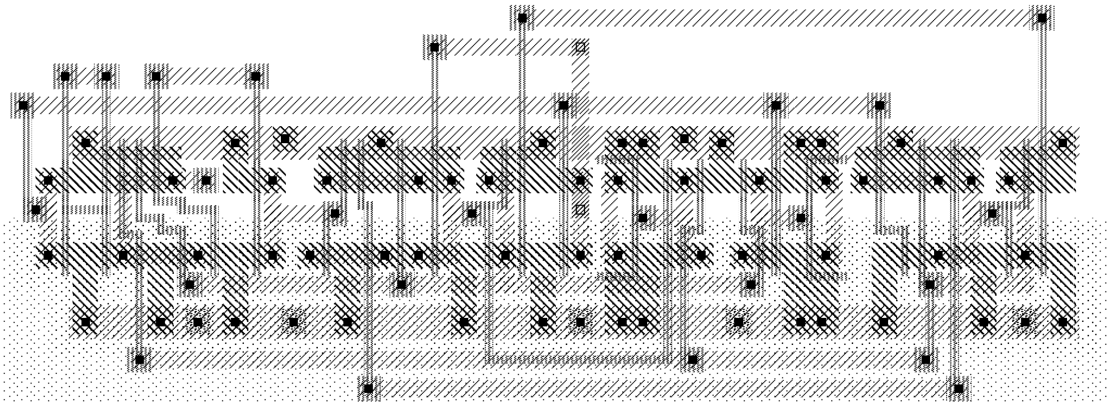


Figure 4.6: Layout of SBuf-Send-Ctl

4.8 Summary

Burst-mode is a multiple input and output change AFSM constraint system developed as part of the Post Office project. The primary advantages of this system is the guarantee of implementations free of function hazards, a formalism for MOC AFSM verifications, and the ability to synthesize compact circuits such that hazards can be localized. Hence macro module components or entire systems can be designed as networks of burst-mode state machines, and the architectural and design techniques are not restricted. However, the layout of the individual burst-mode state machines must be controlled if all hazards cannot be removed. Burst-mode is a significant contribution to the asynchronous design community. A number of different burst-mode design styles have emerged since the MEAT tool including a locally clocked system [ND91b], 3-D system [YD92], and an STG based system [Chu93].

In any sequential circuit, stability cannot be forced between input changes and state changes, so sequential delay hazards may exist, although burst-mode can reduce the occurrence of such hazards. Verification can be used to point out where unremovable hazards exist, and where timing inequalities must hold. None of the 95 different burst-mode state machines in the Post Office required any additional delays or logic to assure the timing inequality would hold under worst-case analysis. However, in a less constrained layout environment such as programmable logic devices, where the locality of AFSMs may be difficult to enforce, the inequality may not hold and additional delays may be necessary.

The MEAT synthesis software and a set of Post Office burst-mode AFSMs have been available to the research community via anonymous ftp since 1989.

Chapter 5

Hardware Equivalences Formalized in CCS

Defining and calculating equality between agents is fundamental to applying formal methods to circuit verification, yet formalizing practical equivalences between asynchronous agents is a formidable challenge. Recent developments have resulted in a number of theories and languages that can be used to specify and then calculate equivalences between a component and its specification.

A circuit is usually viewed as a "black box" (or package) and its specification only describes the necessary observable behaviors. Any design that conforms to this specification could be inserted into the package and function correctly. Making the specifications as "loose" as possible without compromising the design requirements allows designers more freedom of implementation. Requiring that a component and its specification have equivalent behaviors is usually too tight a requirement and one that does not concur with the black box philosophy. Further, it almost always results in slower, more complex, and more expensive circuits.

Three techniques have been used by researchers to prove that different circuit implementations can match a loose specification. One method composes the mirror image (inverse) of the specification with the implementation and then checks for equivalence and illegal communication behavior between the specification and implementation. The second approach uses preorders rather than equivalence testing. Preorders permit the implementation to have more behaviors than the specification, and ensures that the required behaviors are present. The final method uses modal logic equations as behavioral tests.

Probably the first generally useful tool for asynchronous circuit verification was developed by Dill [Dil89]. Here verification is achieved with trace theoretical principles coupled with specification mirroring for implementation flexibility. Mirroring can result in errors if handshaking occurs between the inverse specification and the implementation when the implementation responds too quickly, and interference is not detectable. Trace theory, mirroring, and the complexity of the model Dill uses limit the usefulness of the verifier; for example, complete traces are not employed. Ebergen developed a verifier based on trace semantics for delay-insensitive implementations [EG93]. The delay-insensitive model, although extremely useful for protocols and high-level circuit verification, cannot verify gate level implementations nor many of the common hazard models. The general purpose Concurrency Workbench [Mol91] supports more powerful equivalence theories, but none of the equivalences or partial orders introduced before this thesis are directly applicable to verifying hardware components.

This section reviews the most important equalities used in CCS and by modern asynchronous verifiers and formal languages. CCS is introduced as a useful language for defining equivalences, representing, and reasoning about circuits. A set of partial orders, called **conformances**, are then described. The inadequacy of trace based semantics is demonstrated. A new conformance is introduced based on bisimulation semantics. The conformances presented here are used as the foundation for the prototype synthesis and verification tool discussed and built as part of this thesis.

5.1 Advantageous CCS Properties

CCS is a formalism for reasoning about complex parallel systems [Mil89]. The primary advantages of CCS over other formalisms are very significant. The following is a short summary of the aspects that are most useful for modeling parallel asynchronous hardware, and why the work in this thesis is based on CCS.

- **Simplicity.** CCS utilizes a sparse "object oriented" notation where interfaces and components can be described independently [SABL93]. The object oriented approach allows one to describe complex systems as a set of parallel agents. CCS contains only five constructions, and six distinct transition rules.

- **Unique Minimal Representation.** CCS has a unique, canonical minimal state representation for any behavior. This precision simplifies the application of CAD tools and transformation into other formats such as BDDs and burst-mode.

- **Hierarchical Representations.** Hierarchy or *structure* in a formalism must accomplish two conflicting requirements: (a) hide the complexity of the underlying behavior, and (b) retain all behaviors of the lower levels that directly affect the behavior at higher levels. Requirement (b) limits the amount of simplification that can be accomplished. Precisely modeling this tradeoff is key to CCS transition rules. Internal transitions are represented by the special symbol τ in CCS. These τ transitions have a unique set of rules that differ from other actions and support observational equivalences.

The ability to model hierarchy from first principles is a major strength of CCS over other formalisms. Many rivaling formalisms such as Petri nets treat structure and hierarchy vaguely. Others, such as trace systems, ignore too many behavior in the lower levels of the structure (as will be shown later in this chapter).

- **Equational Reasoning.** The sound semantics and combinators in CCS allows it to embrace most (if not all) of the common equalities that have been formally defined for finitely branching sequential processes. This allows one to explore the utility of different formalisms, such as trace and bisimulation theories. The axiomatization of the language supports the implementation of automatic support tools such as the Concurrency Workbench.

- **Formal Logics** A rich set of equational reasoning and logic systems exist for analyzing the properties of CCS agents. This includes Hennessey-Milner logic and the Modal-μ calculus [Sti91]. These can be applied directly to specifications to verify certain behavioral aspects before carrying out the implementation process. See [Liu92] for some applications of these logics to asynchronous systems.

CCS has found many direct applications. Perhaps the most successful have been verifications of complex protocols [Bre90, Bru92, Par87]. Since asynchronous circuits communicate via handshaking protocols, their correct interaction can be viewed as a form of protocol verification. However, the constraints of hardware implementation require some modifications to CCS as will be discussed in the following chapter.

5.2 Notational Definitions

This section defines the formalisms and terminology that are applicable to the labeled transition system used in this thesis.

Definition 9 *A labeled transition system,* $(S, T, \{\xrightarrow{t} : t \in T\})$*, consists of*

- *a set* S *of* states
- *a set* T *of* transition labels
- *a* transition relation $\xrightarrow{t} \subseteq S \times S$ *for each* $t \in T$.

Definition 10 *The labels (or actions) in labeled transition systems are defined as follows:*

- *Input action* ***name*** $a \in \mathcal{A}$ *(where the set of names* $\mathcal{A}$ *are inputs* $\mathcal{I}$*).*
- *Output action* ***coname*** $\overline{a} \in \overline{\mathcal{A}}$ *(where the set of conames* $\overline{\mathcal{A}}$ *are outputs* $\mathcal{O}$*). By convention,* $\overline{\overline{a}} = a$.
- *The set of* ***labels*** $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$.
- $\tau \notin \mathcal{L}$*, where the label* τ *(**tau**) is the invisible internal action.*
- *The* ***sort*** $\mathcal{L}(P)$ *of an agent* P *is its set of observable input and output actions.*
- *The actions of a system are:* $Act = \mathcal{L} \cup \{\tau\}$

The ability to specify a port as an input or output is essential when modeling hardware. Therefore, the labeled transition systems used here are extended to assign directionality to names and conames. The set of names $\mathcal{A}$ of a system consist of its inputs, while the set of conames $\overline{\mathcal{A}}$ contains the outputs of a system. The normal convention is followed by assuming that placing an overline over a label produces the label of its handshaking partner, even for outputs ($\overline{\overline{o}} = o$). The labels $\mathcal{L}$ of a system is the union of its inputs and outputs, which includes the observable actions (signal names) of the external ports of a hardware block or agent. Restricting the labels to the observable external (input and output) actions that the system can perform yields its *sort*. The set of actions Act the system can make consists of the sort together with the silent internal action τ.

Definition 11 *Agents sets (or hardware components) are defined as follows:*

- $\mathcal{P}$ *is the set of* ***agents*** $P, Q, \ldots$. *By convention, I refers to an implementation agent and S to a specification agent.*
- $\mathcal{E}$ *is the set of agent expressions* $E, F, \ldots$
- $\mathcal{P}$ *is derivation closed over the set of* $\mathcal{E}$

Definition 12 *Let P be an agent. If $P \stackrel{\alpha}{\rightarrow} P'$, then α is an* action *of P and P' is an* α-derivative *of P.*

Definition 13 $\mathcal{P}$ *is* ***derivation closed*** *if* $\forall\ P \in \mathcal{P}$ *and* $\forall\ \alpha \in Act$, *whenever* $P \stackrel{\alpha}{\rightarrow} P'$ *then* $P' \in \mathcal{P}$

This thesis uses agents and agent expressions as behavioral descriptions of hardware components. This restricts these expressions to finite systems where $\mathcal{P}$ is derivation closed over the set of $\mathcal{E}$. There is usually no loss of generality with this assumption, and it eases certain proof obligations. The α-derivative of an agent is always another agent.

When $\mathcal{P}$ is minimized, there is a single agent expression per state and the size of $\mathcal{E}$ is equivalent to the state size. The labeled transition system of Definition 9 takes T to be the actions Act and S to be the minimized agent expressions $\mathcal{E}$. The semantics for agent expressions includes the definition of each transition relation $\stackrel{\alpha}{\rightarrow}$ over $\mathcal{E}$.

This defines the standard CCS labeled transition system. The following definitions simplify reasoning about the observable actions an agent can make.

Definition 14 *If $s \in Act^*$ is an action sequence of an agent, then $\widehat{s}$ is defined to be the projection of s on $\mathcal{L}^*$, i.e. $\widehat{s}$ is the sequence obtained from s by deleting all occurrences of τ. If $s \notin \mathcal{L}^*$ then $\widehat{s} = \epsilon$.*

If a system can perform the sequence of actions s, then $\widehat{s}$ is the observable sequences (inputs and outputs) of that sequence. For example, if $s = in\,\tau\,\overline{out}$ then $\widehat{s} = in\,\overline{out}$. Both s and $\widehat{s}$ may be empty. It is convenient to define a new transition relation $\Rightarrow$ which allows the invisible τ transitions to be abstracted away.

Definition 15 *If $s \in Act^*$ then $\widehat{s} = \alpha_1 \ldots \alpha_n \in \mathcal{L}^*$ and $P \stackrel{\widehat{s}}{\Rightarrow} P'$*
iff $P(\stackrel{\tau}{\rightarrow})^* \stackrel{\alpha_1}{\rightarrow} (\stackrel{\tau}{\rightarrow})^* \ldots (\stackrel{\tau}{\rightarrow})^* \stackrel{\alpha_n}{\rightarrow} (\stackrel{\tau}{\rightarrow})^* P'$

Definition 16 *If $s = \alpha_1 \ldots \alpha_n \in \mathcal{L}^*$ then $P \stackrel{s}{\Rightarrow} P'$*
iff $P(\stackrel{\tau}{\rightarrow})^* \stackrel{\alpha_1}{\rightarrow} (\stackrel{\tau}{\rightarrow})^* \ldots (\stackrel{\tau}{\rightarrow})^* \stackrel{\alpha_n}{\rightarrow} (\stackrel{\tau}{\rightarrow})^* P'$. If $s = \epsilon$ then $\stackrel{\epsilon}{\Rightarrow} = (\stackrel{\tau}{\rightarrow})^*$
The shorthand $P \stackrel{s}{\Rightarrow}$ stands for $P \stackrel{s}{\Rightarrow} P'$ for some P'.

There is no direct control over τ actions in the $\Rightarrow$ transition as they have been filtered from consideration. If an action sequence s contains explicit τ actions they must be filtered out as is done in Definition 15, removing any contribution to the transition.

Note in particular that in Definition 16 the sequence s cannot contain any τ actions. However, any number of τ actions may occur in the transition before and after each action α_k. Because the internal τ transitions are ignored in this transition relation, the agent can utilize internal actions to choose different destination states. For example, assuming $\mathcal{E} = \{\mathrm{E3}_1, \mathrm{E3}_2\}$, $\mathrm{E3}_1 \stackrel{\mathrm{def}}{=} a.\mathrm{E3}_2$, and $\mathrm{E3}_2 \stackrel{\mathrm{def}}{=} \tau.\mathrm{E3}_1 + b.\mathrm{E3}_1$, then both $\mathrm{E3}_1 \stackrel{a}{\Rightarrow} \mathrm{E3}_1$ and $\mathrm{E3}_1 \stackrel{a}{\Rightarrow} \mathrm{E3}_2$ are valid transitions.

Definition 17 *If $s = \alpha_1 \ldots \alpha_n \in Act^*$, then P' is a **s-descendant** of P iff $P \stackrel{\hat{s}}{\Rightarrow} P'$.*

Definition 18 *The s-descendant of an agent P is a τ**-descendant** iff $s \in \tau^*$.*

The agent P and its τ-descendant P' can be the same agent. This occurs when $s \in Act^*$ and $s = \epsilon$.

Consider the labeled transition system

$$(\mathcal{E}, Act^*, \{\stackrel{\hat{s}}{\Rightarrow} : s \in Act^*\}) \tag{5.1}$$

based on sequences of visible actions rather than just single transitions. This results in a notion analogous to that of an α-derivative of an agent. This labeled transition system is a conceptual extension of a standard system based on s-descendants. It is an extremely busy system. Its main advantage is one of notational convenience in describing observable trace based systems.

5.3 Equivalences and Agent Properties

A central aspect of a formal verification system is the power of the equalities used. CCS can choose from a lattice of formalisms upon which one can base verifications, ranging from very strong (strong bisimulation) to very weak (trace equivalence). See [vG90b] for an excellent paper on the semantics of equalities. Figure 5.1 shows a lattice of some of the relative strengths of some practical equalities that may be used for circuit verification. Trace based systems are the weakest in use and bisimulation the most sensitive.

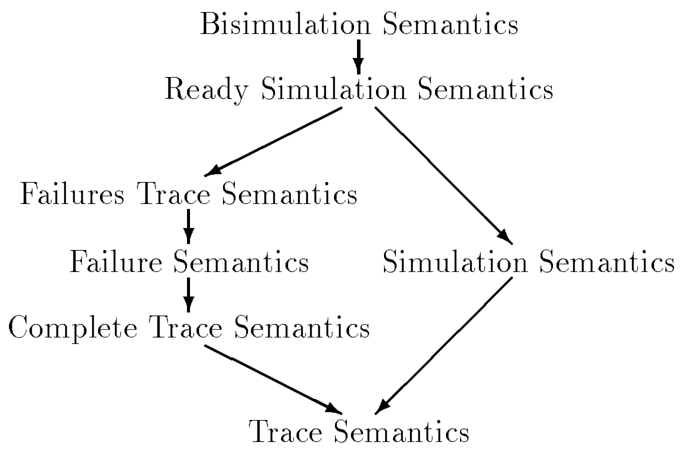


Figure 5.1: Lattice of Equality Relations

This section will cover the traditional trace systems and bisimulation based models and explain their appropriateness to hardware verification.

5.3.1 CCS Equalities

Trace Equivalence

Trace theory equates agents whose sequences of observable actions are the same. Complete traces must be used to assure accuracy under this model. This is the coarsest equivalence and is the least able to distinguish differences between agents. It equates more agents than other equivalences used in practice. The definition for trace equivalence follows.

Definition 19 *Agents P and Q are (weakly)* ***trace equivalent****, written $P =_t Q$, iff $\forall\ s \in \mathcal{L}^*\quad P \stackrel{s}{\Rightarrow}$ if and only if $Q \stackrel{s}{\Rightarrow}$*

As can be seen, this equivalence completely abstracts away the internal actions of agents. Information is not available about derivative actions nor concerning the effect of internal choices. Most trace systems can not even represent internal τ transitions. Perhaps the biggest problem with trace based systems is their inability to detect deadlock. Nonetheless, trace theory is used to verify asynchronous hardware [Dil89, Ebe88, Udd84]. Additional tools must be applied to verify other requirements. However, by adding the additional notational strength of bisimulation, one can get these capabilities in a single tool.

Bisimulation

Observational equivalence in CCS equates processes whose externally observable behavior is identical. The behavior of a system then becomes precisely what can be observed from the outside. Milner's conceptual idea of observational equivalence is similar to the "black box" specification where details of the implementation are unimportant if it conforms to the required external behavior. However, internal actions *can* modify the state of a module and cause a change the observable behavior of the box, so the definition of observation equivalence requires τ transitions.

The notion of (observational) equivalence was formalized by Park as bisimulation [Par81]. Agents P and Q are equivalent if, for every action α, every α-derivative of P is observation equivalent to some α-descendant of Q, and similarly with P and Q interchanged. Bisimulation gets it name from the back and forth nature of the definition. The following two varieties of bisimulation can be found in [Mil89].

Definition 20 *Agents P and Q are **strongly bisimilar**, written $P \sim Q$*
iff $\forall\ \alpha \in Act$

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

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

The notion of strong bisimilarity is not appropriate for our observational model as it requires internal actions of each agent to be matched by the other agent, even when the outcome is observably irrelevant. For example, $a.\tau.\overline{b}.\mathrm{Nil} \not\sim a.\overline{b}.\mathrm{Nil}$. Strong bisimilarity is, however, a foundation for many definitions, including our desired formulation of (weak) bisimulation.

Definition 21 *Agents P and Q are (weakly) **bisimilar**, written $P \approx Q$*
iff $\forall\ \alpha \in Act$

(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \stackrel{\widehat{\alpha}}{\Rightarrow} Q'$ and $P' \approx Q'$

(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \stackrel{\widehat{\alpha}}{\Rightarrow} P'$ and $P' \approx Q'$

In this thesis, bisimulation and equivalence between agents both refer to weak bisimulation (also known as weak equivalence). Weak bisimulation satisfies the notion of equating agents whose observable actions are indistinguishable. However, weak bisimulation is not a congruence, and so equivalent agents cannot be substituted safely. However, when the agents are initially *stable* (having no τ-derivatives) bisimulation is a congruence. See [Mil89] pages 111-113 for further details.

Branching time bisimulation is a second notion of bisimulation equivalence developed by van Glabbeek [vG90a]. This definition is slightly finer than Park's definition found in Milner. The agents $\alpha.(P + \tau.Q) + \alpha.Q = \alpha.(P + \tau.Q)$ cannot be distinguished by external observation, and hence are observationally equivalent. However, due to their significantly different derivation trees and internal branching structure, (the α-derivatives of the two sides are different), they are not considered equivalent in branching time bisimulation.

Branching time bisimulation contains some nice reasoning properties and simplifies some of the analysis algorithms. Hence the tool developed in this thesis is based upon van Glabbeek's branching time bisimulation. Branching time bisimulation is slightly finer than bisimulation because it can detect the difference of the above nondeterminate action. Note that the type of nondeterminism nondeterminism necessary to create a detectable difference between bisimulation and branching time bisimulation unlikely to occur with asynchronous control.

5.3.2 Predictability

The *predictability* of hardware, engineered components, and systems is of paramount importance. There must be a means for fabricating devices that will perform identically to previously built devices. Engineers also wish to test these devices for correct operation, or design self testing circuits. Many circuits require strong predictability – for the same input sequences the same output behavior is expected every time.

This stronger formulation of predictability has been formalized across agents as the **determinate** property, and when applied to hardware components is conceptually similar to declaring a component *deterministic.*

Definition 22 *P is* ***strongly determinate*** *if, for every derivative Q of P and $\forall\, \alpha \in Act$, whenever $Q \xrightarrow{\alpha} Q'$ and $Q \xrightarrow{\alpha} Q''$ then $Q' \sim Q''$ where $\sim$ is strongly bisimilar.*

This requirement states that the same experiment should always yield the same result. Consider the agent E4 $\stackrel{\mathrm{def}}{=} \alpha.\beta.P + \alpha.\gamma.Q$ (whose derivation tree can be seen as the right tree of Figure 5.5(a) on page 126). This agent expression is not strongly determinate because it has two α-derivatives to states which are not strongly equivalent. As with strong bisimulation, strong determinacy is not useful for engineered systems. For example, the agent expression E5 $\stackrel{\mathrm{def}}{=} \alpha.(\beta.P + \tau.\mathrm{Nil})$ *is* strongly determinate, but results in unpredictable behavior! After the α action, the agent E5, at its own choice, can decide to deadlock or accept a β action and evolve into agent P. (The derivation tree of E5 is similar to the left derivation tree of Figure 5.5(a) on Page 126 where γ is replaced by τ.)

By abstracting away from the internal τ transitions a preferred notation of determinacy can be defined.

Definition 23 *P is (weakly)* ***determinate*** *if, $\forall\, s \in \mathcal{L}^*$ whenever $P \stackrel{s}{\Rightarrow} P'$ and $P \stackrel{s}{\Rightarrow} P''$ then $P' \approx P''$*

Intuition tells us that an unpredictable system should not be determinate. This is true with (weakly) determinate systems. For example, the unpredictable agent expression E5 is not determinate. Hence this is the definition used in this thesis and any subsequent reference to determinate systems will refer to weak determinacy.

Proposition 1 *Determinacy is preserved by bisimulation; that is if P is determinate and $P \approx Q$ then Q is determinate.*

Proof Milner [Mil89] page 234. □

Unfortunately, determinacy is not preserved over the summation and composition operators of CCS. By restricting the syntax, determinacy can be preserved over composition and summation. However, the restrictions required to guarantee that determinacy is preserved by composition disallows communication between the parallel agents. This restriction would result in an unusable syntax for modeling parallel hardware. Thus we cannot be assured that a system built out of predictable determinate components (such as AND gates) will itself be predictable and determinate. This necessitates additional analysis of circuits designed from multiple parallel components when predictability is important.

There is a special type of determinacy, called **confluence**. The notion of confluence is one where if there are multiple possible actions, then the occurrence of one of the actions will not preclude the occurrence of the other actions. This is a notion that is similar to the semi-modular property developed by Muller [Mil65].

Similar to bisimulation and determinacy, there are strong and weak forms of confluence. For observational reasons, the work in this thesis is primarily interested in the weak form of confluence.

Definition 24 *P is (weakly)* ***confluent*** *if for every derivative Q of P the following diagrams can be completed such that if the top and left hand derivations exist then the bottom and right hand derivations can be inferred.*

$$
\begin{array}{ccc}
Q & \xrightarrow{\tau} & Q_1 \\
\Downarrow & & \Downarrow \\
Q_2 & \Rightarrow & \approx \\
 & (i) &
\end{array}
\qquad
\begin{array}{ccc}
Q & \xrightarrow{\tau} & Q_1 \\
\widehat{\alpha}\Downarrow & & \Downarrow\widehat{\alpha} \\
Q_2 & \Rightarrow & \approx \\
 & (ii) &
\end{array}
\qquad
\begin{array}{ccc}
Q & \xrightarrow{\alpha} & Q_1 \\
\widehat{\beta}\Downarrow & & \Downarrow\widehat{\beta} \\
Q_2 & \overset{\widehat{\alpha}}{\Rightarrow} & \approx \\
 & (iii)\ (\alpha \neq \beta) &
\end{array}
\qquad
\begin{array}{ccc}
Q & \xrightarrow{\alpha} & Q_1 \\
\widehat{\alpha}\Downarrow & & \Downarrow \\
Q_2 & \Rightarrow & \approx \\
 & (iv) &
\end{array}
$$

Confluent agents preserve determinacy over composition when all communicating actions are restricted. The set of confluent hardware components is extremely limited. However, confluence does have an application in burst-mode state machines as will be discussed in Chapters 4 and 7. For further details on determinacy and confluence, see [Mil89].

5.4 Hardware Conformance to Specifications

We now have enough notational strength to define when an implementation **conforms** to a specification. Conformances are not equivalences, but they determine when an implementation is an acceptable construction of the specification.

Part of the designer's art is to utilize the unspecified state space in such a way as to produce pleasing designs. Good specifications will maximize the unreachable state space without overly restricting the environment and implementation behaviors. This allows implementations to accept inputs which won't be provided by the circuit's environment, and to generate outputs from unreachable states. Such specifications are sometimes called "loose" specifications.

Conformance should be as "loose" as possible – equating as many agents to a specification as possible – without violating the requirements set forth by the specification. Conformance defines the appropriate restrictions applied between the specification and an implementation. The possible set of implementation action sequences can be restricted by the specification, which knows exactly when input and output actions are permissible.

The implementation must be capable of all behaviors dictated by the specification. Further, the implementation must not show any illegal behaviors within the reachable state space. In particular, any outputs the implementation may generate must be matched by the specification. Freedom of implementation is possible because the behavior in unreachable states is completely unrestricted.

Ignoring unreachable behaviors is axiomatized by the equational law shown in Proposition 2. When I_n is an implementation agent, it allows the agent expression $\alpha.I$ to be discarded because it is an unreachable input expression. This results in a preorder between specification and implementation because the valid behaviors of an implementation can be greater than those of the specification.

Proposition 2 $\alpha.I_1 + I_2 \succeq_l S$ *iff* $I_2 \succeq_l S$ *and* $\alpha \in \mathcal{A}$ *and* $S \stackrel{\alpha}{\not\Rightarrow}$

The specification defines the contract between an implementation and its environment. The agreement is twofold. First, the environment agrees to only provide input signals in the restricted ordering designated by the specification. Implementations are nearly always capable of accepting inputs which will not be provided by the environment. Since these agent derivations will not occur by definition, they can be ignored. This is important as it results in a "don't care" freedom for the designer of the implementation to exploit. Secondly, for valid inputs, the implementation will provide outputs precisely as specified.

5.5 Trace Conformance

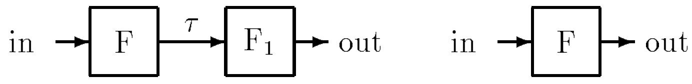
Definition 25(i) assures that under Trace Conformance, the implementation can match the behaviors of the specification. This requirement is identical to Trace Equivalence in Definition 19. Further, all states for which the agents are trace equivalent, the outputs must also match the specification exactly. Definition 25(ii) assures that for any sequence that I and S can perform, the output sequence will match exactly.

Definition 25 *Implementation I is **Trace Conformant** to specification S, written as $I \succeq_t S$, iff $\forall\ s \in \mathcal{L}^*$ and $\forall\ t \in \overline{\mathcal{A}}^*$*

(i) Whenever $S\overset{s}{\Rightarrow}$ then $I\overset{s}{\Rightarrow}$

(ii) Whenever $S\overset{s}{\Rightarrow}$ and $I\overset{st}{\Rightarrow}$ then $S\overset{st}{\Rightarrow}$

The simple FIFO example of Figure 5.2 will be used to demonstrate the intuition behind Trace Conformance. Assume that we want to see if a two place FIFO is a valid implementation of a one place FIFO. The specification of the single FIFO cell F in Figure 5.2 is defined as $\mathrm{F} \stackrel{\mathrm{def}}{=} in.\overline{out}.\mathrm{F}$. Assume also that the cell F_1 Figure 5.2(a) has the same behavior.



(a) Two Element FIFO/Hybrid (b) Single Element FIFO

Figure 5.2: Conformance Example with FIFO Buffers

Example 1

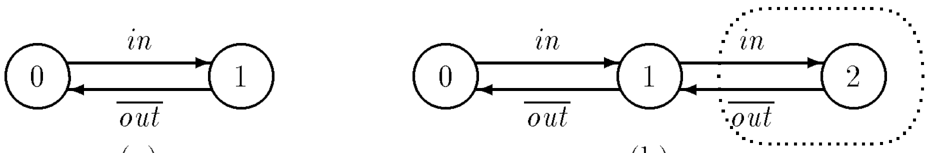
Trace conformance was defined as an operation on possible derivative sequences. If the traces of length three are examined for the single FIFO element F, and the two element FIFO implementation, (F[t1/out]|F[t1/in])\{t1}. The specification has a single valid trace — $in\ \overline{out}\ in$. If we look at the two element FIFO, it has one additional trace of length three, as shown in Table 5.1.

1:	$in\ in\ \overline{out}$
2:	$in\ \overline{out}\ in$

Table 5.1: Traces of Length Three for the Two Element Hybrid Circuit

Using the one place FIFO as the specification, the two place FIFO implementation can be checked for conformance. Definition 25(i) requires that the definition have the same behavior (traces) as the specification. The second trace in Table 5.1 is equal to the specification's trace, so that part holds. Next all traces that are unreachable can be discarded because of input restrictions. The two element FIFO can accept two inputs before producing an output as shown in the first trace of Table 5.1. However, the environment is restricted by the specification such that after presenting in, it must wait for $\overline{out}$ before it can supply another in signal. Since a second input will never be produced by the environment without first consuming an output, this trace can be discarded. This leaves the single trace which matches the specification, showing that for traces of length three or less, the implementation is trace conformant.

Figure 5.3 represents the same behavior as specified by the circuit diagrams of Figure 5.2 with state graphs (or the derivation trees of a labeled transition system). This representation has a more intuitive representation and simpler analysis methods than the traces of Table 5.1. The trace structure can be created from the state graph of Figure 5.3(b).



(a) FIFO Specification (b) FIFO Implementation

Figure 5.3: Two FIFO Derivation Trees

Trace Conformance verifies that the implementation has the same descendant behaviors as the specification, and that there are no illegal output behaviors. State 0 of the specification and implementation in Figure 5.3 both have the same derivative behaviors – they have *in*-derivatives that move to state 1. From state 1, the specification has an $\overline{out}$-derivative, which moves the specification back into state 0. The implementation also has an $\overline{out}$-derivative, which moves it back to the initial state. However, the implementation also has an *in*-derivative which the specification does not have. Since this is an input that will not be generated by the environment the transition can be ignored. State 2 is circled with a "cloud" indicating it is unreachable. This verifies that for all traces the two element FIFO is Trace Conformant to a single element FIFO.

This example can also be conceptually verified by examining the traces as regular expressions. The regular expression for the specification's traces is $(in\,\overline{out})^*$. The regular expression for traces of the two element FIFO is $(in(in\,\overline{out})?\overline{out})^*$[1]. The definition of trace conformance restricts the regular expression of the implementation resulting in the mapping of $(in(in\,\overline{out})?\overline{out})^* \longmapsto (in\,\overline{out})^*$. The internal term $(in\,\overline{out})^*$ is removed, which corresponds to the removal of state 2 in Figure 5.3. Since the traces for the specification and the restricted behavior of the implementation are equivalent, as $(in\,\overline{out})^*$, the implementation conforms to the specification.

Example 2

Suppose that the behavior of box F_1 in Figure 5.2 is redefined. Is this implementation Trace Conformant with the specification when $F_1 \stackrel{\text{def}}{=} in.(\overline{out}.\overline{out}.F_1 + \overline{out}.F_1)$?

1:	$in\ in\ \overline{out}$
2:	$in\ \overline{out}\ in$
3:	$in\ \overline{out}\ \overline{out}$

Table 5.2: Traces of Length Three for the Two Element FIFO

Table 5.2 shows the traces of length three for the new system. Verification can now be applied to the new system to determine whether or not it is trace conformant to the specification. The specification's trace is present as the second trace in Table 5.2, so Definition 25(i) holds. Next, for all reachable traces (or states), the implementation's output behavior must match that of the specification. Initially the specification can do no output; this behavior is matched by the implementation. Because the specification does not allow the environment to produce two adjacent *in* transitions,

[1]The '?' symbol indicates that the expression will be matched zero or one time.

the first trace in Table 5.2 is thrown out as unreachable. Let s be the trace in, which is a valid trace for both the specification and implementation. Then, let trace t be $\overline{out}\,\overline{out}$. The trace $\stackrel{st}{\Rightarrow}$ is possible for the implementation. However, since $\stackrel{st}{\Rightarrow}$ is impossible for for the specification, this implementation is not trace conformant with a single FIFO. This erroneous trace can be seen as the third trace in Table 5.2.

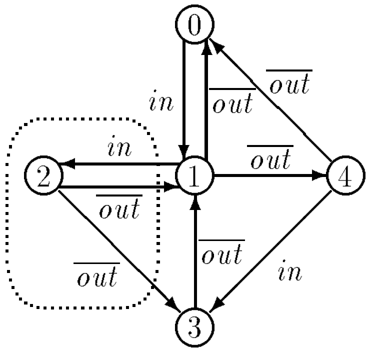


Figure 5.4: Derivation Tree of FIFO-like Structure

The derivation tree of the new FIFO structure is shown in Figure 5.4. Notice that the same required derivative structure of the specification in Figure 5.3(a) is present in the new FIFO structure's states 0 and 1. The in transition to state 2 and the "clouded" region can be thrown out because it is unreachable. Observe that there are two $\overline{out}$ transitions possible from state 1. Assume the $\overline{out}$ transition to state 4 is taken. From there, it is possible to take a second $\overline{out}$ transition to state 0, which is violates conformance. Thus this implementation is not trace conformant to the specification.

5.5.1 Suitability of Trace Conformance

Trace conformance is generally unsuitable for verification because of some undesirable equational laws arising from its inability to distinguish between agents. Proposition 3 contains some equational laws in trace systems that cause problems with circuit verification.

Proposition 3

(1) $\alpha.(P + Q) = \alpha.P + \alpha.Q$

(2) $P + \tau.Q = P + Q$

(3) $\tau.P = P$

(4) $(P + Q)|R = P|R + Q|R$

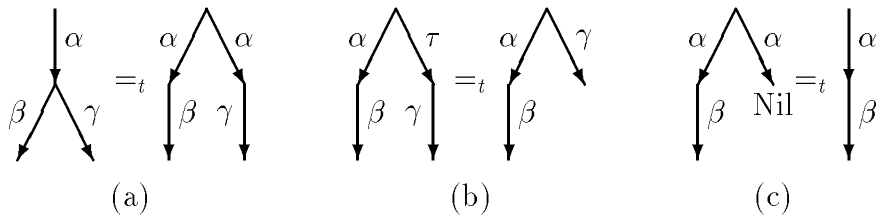


Figure 5.5: Weaknesses in Trace Analysis

These behaviors can result in equivalences between derivative trees which are undesirable for circuit verification. Trace conformance or trace equivalences cannot distinguish between the pairs of derivation graphs in Figure 5.5. Trace models are generally insensitive to the branching structure of agents. They cannot determine when choice is made by a system as shown in Figure 5.5(a) and (b). Likewise, they cannot determine if deadlock has occurred as shown in Figure 5.5(c). The derivation trees in (a) are an example of Proposition 3(1), and the trees in (b) are an example of Proposition 3(2). The fourth law can be derived from the first two.

Because of their insensitivity to the branching structure of agents, trace verified systems can deadlock when interconnected. Assume that two components have the branching structure as shown by the left tree of Figure 5.5(b) and they are interconnected and communicate on labels β and γ. Deadlock will occur if one agent decides it will communicate on β and the other decides it will communicate on γ. The traces of the system will contain complete traces, as well as the truncated ones when deadlock occurs, resulting in a "verified" system. This problem is exacerbated by the **hide** operator of trace systems which discards interconnectivity information because all sources of the problem have disappeared once this operator has been applied! When used this way the hide operator is similar to CCS restriction, but it merely influences the traces that are possible, rather than introducing a silent internal action which can effect the external observable actions.

Three questions will be answered at this point. Why would one use trace based systems at all, given such serious flaws? Is there a way to strengthen trace systems to allow them to detect deadlock and to be more sensitive to the branching structure of agents? Lastly, are there other approaches which will give more confidence in the results – effectively disallowing the equational laws in Proposition 3?

5.5.2 Strengthening Trace Verifications

There are two ways to strengthen trace semantics. The first is to use **complete trace** semantics, which represent complete executions. In the case of recursive or nonterminating processes, the complete traces would be infinite. If a regular expression can be generated, as was done in an example above, then infinite sequences can be represented with a finite representation. However, complete traces are still not strong enough to detect deadlock.

A second technique which strengthens trace semantics is to add **failure sets**. Generally, traces only allow the observation of executable sequences. Failure semantics require that one can additionally determine which operations cannot occur for each possible trace. Failure semantics adds the ability to observe some information about the internal branching structure of agents. The CCS definition of a failure, sometimes called *testing equivalence*, is defined in [dNH83] as follows:

Definition 26 *A* ***failure*** *is a pair* (s, L) *where* $s \in \mathcal{L}^*$ *is a trace and* $L \subseteq \mathcal{L}$ *is a set of labels. The failure* (s, L) *belongs to an agent* P *if there exists* P' *such that*

(i) $P \stackrel{s}{\Rightarrow} P'$

(ii) $P' \stackrel{\tau}{\not\rightarrow}$

(iii) $\forall\ \alpha \in L, P \stackrel{\alpha}{\not\rightarrow}$

Failure semantics is extremely busy, as many potential failures must be associated with each partial trace. Some of the failures of the example of Figure 5.2 on page 122, with boxes F and F_1 using the same definition, are shown in Table 5.3. Note that the failures for the trace *in* are different, yet it has been shown that the implementation conforms to the specification. Hence there needs to be some theory for what constitutes a significant difference in failure traces if they are used for conformance.

Specification	**Implementation**
$(\epsilon, \overline{out})$	$(\epsilon, \overline{out})$
(in, in)	(in, ϵ)
$(in\ \overline{out}, \overline{out})$	$(in\ \overline{out}, \overline{out})$

Table 5.3: Failures for Some Matching Traces of the FIFO Example

The most interesting feature of failures semantics is that it is the weakest verification which can test for deadlock. However, it still doesn't significantly distinguish the branching structure of agents. Figure 5.6 shows two agents which cannot be distinguished with failure semantics. See [vG90a] for a more detailed presentation on equivalences and their comparative concurrency semantics.

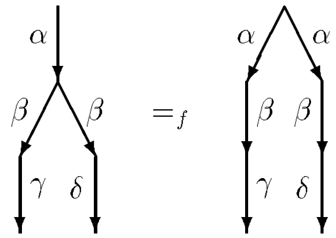


Figure 5.6: Weaknesses in Failures Semantics

5.5.3 Trace Failure Example

The most widely used verifier in the asynchronous community can be attributed to Dill [Dil89]. It was used in the development of the Post Office. Dill's verifier uses modified trace semantics to overcome some of the aforementioned weaknesses in trace theory. Because of the complexity of complete trace semantics, they are not a part of his verifier. Dill uses a failures theory in his verifier, which allows it to distinguish certain classes of the derivation trees in Figure 5.5, but other branching structures such as shown in Figure 5.6 cannot be detected.

An example of the faults that can occur with trace verification is shown. Suppose that a 4-cycle MERGE element is to be built that responds to either an a request or a b request (which are mutually exclusive processes) and produces a request to a third component via a $\overline{c}$ output. The CCS description of the specification is shown in Equation 5.2 as agent E6Spec. Assume that the circuit E6 that is implemented behaves as shown in Equation 5.3. In other words, this process requires that after an a handshake, the b handshake must follow, but after a b handshake, either may proceed.

$$\text{E6Spec} \stackrel{\text{def}}{=} a.\overline{c}.a.\overline{c}.\text{E6Spec} + b.\overline{c}.b.\overline{c}.\text{E6Spec} \tag{5.2}$$

$$\text{E2} \stackrel{\text{def}}{=} a.\overline{c}.a.\overline{c}.b.\overline{c}.b.\overline{c}.\text{E6} + b.\overline{c}.b.\overline{c}.\text{E6} \tag{5.3}$$

The E6 circuit was verified against E6Spec using Dill's burst-mode verifier (ported by Nowick for the burst-mode Post Office application). The input file is shown in Figure 5.7. The circuit that is passed to the verifier is shown in Figure 5.8. This circuit contains two state variables and the output. The description of the circuit is defined as $Y_1 \stackrel{\text{def}}{=} B + Y_0 \times \overline{A} + Y_1 \times A$, $Y_0 \stackrel{\text{def}}{=} \overline{\overline{A}} + Y_0 \times \overline{B}$, and $C \stackrel{\text{def}}{=} Y_1 \times \overline{Y_0} + \overline{Y_1} \times Y_0$.

```
:in    (a b)                 ;list of input variables
:out   (c)                   ;list of output variables
:init-state 0                ;initial state (optional)
:state 0 (a)
       1 (c)
:state 1 (a~)
       0 (c~)
:state 0 (b)
       2 (c)
:state 2 (b~)
       0 (c~)
```

Figure 5.7: E6 Circuit Description for Dill's Verifier

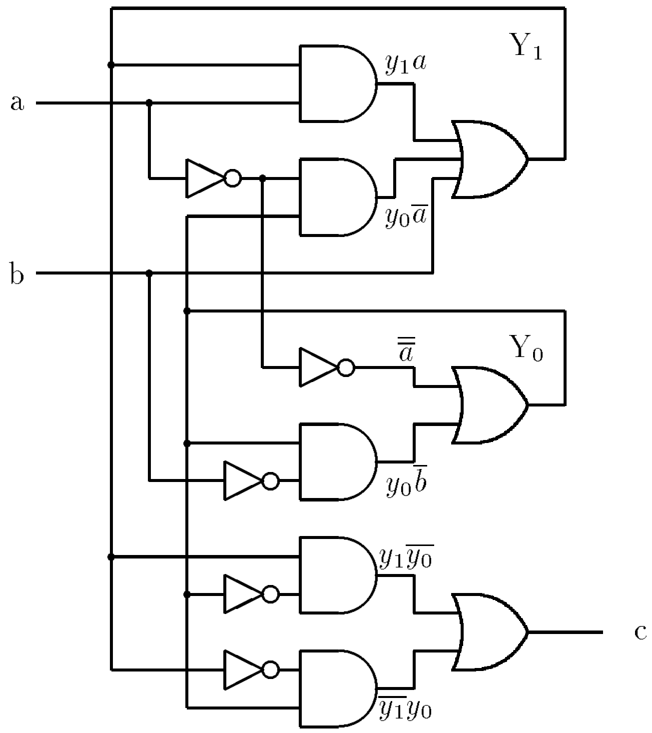


Figure 5.8: Falsely Verified Circuit E6

Dill's burst-mode verifier incorrectly indicates the circuit is a faithful implementation in 79 states. This is most likely due to one of two problems with trace verification methods – an incomplete trace model, or the mirroring of the specification to form the implementation.

Analyze (see Chapter 7.5) points out eight failures in this circuit using its trace conformance verification mode. Computation interference, which is considered an implementation failure, occurs after the $a\,\overline{c}\,a\,\overline{c}\,a$ trace. There is also a deadlock after this trace, which doesn't allow the $\overline{c}$ signal to assert in response to the a input. This occurs when the $Y_1 \times a$ AND gate is unstable[2] producing a high output, and

[2]A gate is considered unstable when changes in the inputs will result in a change on the output that has not yet occured.

the Y_1 OR gate is unstable producing a low output. If the AND gate goes high first, the Y_1 output may remain high, produce a runt pulse, or a static 1 hazard. If the OR output changes first it may stabilize the AND output low, produce a runt pulse, or other races that could result in oscillation.

For this analysis the AND and OR gate specifications are modeled so that when they are unstable, they cannot accept an input that disables the pending output until *after* the output has changed and the gate stabilizes. This results in the deadlock as neither the Y_1 AND nor the y_1a OR gate can fire. From the deadlock, the output $\overline{c}$ cannot be produced, resulting in a verification error with the complete trace semantics of Analyze. This complete analysis is reached in 74 states.

5.5.4 Are Trace Systems Useful?

From the preceding section one could wonder if trace based verification is useful at all given such serious flaws in complexity and analytic coarseness. Trace based systems are too casual, and seem to be overly complex when complete trace and failures are employed. However, experience with Dill's verifier showed the effectiveness of such a tool, and gave good insights into what could be accomplish with Analyze. There are also some mitigating factors which can make trace based systems attractive, which will be discussed in this section.

An asynchronous circuit consists of basic building blocks which are composed in parallel to build larger circuit modules. The building blocks for low latency circuits proposed by this thesis consist of two types of components; asynchronous finite state machines and the nondeterministic, analog mutual exclusion element.

As discussed in Chapter 4, AFSMs are restricted in such a way that they are implementable without combinational logic hazards, and that sequential hazards are controllable. These restrictions require that inputs and outputs occur in bursts, and that any choice by the environment be mutually exclusive. These restrictions also make AFSM modules easier to verify.

Proposition 4 *A minimized burst-mode specification will not contain any τ transitions.*

Proof For a τ transition to exist in a minimized agent, there must be a transition $P \xrightarrow{\tau} P'$ where $P \not\approx P'$. Since burst-mode AFSMs partition activity into distinct input and output phases by Rule 1 on page 83 the τ transition must be part of an input or output burst. By Definitions 1 and 2 each input or output burst must be confluent. Hence by Definition 24(i) no transition $P \xrightarrow{\tau} P'$ can exist where $P \not\approx P'$. $\square$

Proposition 5 *If agent P is determinate and has no τ transitions, then it is also strongly determinate.*

Proof Weakly determinate definitions allow τ transitions to be ignored because they use the transition $Q \stackrel{s}{\Rightarrow} Q'$ where $s \in \mathcal{L}^*$. Since there are no τ transitions, $Q \stackrel{s}{\Rightarrow} Q'$ is equivalent to $Q \xrightarrow{s} Q'$ by Definition 16. Definition 22 states that strong determinism is operational on all derivatives Q of P. Since determinacy is closed under derivation all s-descendants will be derivatives of P. Therefore, only the direct α-derivatives of Q need to be tested, not the s-descendants, making Proposition 5 hold. $\square$

Proposition 6 *Burst-mode AFSMs are determinate.*

Proof Follows from Definitions 1, 2, 23, and 24. □

Proposition 7 *If P and Q are determinate, then $P \approx Q$ iff $P =_t Q$*

Proof Milner [Mil89] page 234. □

If one is to build burst-mode AFSMs then they must be determinate. If the AFSM and its specification are determinate, then trace equivalence suffices as a verification tool! Further, the determinate check can be simplified because it can use strong determinacy, which is a much simpler operation on labeled transition systems. When the agents are minimized this operation is linear on the state space.

Proposition 8 *Trace semantics are not sensitive to determinate agents.*

Proof To be sensitive to determinate systems, one must distinguish between transitions of the type shown in Figure 5.5(a). Trace semantics are not sensitive to determinacy because they cannot distinguish between $a.(b.Nil + c.Nil)$ and $a.b.Nil + a.c.Nil$ □

Proposition 9 *Trace semantics cannot distinguish internal τ choices.*

Proof From Definitions 16 and 25. □

Trace semantics can distinguish neither nondeterminate nor τ transitions. This has two ramifications. Firstly, a stronger semantics must be used to verify that a burst-mode state machine meets its determinate requirements. Section 7.7 discusses some of the extra constraints necessary to prepare state machines for synthesis. CCS semantics has sufficient power to verify determinacy properties.

Secondly, a burst-mode specification can be turned into a **trace determinate specification** – which removes all τ and nondeterminate transitions – because τ actions are indistinguishable using trace semantics. The result allows simple state-by-state matching between the specification and the implementation. *If the implementation is determinate, then trace and bisimilar verifications can be done with linear complexity on the number of states in the specification.* Further, all unreachable states in the implementation are never generated or visited under compositional analysis as is done in the tool developed in this thesis. Additionally, the internal structure of the implementation, including hidden internal transitions, can be preserved over this process, with the unreachable states excluded. This enables the structure of the circuit to be preserved so that the features and capabilities of CCS are not lost in the process.

The good news is that burst-mode state machines can be verified efficiently using trace formalism. Further, many large computational blocks can be determinate, so they too can be efficiently verified. A good example is the register bank in the asynchronous AMULET CPU [PDF+92]. However, a purely trace-based system is not capable of proving the determinate property, a essential step in verifying correct burst-mode specifications (to be pointed out in Section 7.7).

Unfortunately, there are many computational blocks which are nondeterminate due to data dependencies and/or shared resources. The overheads of trace verification can be extremely high for nondeterminate blocks. For example, many required trace transitions from the current implementation state may not be present when verifying the trace conformance of a nondeterministic system against its trace determinate specification. This is not an error unless the trace is not possible from *any* state in the specification. Hence, incomplete trace logs must be stored and matched against completed traces during a verification run to determine if the trace is possible from another state. Only at the end of a run can one determine if the run was successful. This overhead is not present in the logic conformance definitions which follow.

5.6 Logic Conformance

Trace verification has some serious weaknesses because it

- cannot discern determinate systems.
- cannot detect deadlock.
- equates too many branching structures of agents.
- is inefficient for verification of nondeterminate agents.

These are serious weaknesses, as verifications with trace systems cannot detect some of the faults in real circuits, such as those discussed in Section 2.3.4. Particularly crippling is the inability to detect deadlock. Although many of these concerns are not present in small AFSMs, they commonly occur with larger composed systems with data dependencies or shared resources. These systems are the ones which are too complex for designers to analyze in their heads, and where formal methods and automation must be applied. (Although the compositionality of asynchronous circuits alleviates the difficulty of composing systems, there are still ample possibilities for design errors.)

A better conformance definition should have the same properties and equational laws as observational equivalence. The definition of this conformance follows, and is called "observational conformance", or **logic conformance**.

Definition 27 *A binary relation $\mathcal{LC} \subseteq \mathcal{P} \times \mathcal{P}$ over agents is a* ***logic conformation*** *between implementation I and specification S if $(I, S) \in \mathcal{LC}$ then $\forall\ \alpha \in Act$ and $\forall\ \beta \in \overline{\mathcal{A}} \cup \{\tau\}$ (outputs and τ) and $\forall\ \gamma \in \mathcal{A}$ (inputs)*

(i) *Whenever $S \xrightarrow{\alpha} S'$ then $\exists\ I'$ such that $I \stackrel{\widehat{\alpha}}{\Rightarrow} I'$ and $(I', S') \in \mathcal{LC}$*

(ii) *Whenever $I \xrightarrow{\beta} I'$ then $\exists\ S'$ such that $S \stackrel{\widehat{\beta}}{\Rightarrow} S'$ and $(I', S') \in \mathcal{LC}$*

(iii) *Whenever $I \xrightarrow{\gamma} I'$ and $S \stackrel{\gamma}{\Rightarrow}$ then $\exists\ S'$ such that $S \stackrel{\gamma}{\Rightarrow} S'$ and $(I', S') \in \mathcal{LC}$*

Logic conformation is similar to trace conformance, but it is more sensitive to the branching structure of agents. There is also the back and forth comparison between the implementation and the specification which are necessary for a bisimilar relation. Deadlock is detected and must be equal to deadlocks in the specification. Definition 27(ii) requires that all outputs and τ transitions are bisimilar to the specification. This assures that there are no hazards in the implementation and that it produces precisely what the specification dictates. The difference between bisimulation and logic conformance is that inputs not supplied by the environment and their derivative agents (Definition 27(iii)) can be ignored as per Proposition 2.

The back and forth bisimulation nature of Definition 27 assures that Proposition 3(1) on page 126 does not hold. It also catches any dynamic and static hazards the circuit may produce. Clause (ii) checks for correct τ operation and assures that Proposition 3(2) doesn't hold. Without this clause, the τ transition in IMPL the example in the following section would be ignored. This would make IMPL Logically Conformant to SPEC, a case which surely should not hold because IMPL can deadlock while SPEC cannot.

5.6.1 Logic Conformance Example

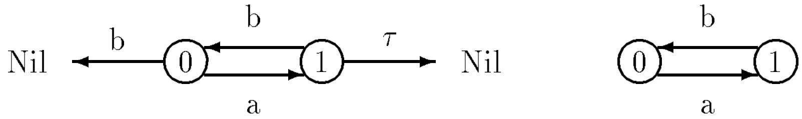
Assume agents E7 and E7Spec are defined as follows:

$$\text{E7Spec} \stackrel{\text{def}}{=} a.b.\text{E7Spec} \tag{5.4}$$

$$\text{E7} \stackrel{\text{def}}{=} a.(\tau.Nil + b.\text{E7}) + b.Nil \tag{5.5}$$

The transition graphs of these agents is shown in Figure 5.9. The implementation E7 is trace conformant to E7Spec, but has no logic conformation. The conformation set $\mathcal{LC}$ starts as a cross product of all ordered pairs of the implementation E7 and specification E7Spec in the order $(\text{E7}, \text{E7Spec})$. There are six such pairs in $\mathcal{LC}$ – $(0,0)(0,1)(1,0)(1,1)(Nil,0)$ and $(Nil,1)$. The last two pairs can be immediately discarded. The pair $(1,0)$ can also be thrown out as the implementation doesn't simulate the specification — it does not have the $\xrightarrow{a}$ transition. The pair $(0,1)$ can likewise be thrown out. The implementation can match the specification's transition requirements through the transitions E7Spec-1 $\xrightarrow{b}$ E7Spec-0 and E7-0 $\xrightarrow{b}$ Nil. Since the pair $(\text{Nil}, 0) \notin \mathcal{LC}$, the pair $(0,1)$ cannot be in $\mathcal{LC}$ and is also thrown out.

Only the pairs $(0,0)$ and $(1,1)$ remain to be checked. The implementation simulates the specification's a transition and requires that $(1,1)$ is in $\mathcal{LC}$. The same occurs for the a transition of the specification. Since the b transition is not a valid transition of E7Spec, Definition 27(iii) is vacuously true. In checking the pair $(1,1)$, note that a bisimulation from state 1 to state 0 exists between the implementation



(a) E7 (b) E7Spec

Figure 5.9: State Graph of Example E7

and specification on label b. Definition 27(ii) requires that the implementation's τ transition is bisimilar to the specification. The only τ transition in the specification that matches E7-1 $\xrightarrow{\tau}$ Nil of the implementation is E7Spec-1 $\overset{\hat{\tau}}{\Rightarrow}$ E7Spec-1. Since the pair $(\mathrm{Nil}, 1)$ is not in $\mathcal{LC}$, the pair $(1, 1)$ is also thrown out. Because $(1, 1)$ is thrown out so is $(0, 0)$. $\mathcal{LC}$ is empty, showing there is no logic conformation between E7 and E7Spec.

If the τ.Nil agent expression were not part of E7 then these agents would be logically conformant. Secondly, if the b input were changed to an output $\overline{b}$, then E7 is no longer trace conformant to the specification because there is an output possible in the initial state before an a input has arrived.

5.6.2 Properties of Logic Conformance

Because the definition of Proposition 27 uses sets, there are many possible solutions, including the empty relation $\mathcal{LC} = \epsilon$. Therefore, this definition must be put into a different form which will be the largest conformance. This is done by examining some of the properties that are preserved by various operations on relations.

Proposition 10 *Assume that each $\mathcal{LC}_i$ $(i = 1, 2, \ldots)$ is a logic conformation. Then the following relations are all logic conformantions:*

$$\text{(1) } Id_{\mathcal{P}} \qquad \text{(2) } \mathcal{LC}_1\,\mathcal{LC}_2 \qquad \text{(3) } \bigcup_{i \in I} \mathcal{LC}_i$$

Proof

(1) The identity can be defined as a relation $\mathcal{R} = \{(x, x) : (x, x) \in \mathcal{R}\}$

Suppose that for some P, $(P, P) \in \mathcal{LC}$. Each action $P \xrightarrow{\alpha} P'$, $P \xrightarrow{\beta} P'$, and $P \xrightarrow{\gamma} P'$ from Proposition 27 can be equaled by the same action, so $P \overset{\widehat{\alpha}}{\Rightarrow} P'$, $P \overset{\widehat{\beta}}{\Rightarrow} P'$, and $P \overset{\widehat{\gamma}}{\Rightarrow} P'$ all hold. Since $(P, P) \in \mathcal{LC}, (P', P') \in \mathcal{LC}$.

(2) The composition of two binary relations is defined as the relation

$\mathcal{R}_1\,\mathcal{R}_2 = \{(x, z) : \text{for some } y, (x, y) \in \mathcal{R}_1 \text{ and } (y, z) \in R_2\}$

First the most general case is exercised, going from right to left. Suppose that $(P, R) \in \mathcal{LC}_1\,\mathcal{LC}_2$. Then for some Q there must be $(P, Q) \in \mathcal{LC}_1$ and $(Q, R) \in \mathcal{LC}_2$.

Let $R \xrightarrow{\alpha} R'$. By Definition 27, since $(Q, R) \in \mathcal{LC}_2$,

$$Q \overset{\widehat{\alpha}}{\Rightarrow} Q' \text{ and } (Q', R') \in \mathcal{LC}_2$$

Since $(P, Q) \in \mathcal{LC}_1$, if $Q \xrightarrow{\alpha} Q'$, $\exists\ P'$ such that

$$P \overset{\widehat{\alpha}}{\Rightarrow} P' \text{ and } (P', Q') \in \mathcal{LC}_1$$

From Definition 15, $Q \stackrel{\widehat{\alpha}}{\Rightarrow} Q'$ can be rewritten as $Q \stackrel{\tau^*\alpha\tau^*}{\longrightarrow} Q'$.

Also, since $(P, Q) \in \mathcal{LC}_1$, if $Q \stackrel{\tau}{\rightarrow} Q''$ then we have some P'' such that

$$P \stackrel{\tau^*}{\rightarrow} P'' \text{ and } (P'', Q'') \in \mathcal{LC}_1$$

Therefore, if $Q \stackrel{\tau^*}{\rightarrow} Q'''$ then there exists a P''' such that

$$P \stackrel{\tau^*}{\rightarrow} P''' \text{ and } (P''', Q''') \in \mathcal{LC}_1$$

By Definition 15 $P \stackrel{\widehat{\tau^*}}{\Rightarrow}\stackrel{\widehat{\alpha}}{\Rightarrow}\stackrel{\widehat{\tau^*}}{\Rightarrow}$ is equivalent to $P \stackrel{\widehat{\alpha}}{\Rightarrow}$. Since $(P''', Q''') \in \mathcal{LC}_1$ and $P \stackrel{\widehat{\tau}}{\Rightarrow} P'''$ then there exists a P'''' where if $Q \stackrel{\tau^*}{\rightarrow} Q''' \stackrel{\alpha}{\rightarrow} Q''''$ then

$$P \stackrel{\widehat{\alpha}}{\Rightarrow} P'''' \text{ and } (P'''', Q'''') \in \mathcal{LC}_1$$

By similar reasoning, since $(P'''', Q'''') \in \mathcal{LC}_1$, there exists a P' such that if $Q \stackrel{\tau^*}{\rightarrow} Q''' \stackrel{\alpha}{\rightarrow} Q'''' \stackrel{\tau^*}{\rightarrow} Q'$ then

$$P \stackrel{\widehat{\alpha}}{\Rightarrow} P' \text{ and } (P', Q') \in \mathcal{LC}_1$$

Since $Q \stackrel{\tau^*\alpha\tau^*}{\longrightarrow} Q'$ is equivalent to $Q \stackrel{\widehat{\alpha}}{\Rightarrow} Q'$ and if $Q \stackrel{\widehat{\alpha}}{\Rightarrow} Q'$ and $(P, Q) \in \mathcal{LC}_1$ then there exists P' such that $P \stackrel{\widehat{\alpha}}{\Rightarrow} P'$ and $(P', Q') \in \mathcal{LC}_1$.

Hence $(P', R') \in \mathcal{LC}_1\mathcal{LC}_2$ from right to left

Going from right to left for β and γ transitions uses the same structure as above. Going from left to right for β transitions also follows the same structure.

The γ transitions are what create the partial order. Any transition in R will have similar transitions in Q and P as shown above. Likewise, any transition in Q will have a similar transition in P. However, input (γ) transitions may exist in P that do not exist in Q, and likewise for Q and R. Using similar reasoning as above, suppose $(P, R) \in \mathcal{LC}_1\mathcal{LC}_2$. Then $(P, Q) \in \mathcal{LC}_1$ and $(Q, R) \in \mathcal{LC}_2$.

Let $P \xrightarrow{\gamma} P'$. If there is no transition $Q \stackrel{\gamma}{\Rightarrow}$ then we are done. If $Q \stackrel{\gamma}{\Rightarrow}$ then

$$Q \stackrel{\gamma}{\Rightarrow} Q' \text{ and } (P', Q') \in \mathcal{LC}_1.$$

If $Q \stackrel{\gamma}{\Rightarrow} Q'$ and $R \stackrel{\gamma}{\not\Rightarrow}$ then we are done.

Definition 16 allows $Q \stackrel{\gamma}{\Rightarrow} Q'$ to be rewritten as $Q \xrightarrow{\tau^*\gamma\tau^*} Q''$. Since β transitions include the τ transition, and all β transitions hold between P, Q and Q, R, the above reasoning can be used to deduct that since $(Q, R) \in \mathcal{LC}_2$ and if $Q \stackrel{\gamma}{\Rightarrow} Q'$ and $R \stackrel{\gamma}{\Rightarrow}$ then there exists an R' such that

$$R \stackrel{\gamma}{\Rightarrow} R' \text{ and } (Q', R') \in \mathcal{LC}_2$$

Hence $(P', R') \in \mathcal{LC}_1\mathcal{LC}_2$

(3) For all $\mathcal{LC}_i$, the largest set is created by the union of all such sets. If this were not the case, a pair would exist $(P, Q) \in \bigcup_{i\in I}\mathcal{LC}_i$ that forced $(P', Q') \notin \mathcal{LC}$. But if that were the case, (P, Q) would not be in $\mathcal{LC}$.

$\square$

Definition 28 *Implementation I is* ***logic conformant*** *to specification S, written $I \succeq_l S$ if $(I, S) \in \mathcal{LC}$ form some logic conformation $\mathcal{LC}$. This may be equivalently expressed as* $\succeq_l = \bigcup\{\mathcal{LC} : \mathcal{LC} \textit{ is a logic conformation}\}$

Proposition 11

(1) $\succeq_l$ *is the largest logic conformation*

(2) $\succeq_l$ *is a partial order*

Proof

(1) By proposition 10(3), $\succeq_l$ is a logic conformation and includes any other such conformation.

(2) *Reflexivity:* For any $P, P \succeq_l P$ by Proposition 10(1).

Transitivity: If $P \succeq_l Q$ and $Q \succeq_l R$ then $(P, Q) \in \mathcal{LC}_1$ and $(Q, R) \in \mathcal{LC}_2$ for logic conformances $\mathcal{LC}_1, \mathcal{LC}_2$. Therefore, by Proposition 10(2), $(P, R) \in \mathcal{LC}_1 \mathcal{LC}_2$, and so $P \succeq_l R$.

Antisymmetric: If $P \succeq_l Q$ and $Q \succeq_l P$ then $P \approx Q$ (bisimilar equivalence) by the definitions of Bisimulation and logic conformance. $P \succeq_l Q$ does not imply $Q \succeq_l P$. For example, let E8-0 $\stackrel{\text{def}}{=} a.$E8-0 and E8-1 $\stackrel{\text{def}}{=} a.$E8-1 $+ \, b.$E8-1. Clearly E8-1 $\succeq_l$ E8-0 and E8-0 $\not\succeq_l$ E8-1

□

Definition 29 *Define the function $\mathcal{F}$, over subsets of binary relations over agents $\mathcal{P} \times \mathcal{P}$ such that if $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$, then $(P, Q) \in \mathcal{F}(R)$ iff $\forall\ \alpha \in Act$ and $\forall\ \beta \in \overline{\mathcal{A}} \cup \{\tau\}$ and $\forall\ \gamma \in \mathcal{A}$*

(i) *Whenever $S \xrightarrow{\alpha} S'$ then $\exists\ I'$ such that $I \stackrel{\widehat{\alpha}}{\Rightarrow} I'$ and $I' \mathcal{R} S'$*

(ii) *Whenever $I \xrightarrow{\beta} I'$ then $\exists\ S'$ such that $S \stackrel{\widehat{\beta}}{\Rightarrow} S'$ and $I' \mathcal{R} S'$*

(iii) *Whenever $I \xrightarrow{\gamma} I'$ and $S \stackrel{\gamma}{\Rightarrow}$ then $\exists\ S'$ such that $S \stackrel{\gamma}{\Rightarrow} S'$ and $I' \mathcal{R} S'$*

Although $\mathcal{F}$ operates on any R, it is used in such a manner that it succinctly defines logic conformance.

Proposition 12

(1) *$\mathcal{F}$ is monotonic. If $R_1 \subseteq R_2$ then $\mathcal{F}(R_1) \subseteq \mathcal{F}(R_2)$*

(2) *$\mathcal{LC}$ is a logic conformance iff $\mathcal{F} \subseteq \mathcal{F}(\mathcal{LC})$*

Proof (1) follows directly from Definition 29, and (2) reformulates Definition 27. □

$\mathcal{R}$ is called a *pre-fixed-point* of $\mathcal{F}$ if $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$. Also, $\mathcal{R}$ is a *fixed-point* of $\mathcal{F}$ if $\mathcal{R} = \mathcal{F}(\mathcal{R})$. Therefore logic conformations are exactly the pre-fixed-points of $\mathcal{F}$, and it will be further shown that $\succeq_l$, the largest pre-fixed-point, is a fixed-point of $\mathcal{F}$.

Proposition 13 *Logic conformance is a fixed point of $\mathcal{F}$, hence $\succeq_l = \mathcal{F}(\succeq_l)$. Logic conformance is likewise the maximum fixed-point of $\mathcal{F}$.*

Proof Since $\succeq_l$ is a logic conformation, $\succeq_l \subseteq \mathcal{F}(\succeq_l)$ by Definition 29. Because $\mathcal{F}$ is monotonic, $\mathcal{F}(\succeq_l) \subseteq \mathcal{F}(\mathcal{F}(\succeq_l))$, so $\mathcal{F}(\succeq_l)$ is also a pre-fixed-point of $\mathcal{F}$. Since $\succeq_l$ is the largest pre-fixed-point of $\mathcal{F}$ (from Proposition 11), it includes $\mathcal{F}(\succeq_l)$, so $\mathcal{F}(\succeq_l) \subseteq \succeq_l$. Therefore $\succeq_l = \mathcal{F}(\succeq_l)$. Because $\succeq_l$ is the largest pre-fixed-point, it is also the maximum fixed-point of $\mathcal{F}$. □

Because $\succeq_l$ is the maximum fixed point, logic conformance can now be defined as follows:

Definition 30 ***Logic conformance*** *between implementation I and specification S is written as* $I \succeq_l S$ *and holds iff* $\forall\, \alpha \in Act$ *and* $\forall\, \beta \in \overline{\mathcal{A}} \cup \{\tau\}$ *and* $\forall\, \gamma \in \mathcal{A}$

(i) *Whenever* $S \xrightarrow{\alpha} S'$ *then* $\exists\, I'$ *such that* $I \stackrel{\widehat{\alpha}}{\Rightarrow} I'$ *and* $I' \succeq_l S'$

(ii) *Whenever* $I \xrightarrow{\beta} I'$ *then* $\exists\, S'$ *such that* $S \stackrel{\widehat{\beta}}{\Rightarrow} S'$ *and* $I' \succeq_l S'$

(iii) *Whenever* $I \xrightarrow{\gamma} I'$ *and* $S \stackrel{\gamma}{\Rightarrow}$ *then* $\exists\, S'$ *such that* $S \stackrel{\gamma}{\Rightarrow} S'$ *and* $I' \succeq_l S'$

Proposition 14 *if* $P \succeq_l Q$ *and* $Q \succeq_l P$ *then* $P \approx Q$

Proof Follows from Definitions 21 and 30 □

5.7 Summary

Two formalism have been presented, based on trace and bisimulation semantics, for comparing circuit specifications and implementations. Informally, these are designed to maximize the implementations that will conform to the specification while retaining the correct operation as far as the model's capabilities allow.

Trace conformance is useful for simple cases where the circuit implementation is determinate. Then verification can be done linearly on the state space of these processes. An additional feature is that trace equivalence also implies the preferred notion of logic conformance when the specification and implementation are determinate. For these simple examples, logic conformance is no more complex to calculate than trace conformance, and can be done linearly on the state space as well.

For the more difficult cases of hierarchical, complex, or nondeterminate logic, the trace model becomes slow and inefficient. Trace verification of these models can also validate circuits which will not operate as desired. logic conformance contains sufficient rigor to verify circuits hierarchically, while catching hazards and inconsistent behavior caused at any level in the hierarchy. The computational complexity of logic conformance is significantly less than that of complete trace failures testing. When no τ transitions exist, the check is linear on the state space in time.

Several attempts at creating loose specifications have been made in the theoretical community [CS90, DHWT91, FM91, Lar89, Xin92]. These groups have used partial orders to achieve the looseness (or partiality) of specifications for behavioral systems. Some of the partial orders include $\frac{2}{3}$ bisimulation, divergence preorders, and network preorders. Although these may have practical applications in some areas, none are appropriate for asynchronous hardware verification. Practical applications have also been applied by using a "mirroring" or inversion operation on the specification and composing that with the implementation. The goal of the mirrored specification is to supply a restricted environment for the implementation. These methods either place constraints on systems which are unreasonable, hide hazardous behavior, or are not "safe" as they permit hazards and illegal transitions to occur in the implementation with handshaking communication.

A significant contribution of this thesis is the definition of a generally applicable approach to loose behavioral specifications called *conformances*. A conformance is a partial order which restricts the behavior of an implementation to contain at least all the behaviors of the specification, and to exclude any bad behaviors. Definitions for trace conformance and bisimulation (or logic) conformance have been presented.

A second contribution to the asynchronous community is the ability to apply a larger range of equivalence formalisms to verification, as only trace based systems have previously been available. The tools developed with this thesis include verification based on trace and branching time bisimulation formalisms.

Proving pre-congruence on the conformance partial orders results in many advantages for hardware synthesis. The primary advantage is that conformance now supports compositional (or substitutive) replaceability between specifications and their looser implementations. This allows systems to be hierarchically synthesized in a top-down fashion, where conformance only needs to be proven at each level in the hierarchy.

Many safety features cannot be verified hierarchically, so good global planning is still necessary to achieve the full potential of the tools. Good planning is also necessary to hide the explosion in complexity due to parallelism.

Chapter 6

Practical Applications of Process Logics

Process logics can verify that a behavior holds invariantly across a set of states. Such invariant analysis is analogous to simulation, but circuit simulation only tests the timing and threads of behavior explicitly exercised by the simulation vectors. For instance, deadlock might be discovered using simulation, but with invariants the deadlock *property* will or will not exist be found to exist. Temporal logics can be used to prove that the existence of behavioral invariants and properties that are essential for reliable circuit implementations.

This chapter applies process logics to the practical verification of behaviors and properties of asynchronous circuits. First Hennessey-Milner logic and the modal-μ calculus are introduced. Invariant properties are then broken into two categories as they relate to circuit descriptions – application independent and application specific formulae. Independent formulae are generic for all circuit implementations, while the application specific properties rely on the circuit specification and implementation. Deadlock and liveness are the independent invariant properties of primary importance. Some new definitions of these properties, analogous to the liveness definition or Petri net theory, are developed here. They are more appropriate definitions for parallel VLSI than the forms typically used with temporal logics. New application specific invariants are presented and some old forms are strengthened so that they are appropriate for hardware. Finally, an example of applying temporal logic as a conformance verification is presented.

6.1 Hennessey-Milner Logic

Hennessey-Milner logic (**HML**) is a process logic that applies the following logic formulae to labeled transition systems [Sti92]. Recall the definition of labeled transition systems from Definition 9: $(\mathcal{P}, Act, \{\xrightarrow{\alpha} : \alpha \in Act\})$ where $\xrightarrow{\alpha}$ maps to the agents in $\mathcal{P} \times \mathcal{P}$. Terms in Hennessey-Milner logic are defined as:

$$A ::= T \mid \neg A \mid A \wedge A \mid \langle a \rangle A \tag{6.1}$$

where

- T is a constant representing a true formula
- $\neg A$ negates the formula A
- $A \wedge A$ is the conjunction of two formulae
- $\langle a \rangle A$ is the modalized term where the formula A holds after some action a.

A modalized operator is a formula that makes an assertion about a changing state. A common set of duals apply, including $\neg \mathrm{T} \stackrel{\mathrm{def}}{=} \mathrm{F}$, the disjunctive operator $\neg(\neg A \wedge \neg B) \stackrel{\mathrm{def}}{=} A \vee B$, and a second modalized term $\neg\langle a \rangle \neg A \stackrel{\mathrm{def}}{=} [a]A$. Each agent has its own HML system because formulae are parameterized by an action set. The action set of the formulae should be a subset of the sort of the processes being analyzed.

HML is applied to express conditions on the behavior of agents, which can be formally defined by the *satisfaction* relation '$\models$'.

Definition 31 *The **satisfaction** relation $\models$ between terms in a process logic $A \in \mathcal{PL}$ and the agent set $P \in \mathcal{P}$ is defined by induction on the structure of formulae as:*

$$
\begin{array}{llll}
(i) & P & \models & T \qquad\qquad\qquad\qquad \forall P \\
(ii) & P & \models & \neg A \qquad\qquad\qquad \textit{iff } P \not\models A \\
(iii) & P & \models & \bigwedge_{i \in I} A_i \qquad\qquad \textit{iff } \forall\ i \in I, P \models A_i \\
(iv) & P & \models & \langle\alpha\rangle A \quad \textit{iff } \exists\ P' \textit{ where } P \xrightarrow{\alpha} P' \textit{ and } P' \models A
\end{array}
$$

Consequently, the derived dual operators introduced above can also have the satisfaction relation defined. Of particular interest is the modalized term $[a]A$. For this term

$$P \models [a]A \ \text{iff} \forall\ P' \text{where} P \xrightarrow{\alpha} P' \text{then} P' \models A \tag{6.2}$$

Note that the modal equation $[\alpha]A$ can be *vacuously true*. If the α action is not possible from P, then the equation is satisfied. If the action *is* possible, then the equation is true if and only if for all actions $P \xrightarrow{\alpha} P'$, A is satisfied by all states P'.

Agents can now be tested for satisfaction of specific properties using HML. For example, Table 6.1 shows various property and behavioral tests for the standard C-element definition $\mathrm{C} \stackrel{\mathrm{def}}{=} a.b.\overline{c}.\mathrm{C} + b.a.\overline{c}.\mathrm{C}$.

$\mathrm{C} \models [\overline{c}]\mathrm{F}$	C cannot make a $\overline{c}$ transition
$\mathrm{C} \models \langle a\rangle\mathrm{T}$	C can make an a transition
$\mathrm{C} \models \langle b\rangle\mathrm{T}$	C can make a b transition
$\mathrm{C} \models \langle a\rangle\langle b\rangle\langle\overline{c}\rangle\mathrm{T}$	C can make a, then b, then $\overline{c}$ transitions.
$\mathrm{C} \models [a]\mathrm{F}\wedge[b]\mathrm{F}\wedge[\overline{c}]\mathrm{F}$	C is deadlocked (false)
$\mathrm{C} \models \langle a\rangle\mathrm{T}\wedge[b]\mathrm{F}\wedge[\overline{c}]\mathrm{F}$	C can only make an a transition (false)

Table 6.1: HML Formulae Testing a C-element

Such tests can be used to examine specifications of asynchronous circuits [Liu92]. Notational extensions can be used which clarify the meaning of a modal formula and reduce the chance of an error. Multiple actions can be placed inside a single modal formula which can greatly clarified formulae for testing the C-element and other processes.

Definition 32

$$\langle a, b\rangle A \quad \stackrel{\text{def}}{=} \quad \langle a\rangle A \vee \langle b\rangle A$$

and dually,

$$[a, b, \overline{c}]A \quad \stackrel{\text{def}}{=} \quad [a]A \wedge [b]A \wedge [\overline{c}]A$$

Further, the '$-$' character can be used to substitute for any action of an agent.

Definition 33

$$\langle -\rangle A \quad \stackrel{\text{def}}{=} \quad \bigvee_{\forall \alpha \in Act} \langle \alpha\rangle A$$

$$\langle -a, b\rangle A \quad \stackrel{\text{def}}{=} \quad \bigvee_{\forall \alpha \in Act \ and \ \alpha \neq a, b} \langle \alpha\rangle A$$

and dually,

$$[-]A \quad \stackrel{\text{def}}{=} \quad \bigwedge_{\forall \alpha \in Act} [\alpha]A$$

$$[-a, b]A \quad \stackrel{\text{def}}{=} \quad \bigwedge_{\forall \alpha \in Act \ and \ \alpha \neq a, b} [\alpha]A$$

At times a weaker transition rule may be desired when reasoning about circuit behaviors. Rather than using the relation $\stackrel{\alpha}{\rightarrow}$ over $\mathcal{P}$ for modal formulae, the weaker transition relation $\stackrel{\widehat{\alpha}}{\Rightarrow}$, defined in Proposition 15, can be used. For such transitions, the modal formula $[\alpha]$A is replaced with $[[\alpha]]$A and $\langle\alpha\rangle$A is replaced with $\langle\langle\alpha\rangle\rangle$A where the abbreviations preserve their duality.

6.2 Modal-μ Calculus

All interesting hardware agents are reusable and thus have recursive definitions in CCS. An agent set $\mathcal{R}$ ($\mathcal{R} \subseteq \mathcal{P}$) is called a **fixed point** of the function $\mathcal{F}$ if $\mathcal{R} = \mathcal{F}(\mathcal{R})$. For example, the behavior of a C-element keeps repeating itself so a fixed point of the C-element can be expressed with the following formula in which C is the fix point variable.

$$C = \langle a \rangle \langle b \rangle \langle \overline{c} \rangle C \tag{6.3}$$

There may be many solutions or no solution for a fixed point formula. However, if a fixed point variable is prefixed by an even number of negations, minimal and maximal solutions are guaranteed to exist [Tar55]. Therefore, forms with an odd number of negations are not interesting. Multiple solutions can exist because a property (or formula) is associated with a set of states. When multiple solutions exist, the sets form a lattice with unique minimal and maximal solutions. There is no quick way to compute all the fixed points of an equation – all possible sets of states must be examined. However, there are efficient algorithms for finding the *minimal* and *maximal* fixed point solutions. Luckily these are the ones in which all of our characterizing properties can be expressed. The maximal fixed point is calculated by iteration starting with all states in the system, throwing out states that are necessarily false. The minimum fixed point iteration starts with the empty set and includes the states which are necessarily true.

Modal-μ extends Hennessey-Milner logic with fixed points:

$$A ::= HML \mid min(X.A) \mid max(X.A) \mid X \tag{6.4}$$

where X is a fixed point variable. These fixed point variables allow properties to be tested as invariants across the entire set of states in an agent. Two standard branching time temporal logic operations based on fixed points are the box '$\Box$' and diamond '$\Diamond$' operators [MP92, And93].

Definition 34 *The* ***always*** *operator* $\Box$ *is defined as*

$\Box P \stackrel{\text{def}}{=} max(X.P \wedge [-]X)$

Definition 35 *The* ***possibility*** *operator* $\Diamond$ *is defined as*

$\Diamond P \stackrel{\text{def}}{=} min(X.P \vee \langle - \rangle X)$

The $\Box$ (BOX) macro assures that the property P holds invariantly across all reachable states from a process if Process $\models \Box P$. The $\Diamond$ (POSS) macro holds when any state in the system reachable from the process has property P. These operators are duals of each other in the sense that $\Box P = \neg\Diamond\neg P$. Other useful macros that have been applied to circuit verification include the EVENTUALLY and PATH macros (which are also duals) [Liu92, LABS93]. EVENTUALLY ensures that there is at least one state on every trace that contains property P, whereas the PATH operator verifies that the property P holds on each state in at least one trace.

Definition 36 *The* ***eventually*** *operator EVENTUALLY is defined as*

EVENTUALLY $P \stackrel{\text{def}}{=} min(X.P \vee [-]X)$

Definition 37 *The* ***path*** *operator is defined as*

$$\text{PATH } P \stackrel{\text{def}}{=} max(X.P \wedge \langle - \rangle X)$$

Refer back to the example in Table 6.1 on Page 150. Note that the HML test for a deadlocked C-element only assures that the initial state does not deadlock (much like a simulation test). The modal-μ macro from Definition 38 can be applied to prove that the circuit does not contain the "temporal logic" deadlocking *property*.

Definition 38 *A process contains the* ***complete deadlock*** *property if it is satisfied by:* $\Diamond[-]\text{F}$ *or its dual* $\neg(\Box\langle - \rangle \text{T})$

This formula is satisfied if there is any state in the system where no action is possible. Hence the system can become stuck where it cannot make any moves at all. The Concurrency Workbench can be used to test this and other properties using the modal-μ formulae presented throughout this chapter.

6.3 Application Independent Invariant Properties

Application independent invariant properties do not rely on an agent's sort or structure. The properties presented in this section are essential for well defined circuit specifications.

6.3.1 Deadlock

The complete system deadlock expressed in Definition 38 is too restrictive to be useful for most hardware implementations that employ parallelism because it requires that an agent can arrive in a state where *no* action can be performed. The definition of a live system in Petri net theory more closely models the hardware designers mental model of a deadlock [Pet81].

For example, assume a handshake interface is implemented as shown in Figure 6.1. Upon receiving a request, the circuit can move into state 1 where it will handshake correctly by responding with an acknowledgment, or it will nondeterministically move to state 2 where it will accept any number of requests but will never respond with an acknowledgment.

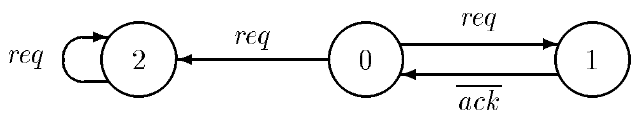


Figure 6.1: Weakly Deadlocking Handshake

This agent does not contain the complete system deadlock of Definition 38. However, this agent *does* contain a deadlock on the $\overline{ack}$ signal, because it is possible to arrive in a state where the circuit will no longer issue an acknowledgment. The following more useful definitions of deadlock for parallel systems are proposed:

Definition 39 *A process $P \in \mathcal{P}$ contains a* ***deadlock*** *if $\exists\ \alpha \in \mathcal{L}(P)$ which allows P to satisfy the following formula:*

$$\text{DEADLOCK } \alpha \overset{\text{def}}{=} \Diamond\Box[\alpha]\text{F}$$

Definition 40 *A process $P \in \mathcal{P}$ contains a* ***strong deadlock*** *if $\exists\ \alpha \in \mathcal{L}(P)$ which allows P to satisfy the following formula:*

$$\text{SDEADLOCK } \alpha \overset{\text{def}}{=} \Diamond \mathit{PATH}[\alpha]\text{F}$$

The stronger and weaker forms of deadlock in Definition 39 and 40 are similar. The states that satisfy deadlock are a subset or equal to the states that satisfy strong deadlock for any process. They both assure that it is possible to arrive in a state where the action being checked cannot occur, nor can the action occur in all states that are reachable from that state. These deadlock definitions are weaker than the Petri net definition since they test liveness of *labels* rather than *transitions*. The stronger version is satisfied if there is some path where the action α cannot occur, whereas the weaker version is satisfied only if there is no reachable state where the action can occur. Strong deadlock considers *livelock* because it may be possible that some path is always taken which never allows a label to be exercised. The stronger deadlock may be satisfied for nondeterminate agents, and so the weaker version is typically used. However, for individual burst-mode AFSMs the stronger version should be used.

This definition of deadlock is an important invariant test, because dead branches of a parallel circuit can be extremely difficult to detect using simulation techniques. An example of such a failure can be shown with the distributed arbiter example taken from [SABL93]. The distributed arbiter is specified as shown in Table 6.2. This definition does not contain deadlock since DINode $\not\models$ DEADLOCK α, $\forall\, \alpha \in \mathcal{L}$(DINode). The possibility of livelock results in the strong deadlock property holding for the signals $\overline{grant}$, $done$, and $\overline{ack}$ because DINode $\models$ SDEADLOCK $\overline{grant}$. This livelock occurs when the token never chooses to serve the user interface of the arbiter, resulting in starvation.

Again, suppose that one of the nodes of the distributed arbiter uses a different definition for the interface, where the $\overline{ack}$ signal has inadvertently been left out, and the definition for 'Interface' is replaced with the definition of "LameInterface" in Table 6.3. The module interfacing with the lame arbiter interface will wait for the

$$\begin{array}{lcl}
\text{Interface} & \stackrel{\text{def}}{=} & req.\overline{ok}.\overline{grant}.done.\overline{ko}.\overline{ack}.\text{Interface} + \overline{no}.\text{Interface} \\
\text{Token} & \stackrel{\text{def}}{=} & tin.(ok.ko.\overline{tout}.\text{Token} + no.\overline{tout}.\text{Token}) \\
\text{DINode} & \stackrel{\text{def}}{=} & (\text{Interface} \mid \text{Token})\backslash\{ok, ko, no\}
\end{array}$$

Table 6.2: Distributed Arbiter Definition

$\overline{ack}$ handshake which will never occur, resulting in a *partial* system deadlock. This type of deadlock occurred in the first silicon of the Post Office [SDC93], resulting in months of work to discover the cause of the flaw. Unaffected portions of the chip continued to function properly, while an entire logic block was deadlocked. The weak form of deadlock detection from (Definition 39) detects this situation as can be seen by the satisfaction LameInterface $\models$ DEADLOCK $\overline{ack}$.

$$\text{LameInterface} \stackrel{\text{def}}{=} req.\overline{ok}.\overline{grant}.done.\overline{ko}.\text{LameInterface} + \overline{no}.\text{LameInterface}$$

Table 6.3: Erroneous Distributed Arbiter Interface

6.3.2 Liveness

A new definition for liveness of agents in labeled transition systems is proposed. This definition is intended for complex parallel systems and is an application independent invariant that can be automatically checked for any process. A live process must be capable of exercising every label in the process from every state in the process. This definition is similar to that of liveness in petri nets [Pet81]. Strong and weak liveness definitions using modal-μ calculus are presented in Definition 42 and 43. The new liveness definitions are the duals of the deadlock definitions as can be seen by examining Definition 39 and 40. Hence a system that is live will not contain deadlock, and a system that contains deadlock will not be live, and only one of the tests is required to assure the live and deadlock free properties of a process.

The strong definition of liveness requires the strengthening of the EVENTUALLY macro of Definition 36. Note that strong modalities (such as $[-]X$) are vacuously true when no label maps the the label argument. For example, the test $\Box[a]\langle b\rangle\mathrm{T}$ will satisfy *any* process that cannot do an a action, including the agent Nil. Hence a stronger version of EVENTUALLY, defined as EV in [Liu92] is required. This definition ensures that the argument to the fixed point variable using a strong modality must contain at least one label or it will fail.

Definition 41 *The* ***eventually*** *operator EV is defined as*

$$\mathrm{EV}\ P \stackrel{\mathrm{def}}{=} min(X.P \vee ([-]X \wedge \langle -\rangle \mathrm{T}))$$

Definition 42 *A process $P \in \mathcal{P}$ is* ***live*** *if $\forall\ \alpha \in \mathcal{L}(P), P$ is satisfied by the following formula:*

$$\mathrm{LIVE}\ \alpha \stackrel{\mathrm{def}}{=} \Box\Diamond\langle\alpha\rangle\mathrm{T}$$

Definition 43 *A process $P \in \mathcal{P}$ is* ***strong live*** *if $\forall\ \alpha \in \mathcal{L}(P), P$ is satisfied by the following formula.*

$$\mathrm{SLIVE}\ \alpha \stackrel{\mathrm{def}}{=} \Box\mathrm{EV}\langle\alpha\rangle\mathrm{T}$$

The strong definition of liveness proves that not only can every signal in the system be exercised from each state, but that it must also be *fair* in that no activity can preempt the occurrence of other signals. The possibility of service may exist, but the signal also may never be served.

For example, all of the signals in the distributed arbiter of Table 6.2 are live. However, the $\overline{grant}$, $done$, and $\overline{ack}$ signals are not strongly live. The blocking nature of the arbiter allows paths to be chosen which can continually select the token interface over the user interface of the arbiter, resulting in livelock. This is proven by the tests DINode $\models$ LIVE $\overline{grant}$ and DINode $\not\models$ SLIVE $\overline{grant}$.

As a final example, a FIFO storage management controller explained by Dill et. al. in [DNS92] was specified using CCS in [LABS93]. The CCS specification is shown in Table 6.4. The circuit does not contain a complete system deadlock as proven by Definition 38. However, both the live and strong live definitions from Definitions 42 and 43 prove that the $\overline{err}$ signal can deadlock (CSM $\not\models$ LIVE $\overline{err}$). A transition occurs in this design on $\overline{err}$ when an illegal access has occurred through the controller – either an under or overflow of the FIFO. If the $\overline{err}$ signal cannot occur, then the controller is correctly specified in that aspect. This circuit will be examined further in Section 6.4.3.

$$
\begin{array}{lcl}
\mathrm{CSM} & \stackrel{\mathrm{def}}{=} & (\mathrm{W} \mid \mathrm{E} \mid \mathrm{C0} \mid \mathrm{S})\backslash\{down, up, f, nf, e, gS, pS\} \\
\mathrm{W} & \stackrel{\mathrm{def}}{=} & wr.nf.gS.\overline{up}.din.\overline{pS}.\overline{wa}.\mathrm{W} \\
\mathrm{E} & \stackrel{\mathrm{def}}{=} & cr.ne.gS.\overline{dout}.\overline{down}.\overline{pS}.\overline{ca}.\mathrm{E} \\
\mathrm{S} & \stackrel{\mathrm{def}}{=} & \overline{gS}.pS.\mathrm{S} \\
\mathrm{C0} & \stackrel{\mathrm{def}}{=} & down.\overline{err}.\mathrm{Nil} + up.\mathrm{C1} + \overline{nf}.\mathrm{C0} + \overline{e}.\mathrm{C0} \\
\mathrm{C1} & \stackrel{\mathrm{def}}{=} & down.\mathrm{C0} + up.\mathrm{C2} + \overline{nf}.\mathrm{C1} + \overline{ne}.\mathrm{C1} \\
\mathrm{C2} & \stackrel{\mathrm{def}}{=} & down.\mathrm{C1} + up.\mathrm{C3} + \overline{nf}.\mathrm{C2} + \overline{ne}.\mathrm{C2} \\
\mathrm{C3} & \stackrel{\mathrm{def}}{=} & down.\mathrm{C2} + up.\overline{err}.\mathrm{Nil} + \overline{f}.\mathrm{C3} + \overline{ne}.\mathrm{C3}
\end{array}
$$

Table 6.4: Specification for FIFO CSM Controller

6.4 Application Specific Invariant Properties

Application specific invariant properties are dependent on the structure and behavior of a circuit. There are three main classes of specific invariant properties which must be tested for complete verification.

1. Behavioral
2. Operational safety
3. Conformance

6.4.1 Behavioral Proofs

Before embarking on the implementation of a process, the specification should be tested for its behavioral requirements. The interface specification for a circuit typically dictates these requirements. Behavioral testing is also required when a complex operation has been abstracted out of a higher level specification. Behavioral proofs of CCS processes are typically made using modal-μ calculus. A good treatment of using modal-μ to test the behavior of asynchronous circuits can be found in [Liu92].

6.4.2 Logical Conformance

Conformance proves that an implementation or more detailed specification contains all of the necessary behaviors and none of the illegal behaviors of the specification. This is probably the most critical verification step. Conformance is automated in Analyze when the sorts of the specification and implementation are equivalent.

Unfortunately conformance alone may not be sufficient to guarantee an operational circuit, even when the top level behavior has been correctly specified. Certain assumptions of correct component interfacing and utilization may need to be explicitly tested because (1) shared components are accessed in a distributed fashion, or (2) the safe usage details are not contained in higher levels of the specification. The temporal logics of the following section fulfills these verification requirements.

6.4.3 Operational Safety Proofs

Operational conditions, (often referred to as *safety* constraints), must be met in the design of a circuit. These safety conditions are typically implementation dependent. For instance, when a shared bus is used, operational safety conditions require mutually exclusive access. Three types of safety conditions in asynchronous circuits will be discussed here. Modal-μ formulae are presented along with a proof technique that can verify safe operation.

Access Violations

Access violations occur when a component is improperly used. For instance, the FIFO controller of Table 6.4 contains an up/down counter with a legal range of values equivalent to the number of slots in the FIFO buffer. Illegal counter access occurs when the counter holds the value zero and a request is made to count down resulting in FIFO underflow.

This type of illegal access violation can be proven by adding a test signal to the specifications. The signal $\overline{err}$ (or $\perp$) is placed as a response to unsafe access in Table 6.4. Applying the process under question (CSM) to the NOTPOSS formula of Definition 44 with the error label ($\overline{err}$) is sufficient to prove that the controller correctly accesses the counter so that FIFO over and underflow cannot occur.

Definition 44 *Signal α cannot occur in process P if P satisfies the following formula:*

$$\text{NOTPOSS } \alpha \stackrel{\text{def}}{=} \Box[\alpha]\text{F}$$

Since CSM $\models \Box[\overline{err}]$F the safety condition holds and the $\overline{err}$ signal can never occur. Explicitly testing for most types of illegal access is not necessary when using Analyze. The NOTPOSS test in this circuit is redundant if the $down.\overline{err}.$Nil actions are removed from C0 and $up.\overline{err}.$Nil is removed from C3. Analyze can then automatically detect when the E or W interfaces attempt to count down or up in an illegal state.

Mutual Exclusion

Safety conditions are violated when illegal access occurs to a restricted or shared process. This safety condition can typically be verified by proving mutually exclusive access to the process. The modal formula of Definition 45 verifies that the two signal arguments do not have mutually exclusive transitions – there is some reachable state where both α and β can transition. This equation has a dual in Definition 46 that proves that for all states in the system the two signal arguments enjoy the mutually exclusive signal transition property.

Definition 45 *Signals α and β have* ***concurrent transitions*** *in process $P \in \mathcal{P}$ if P satisfies the following formula:*

$$\text{CONCURRENT } \alpha\ \beta \stackrel{\text{def}}{=} \Diamond(\langle\alpha\rangle\text{T} \wedge \langle\beta\rangle\text{T})$$

Definition 46 *Transitions for signals α and β are* ***mutually exclusive*** *in process $P \in \mathcal{P}$ if P satisfies the following formula:*

$$\text{MUTEX2 } \alpha\ \beta \stackrel{\text{def}}{=} \Box([\alpha]\text{F} \vee [\beta]\text{F})$$

Refer to the distributed arbiter definition of Table 6.2 on Page 157. When a token arrives at a node the controller must either handshake with the module interface or pass the token on to an adjacent node in a mutually exclusive fashion. Applying the formula DINODE $\models$ MUTEX2 $\overline{grant}$ $\overline{tout}$ proves that the $\overline{grant}$ signal is mutually exclusive with the $\overline{tout}$ signal. The same invariant holds between the *done* and $\overline{tout}$ signals. Data input and output on the bidirectional link of the CSM circuit of Table 6.4 on Page 159 can likewise be proven mutually exclusive.

The above formulae have a drawback in that they only assure that the two signals have mutually exclusive *transitions*. This is sufficient for transition (or 2-cycle) signaling protocols, but not for four-cycle protocols, where there must be a mutually exclusive *region* between the two signals. The formula in Definition 47 assures that the two signals are mutually exclusive between *pairs* of transitions. This is formula must be used when a mutually exclusive region is required, as is the case with four-cycle protocols.

Definition 47 *The signals α and β define a* ***mutually exclusive region*** *in process $P \in \mathcal{P}$ if P satisfies the following formula:*

$$\text{MUTEX4} \quad \alpha\ \beta \stackrel{\text{def}}{=} max(M4X.[\alpha]M4A \wedge [\beta]M4B \wedge [-\alpha,\beta]M4X)$$

with

$$max(M4A.[\beta]\mathrm{F} \wedge [-\alpha]M4A \wedge [\alpha]M4X)$$

$$max(M4B.[\alpha]\mathrm{F} \wedge [-\beta]M4B \wedge [\beta]M4X)$$

A specification of the analog *mutual exclusion element* (or **ME**) commonly used in asynchronous design is shown in Table 6.5. The above macros can be applied to this definition, so MESpec $\models$ CONCURRENT *r1 r2* and MESpec $\models$ MUTEX4 $\overline{a1}$ $\overline{a2}$

proves that the inputs are concurrent, and that the outputs have a four-cycle mutually exclusive region.

$$\begin{array}{lll}
\text{MEifc} & \stackrel{\text{def}}{=} & r.g.\overline{a}.r.\overline{a}.\overline{p}.\text{MEifc} \\
\text{MESem} & \stackrel{\text{def}}{=} & \overline{g}.p.\text{MESem} \\
\text{MESpec} & \stackrel{\text{def}}{=} & (\text{MEifc}[r1/r, a1/a] \ \mid\ \text{MEifc}[r2/r, a2/a] \ \mid\ \text{MESem})\backslash\{g, p\}
\end{array}$$

Table 6.5: Mutual Exclusion Element Specification

Handshake Protocol

The final type application specific invariant checks for violations of the asynchronous handshake protocol, either due to a delayed or improper response, or a hazard. The following two equations are the basis for strong and weak forms of the handshake protocol verification. Testing both two and four cycle handshaking is identical because the order of signal transitions remains the same in either method. Hence a single equation can be used for both protocols. The formulae in Definitions 48 and 49 verify the handshake protocol.

Definition 48 *A process obeys the **handshake** protocol for the signal arguments α and β when the following modal formula is satisfied:*

$$\begin{aligned}
\text{HANDSHAKE}\quad \alpha\ \beta \stackrel{\text{def}}{=}\ & max(X.[\beta]\text{F} \wedge [-\alpha]X \wedge \Diamond\langle\alpha\rangle\text{T} \\
& \wedge [\alpha]max(Y.[\alpha]\text{F} \wedge [-\beta]Y \wedge [\beta]X \wedge \Diamond\langle\beta\rangle\text{T}))
\end{aligned}$$

Definition 49 *A process obeys the **strong handshake** protocol for the signal arguments α and β when the following modal formula is satisfied:*

$$\begin{aligned}
\text{SHANDSHAKE}\quad \alpha\ \beta \stackrel{\text{def}}{=}\ & max(X.[\beta]\text{F} \wedge [-\alpha]X \wedge \text{EV}\langle\alpha\rangle\text{T} \\
& \wedge [\alpha]max(Y.[\alpha]\text{F} \wedge [-\beta]Y \wedge [\beta]X \wedge \text{EV}\langle\beta\rangle\text{T}))
\end{aligned}$$

Examining these equations shows that they are defined using mutual recursion. Assume that *req* is substituted for α and $\overline{ack}$ for β in Definition 48. This formula requires that in *every* state of the system the $\overline{ack}$ signal cannot occur until a *req* signal occurs, and there must be a path from every state that allows the *req* signal to occur. After the *req* occurs, the same test is applied to the $\overline{ack}$ signal, disallowing any *req* signals until the acknowledgment has occurred. Note in particular that if a process has a path where no acknowledgment can be made for a request (or vice versa), the handshake protocol is incorrect, and this formula will not be satisfied. The strong formula of Definition 49 is similar to the weak formula, except that the appropriate handshake response must occur on *every* path.

These two definitions strengthen the CYCLE definition found in [LABS93] that can be vacuously true when the appropriate response is not possible. For example, CYCLE is satisfied by the Nil process, the process of Figure 6.1, and Table 6.3 whereas none of these are satisfied by the HANDSHAKE and SHANDSHAKE macros. Such vacuously true results imply that the nonfunctional circuits are correct.

The handshaking formulae presented in this section can be strengthened further. Handshake signals, once offered, cannot be retracted. Hence handshake signals must be **persistent** [Mil65]. Definition 50 tests persistence by verifying that for all states where an α action is possible, that action must remain possible until taken.

Definition 50 *Process $P \in \mathcal{P}$ is* ***persistent*** *if $\forall\ \alpha \in \mathcal{L}(P)$ the following formula is satisfied*

$$\text{PERSISTENT } \alpha \stackrel{\text{def}}{=} \Box(\langle\alpha\rangle \mathrm{T} \Rightarrow max(X.\langle\alpha\rangle\mathrm{T} \wedge [-\alpha]X))$$

The persistence property must be tested separately when using the handshaking definitions in this section. Unfortunately the persistency formula from Definition 50 cannot be applied directly to an implementation agent. Loose specifications permit a large set of "don't care" states in an implementation agent that typically will not satisfy Definition 50, invalidating the results. However, this definition can be applied to verify the persistence of specifications.

The transfer of bundled data back and forth between two components is typically integrated into the handshake protocol. The handshake formulae can be expanded to include a trio of signals, for the request, data transfer, and acknowledge, as is done in Definitions 51 and 52.

Definition 51 *A process $P \in \mathcal{P}$ obeys the 2-cycle* ***bundled data handshake*** *protocol for signal arguments $\alpha, \beta, \gamma \in \mathcal{L}(P)$ when the following modal formula is satisfied:*

$$\mathrm{BDHS} \quad \alpha\ \beta\ \gamma \ \stackrel{\mathrm{def}}{=}\ max(BDHSX.[\beta,\gamma]\mathrm{F} \wedge [\alpha]BDHSA \wedge \Diamond\langle\alpha\rangle\mathrm{T} \wedge [-\alpha]BDHSX)$$

with

$$max(BDHSA.[\alpha,\gamma]\mathrm{F} \wedge [\beta]BDHSB \wedge \Diamond\langle\beta\rangle\mathrm{T} \wedge [-\beta]BDHSA)$$

$$max(BDHSB.[\alpha,\beta]\mathrm{F} \wedge [\gamma]BDHSX \wedge \Diamond\langle\gamma\rangle\mathrm{T} \wedge [-\gamma]BDHSB)$$

Definition 52 *A process $P \in \mathcal{P}$ obeys the 2-cycle* ***strong bundled data handshake*** *protocol for the signals $\alpha, \beta, \gamma \in \mathcal{L}(P)$ when the following formula is satisfied:*

$$\mathrm{SBDHS} \quad \alpha\ \beta\ \gamma \ \stackrel{\mathrm{def}}{=}$$

$$max(SBDHSX.[\beta,\gamma]\mathrm{F} \wedge [\alpha]SBDHSA \wedge \mathrm{EV}\langle\alpha\rangle\mathrm{T} \wedge [-\alpha]SBDHSX)$$

with

$$max(SBDHSA.[\alpha,\gamma]\mathrm{F} \wedge [\beta]SBDHSB \wedge \mathrm{EV}\langle\beta\rangle\mathrm{T} \wedge [-\beta]SBDHSA)$$

$$max(SBDHSB.[\alpha,\beta]\mathrm{F} \wedge [\gamma]SBDHSX \wedge \mathrm{EV}\langle\gamma\rangle\mathrm{T} \wedge [-\gamma]SBDHSB)$$

Definitions 51 and 52 can be modified to verify various types of four-cycle handshake protocols.

The specification of the FIFO storage management controller of Table 6.4 contains data transfers labeled as din and $\overline{dout}$. These transfers are controlled by the $cr/\overline{ca}$ and $wr/\overline{wa}$ request acknowledge pairs. Since the handshake protocols CSM $\models$ HANDSHAKE cr $\overline{ca}$ and CSM $\models$ SHANDSHAKE cr $\overline{ca}$ and the persistency properties CSM $\models$ PERSISTENT cr and CSM $\models$ PERSISTENT $\overline{ca}$ can be verified, the read interface of the FIFO obeys the correct bundled data protocol. The same properties hold when testing the write interface.

6.5 Conformance Applications

The modal-μ calculus, used extensively for application specific invariant verifications in Section 6.4, can be used to verify circuits. For example, the verification of a C-element implementation can be carried out with the modal-μ formulae in a speed-independent (shown in Table 6.6) and burst-mode (Table 6.7) fashion. The circuit is not being verified against the specification per se, but against a set of formulae that are constructed to model the critical behavioral aspects of the specification. In that sense the modal formulae themselves specify the desired behavior.

$$
\begin{array}{llll}
\mathrm{max}(\mathrm{SIC0}.\langle\langle a\rangle\rangle\mathrm{T} \wedge [[a]]\mathrm{SIC1} \wedge & \langle\langle b\rangle\rangle\mathrm{T} \wedge [[b]]\mathrm{SIC2} & \wedge\, [[\overline{c}]]\mathrm{F}) & \\
\mathrm{max}(\mathrm{SIC1}. & \langle\langle b\rangle\rangle\mathrm{T} \wedge [[b]]\mathrm{SIC3} & \wedge\, [[\overline{c}]]\mathrm{F}) & \\
\mathrm{max}(\mathrm{SIC2}.\langle\langle a\rangle\rangle\mathrm{T} \wedge [[a]]\mathrm{SIC3} & & \wedge\, [[\overline{c}]]\mathrm{F}) & \\
\mathrm{max}(\mathrm{SIC3}. & & \langle\langle \overline{c}\rangle\rangle\mathrm{T} \wedge [[\overline{c}]]\mathrm{SIC0}) &
\end{array}
$$

Table 6.6: Modal-μ formulae for SI C-element Verification

$$
\begin{array}{llll}
\max(\mathrm{BMC0}.\langle a\rangle \mathrm{T} \wedge [[a]]\mathrm{BMC1} \wedge & \langle b\rangle \mathrm{T} \wedge [[b]]\mathrm{BMC2} & \wedge\, [[\overline{c}]]\mathrm{F}) \\
\max(\mathrm{BMC1}. & \langle b\rangle \mathrm{T} \wedge [[b]]\mathrm{BMC3} & \wedge\, [[\overline{c}]]\mathrm{F}) \\
\max(\mathrm{BMC2}.\langle a\rangle \mathrm{T} \wedge [[a]]\mathrm{BMC3} & & \wedge\, [[\overline{c}]]\mathrm{F}) \\
\max(\mathrm{BMC3}. \quad \langle\langle \overline{c}\rangle\rangle \mathrm{T} \wedge [[\overline{c}]](\min(\mathrm{BMC3X}.\mathrm{BMC0} \vee ([\tau]\mathrm{BMC3X} \wedge \langle\tau\rangle \mathrm{T})))) & &
\end{array}
$$

Table 6.7: Modal-μ formulae for Burst-mode C-element Verification

The formulae presented here are not as rigorous as verifications based on the conformance equations of Sections 5.5 and 5.6. However, the burst-mode formulae in Table 6.7 are more rigorous than the speed-independent formulae because they verify signal persistence. This is done with the strong modalities of the $\langle a\rangle T$ and $\langle b\rangle T$ transitions in formulae BMC0, BMC1, and BMC2. The strong modalities assure that the associated label is persistent because it must always be capable of making a transition in the states that satisfy the formulae. Persistence of burst-mode AFSMs is relatively easy to verify because the stability requirement walks the τ transitions after the output burst. The speed-independent formulae do not assure that persistency is retained in the circuits, although a more complex set of equations could be used which does verify this constraint. Since persistence is a necessary condition for an AFSM, the SI formulae by themselves are incomplete.

6.6 Performance of Analyze

Although it may be possible to automatically derive modal formulae from specifications that are sufficient to verify the behavior of an implementation, this method is fairly inefficient as will be shown with a simple example that is comparable with other applications.

```
bi C-ELEMENT
( ANDNB000[ab/c]                      \
 | ANDNB000[c/b, ac/c]                \
 | ANDNB000[b/a, c/b, bc/c]           \
 | ORNB0000[ab/a, ac/b, bc/c, c/d]    \
) \{ab, ac, bc}
```

Table 6.8: CCS Description of C-element Implementation

Table 6.8 shows the CCS definition of the AND-OR implementation of the C-element shown in Figure 3.6 on Page 63. The definitions of the AND and OR gates are library definitions which were not included for clarity. The gates are *nonblocking* definitions, indicated by the "NB" in the component name. Nonblocking gates allow inputs into an unstable device to change so long as the changes will not invalidate the pending output. For instance, as soon as one input into an OR gate asserts, the device becomes unstable until the output fires. Further input transitions are allowed without requiring the output to fire so long as at least one input remains asserted. The numbers '0' or '1' at the end of the definitions represent the initial voltage states of the inputs and output. Signal names in these devices start with a, b, and so on through the alphabet, with the last signal name being the output.

Note that this direct definition of the C-element will not parse correctly using the parallel composition of CCS because of the speed-independent "broadcast" nature of the interconnection in the a and b inputs and $\overline{c}$ output. Hence analysis of this device in the Concurrency Workbench requires the circuit to be parsed in Analyze first. Once the correct definition is loaded into the CWB from Analyze, Table 6.9 shows the performance of the verifications in the CWB and Analyze.

Delay Model	**Analyze: Conformance**	**CWB: Modal-μ**
Speed-independent	0.2 sec (8 errors)	2 sec (False)
Burst-mode	0.2 sec (True)	9 sec (True)

Table 6.9: Performance of Analyze and Modal-μ Verifications

Verifications in Analyze are much more efficient than those using modal-μ on the workbench as can be seen from Table 6.9. The difference is even more significant when one realizes that the workbench takes 193 seconds just to parse the C-element description, a number which is not included in the table! Analyze requires 0.6 seconds to parse the C-element description that is subsequently loaded into the CWB. Due to the compositional nature of the Analyze conformance verifications, this is three times as long as the combined parsing and verification time of 0.2 seconds.

Note that the run time for Analyze is the same whether the verification is true or false, but there is a significant difference in the run times using modal-μ calculus. In the CWB, as soon as a formula will not satisfy the process, false is returned. When conformance does not hold, the current version of the Analyze prototype continues to evaluate the entire state space and produces a list of the failures. This points out the circuit failures, but requires a full run time.

The results using Analyze are also more accurate than the modal-μ formulae. For example, the speed-independent equations do not ensure that the implementation is persistent as the input signals need not remain enabled. Further, the modal-μ expressions are not automatically generated at this time, although they could be automatically produced from a specification with the proper tools.

There are several arguments against using modal formulae for specifications given the current software technology. Evaluation of modal equations can be quite time consuming with the currently available tools. For example, 401 CPU seconds were required to satisfy process CSM $\models$ SHANDSHAKE $cr\ \overline{ca}$ on the Concurrency Workbench. Typically, adding fixed point constraints results in slower satisfaction results (which is one reason why the persistence property is not included in these macros).

Modal formulae are more difficult to understand than CCS and other representations. However, there appear to be a few standard tests and styles for those tests which can be made quite readable [Liu92, LABS93]. Therefore, constructing tailored macros and formulae shouldn't be overly difficult given a library of case studies and some basic intuition. "Object oriented" representations in temporal logics are also less obvious than with CCS specifications. Hierarchy and structure is not modeled well with these formulae, so synthesis and decompositions are problematic. Finally, the problems to be presented in Section 7.2 must be addressed to accurately model hardware, as even in the above example the modal calculus could only be used after circuit description was parsed by Analyze.

6.7 Summary

Temporal logics are an attractive means of testing invariant behaviors of a system. Formulae were presented in this chapter further support the verification tool Analyze by verifying invariant properties of specifications and implementations that cannot be tested hierarchically or are necessary properties of a specification. This chapter continues with the approach in [Liu92, LABS93] where there is a certain class of properties, such as liveness, which must hold for all specifications and implementations. Once the basic properties hold, there are a number of additional tests which may be required to verify correct specifications.

The formulae presented in this thesis make the following contributions.

1. A new definition for the "liveness" of circuits was developed for labeled transition systems. This definition is also the dual of a new "deadlock" definition. Conceptually a circuit is live if from every state each action can be exercised. Conversely, a deadlock exists in the circuit if a state can be reached where some actions can never occur.

2. A set of modal formulae were created or strengthened that raise the level of abstraction for design testing. These formulae can be expressed as macros and applied to specifications for satisfaction. Most of the formulae are dependent on the sort of the processes, while others also require knowledge of the access assumptions of various modules.

   There are only a few types of tests that are necessary for verification of most circuits. Most of the formulae have been developed for both transition and level based asynchronous handshake, as well as bundled data protocols. These formulae, applied in concert with Logic Conformance, are sufficient to prove a specification will be functional.

Temporal logics have two main drawbacks. First they are quite inefficient on the currently available software tools. Second, many of these formulae are only valid when applied to specifications because the unreachable states of an implementation usually invalidate the results.

Chapter 7

Synthesis and Verification using Analyze

"If every tool, when ordered, or even of its own accord, could do the work that befits it ... then there would be no need either of apprentices for the master workers or of slaves for the lords."

Aristotle

Verification consists of using formal models to prove that the properties of one specification are equivalent to the properties of another. Additional flexibility and simplicity of implementations can be achieved if the specifications are "loose", which can be achieved by including information regarding behaviors which are <u>necessary</u>, <u>illegal</u>, and <u>irrelevant</u>. These properties can be expressed in a number of ways in CCS, including labeled transition system semantics and partial orders such as logic conformance, or temporal logics as was presented in the previous chapter.

The assumption that one of the specifications describes a *physical circuit* rather than an interface behavior requires modifications to the standard CCS model. The abstractions must be useful in that they simplify reasoning about the circuit, but they must also be accurate or no benefit will accrue from the verification process. Other techniques such as CCS with priority choice [Cam91] and TCCS [MT90] were investigated. None of these extensions to CCS resolved all of the issues, and they required specifications that are much more difficult to understand. The extensions applied to CCS as part of this thesis are accurate and simple to use.

This section describes a prototype software tool called **Analyze** that was developed as part of this thesis. Formal methods used in this tool can make at least three major contributions toward aiding engineers design circuit implementations.

1. Formally prove properties of a specification so it is well characterized before attempting an implementation.
2. Formally verify equivalence between an implementation and specification to assure a faithful implementation.
3. Aid the designer in structured hierarchical design practices to achieve verified top-down implementations.

These three issues, and their embodiment in a prototype tool, are covered in this chapter. Impediments to reasoning about certain hardware implementation levels and asynchronous hazard models in CCS are discussed. Changes to the restriction and parallel composition operators are implemented that empower CCS greater flexibility for reasoning about asynchronous hardware. The relation of minimization to verification is described, and an efficient minimization algorithm is presented. The necessity of observability of agent expansion is presented in a discussion of "computation interference". Detecting interference is required for accurate verifications, and is used to drive high level synthesis. The verification of correctly constructed burst-mode specifications developed. The usage of the software tool is then introduced and a small example session is presented. Finally the high level synthesis process is described.

7.1 The Concurrency Workbench

The Concurrency Workbench, or **CWB**, is a software tool which implements CCS semantics as described in Milner [Mol91]. It also includes many equivalences, some partial orders, and a set of formal logics which have been integrated into the CCS labeled transition system.

The CWB is extremely useful for reasoning about asynchronous circuits. Ying Liu's thesis is a good initial reference on how the CWB can be applied to reasoning about asynchronous circuits [Liu92]. The CWB is also vital to the synthesis and verification approach presented in this thesis as general formal logic formulae and fixed point calculations cannot be entered into Analyze.

7.2 Problems with CCS and the Workbench

CCS is a general model which can be readily applied towards verification of coarse models and protocols, to which it has been applied with great success [Bre90, Bru92, Par87]. Notational simplicity and succinctness permit this higher level of abstract modeling in a hierarchical manner. However, it also imposes some limitations as there is no inductive proof system as exists in higher order logics which simplify the verification of replicated components such as RAM cells, latches, and so forth. Unfortunately the model also imposes some limitations as to the accuracy and capability of modeling hardware and the various asynchronous delay models.

There are two basic approaches to this problem. The first is to live within the limitations of CCS while modeling hardware [LBP94]. This approach allows one to utilize the CWB for verifications. This approach has a number of drawbacks.

First there must be a disciplined use of the syntax to assure that the verifications do not model inappropriate structure. For example, judicious use of handshake communication is required so that choice is not being modeled. Such requirements are not automatically checked in the CWB and this weakens the value of verifications with this method. Further, the lowest level that can be modeled this way is at the coarse block or protocol level. Modeling AND gates and any asynchronous hazard model other than the delay-insensitive is not possible.

The second approach is to pinpoint the areas which prevent the accurate modeling of hardware gates and other asynchronous delay models, and modify CCS to support these requirements. This is the approach taken in this thesis and implemented in Analyze. Care has been taken to only modify and extend aspects necessary to support such features. The extensions in this thesis keep the look and feel as similar to CCS as possible; the modifications are nearly invisible to the user. Both standard CCS transitional semantics and those necessary for modeling hardware are present in Analyze, permitting it to accurately reason about hardware as well as retain the clarity of CCS specifications.

This second approach has also been taken by Milne in the development of Circal, a modification of CCS for verifying and modeling hardware [Mil85, MM92]. Circal is intended as a method for specifying general purpose circuit structures, and introduces several new syntactic symbols with new semantics. It is intended for synchronous systems, but recent research has applied this language to asynchronous circuits [Bai94]. Unfortunately, as with TCCS and other extensions to pure CCS, the representation of asynchronous circuits becomes cumbersome and awkward and it may not easily represent all the hazard models.

The remainder of this section points out the shortcomings that prevent the verification and modeling of gates and the more useful asynchronous hazard models in CCS. This thesis assumes that when reasoning about hardware components, the communication between names and conames models a physical communication link in the circuitry (typically an aluminum wire). Unless otherwise noted, the unbounded delay model is used with speed-independent verification, and equivalences are based on branching time bisimulation.

Input and Output Recognition

CCS assigns no meaning to the set of labels $\mathcal{A}$ and colabels $\overline{\mathcal{A}}$ other than for synchronization purposes. However, physical circuits have two main terminal types – inputs and outputs – with critical differences. This difference goes beyond the complementary naming convention of CCS as can be seen by examining the equations for conformance, and static invariant checks of Section 6.3. This thesis assumes that names $\mathcal{A}$ always map to inputs, and conames $\overline{\mathcal{A}}$ always map to outputs.

Verification

Equivalences overly restrict the freedom of design choice. The logic conformance definitions of Section 5.6 are partial orders allowing standard CCS agent descriptions to be used as loose specifications. The necessary set of actions includes all inputs and outputs exactly as they can occur in a transition diagram from the specification. The behaviors that may *not* occur include all output actions from any of the states in the transition diagram of the specification which do not explicitly exist. The set of irrelevant actions include all others, as the specification guarantees that they are non-reachable.

Handshake Synchronization

The communication primitive in CCS is analogous to the handshake primitive of asynchronous circuits as both a label and colabel must be offered before a communication can occur. This handshake communication primitive in CCS can greatly simplify specifications. However, the handshaking synchronization can result in failures when modeling hardware because the synchronization rules of CCS will not allow restricted communication signals from "firing" until both the label and the colabel are offered. A physical circuit can wait arbitrarily long for an input to occur, but when an output is offered by a circuit, the physical wire is immediately driven to the new voltage level. If this two-way agreement is necessary in a physical circuit, it must be modeled by a pair of wires.

Handshaking synchronization also hides many implementation errors if specification mirroring is used for verifications as discussed in Chapter 5. The failure caused by mirroring and the the handshake primitive is modeled as *computation interference*, which will be discussed further in Section 7.3.

Nondeterministic Choice Operator

The CCS choice operator '+' is nondeterministic. This can be exploited in specifications to simplify or clarify behaviors. However, current digital state of the art does not permit nondeterministic AFSM implementations. Nondeterministic behavior is attained through the use of analog ME elements, and even so choice is not entirely nondeterministic. The inability to reason about fairness in a CCS specification is primarily due to the nondeterministic nature of the '+' operator.

CCS choice typically models different legal trajectories that a circuit can take when parallelism or an externally determined (circuit environment) choice is possible.

Choice is necessary and can be modeled in a deterministic fashion. The burst-mode requirements discussed later in this chapter assure that choice is used in a fair, deterministic fashion in AFSMs.

Interconnection Modeling

Hardware components that are composed in parallel are considered interconnected by a wire on all matching labels and colabels. The primitive interconnection in CCS is a one-to-one event structure as can be seen by the Com_3 rule of Figure 1.3. The one-to-one mapping must be possible to reason about DI and QDI delay models. CCS rules also allow numerous illegal communication structures and primitives for the DI and QDI models. For example, the CCS agent $\mathrm{E9} \stackrel{\text{def}}{=} (\overline{a}.\mathrm{A1} \mid a.\mathrm{A2} \mid a.\mathrm{A3})\backslash\{a\}$ results in a $\xrightarrow{\tau}$ transition where $\overline{a}$ handshakes nondeterministically with either of the two a signals, evolving into $(\mathrm{A1} \mid a.\mathrm{A2} \mid \mathrm{A3})$ or $(\mathrm{A1} \mid \mathrm{A2} \mid a.\mathrm{A3})$. This competition for communication between sets of names and conames resulting in nondeterministic choice can greatly simplify a specification, but cannot be implemented with a wire!

Parallel Composition

Broadcast communication amongst a set of composed parallel agents, necessary for the QDI, SI, and burst-mode delay models, cannot be modeled in CCS. This is a serious shortcoming as there are very few valid delay-insensitive circuit implementations. The isochronous fork assumption and speed-independent assumptions, which permit realizable circuits, are based on a broadcast communication primitive.

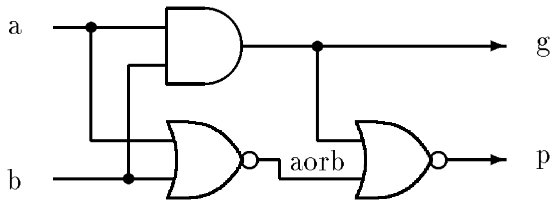


Figure 7.1: Manchester Carry Chain

```
bi MANCHESTER-CARRY
( AND [g/c]                  \
 | NOR [aorb/c]              \
 | NOR [g/a, aorb/b, p/c]    \
) \{aorb}
```

Table 7.1: CCS Description of Manchester Carry Chain

7.2.1 Parallel Conjunction

The Com transition rules in Figure 1.3 on Page 19 do not correctly model the behavior of interconnected circuits operating in parallel. Whenever inputs or outputs of a circuit contain the same name or coname, hardware convention assumes that the circuit has been connected by a communication channel. Assume the simple circuit of Figure 7.1, specified by the CCS agent in Table 7.1. The AND and NOR agents are library component specifications having inputs of a and b, and an output of $\overline{c}$. Note from the diagram and table that there are four labels which can communicate via the '|' (Com) transition – a, b, g and $aorb$. Only the $aorb$ signal is restricted to the local domain – the other signal needs to communicate internally as well as with external agents.

There are three different actions that this circuit can make on wire g according to pure CCS using the parallel communication rule Com. The g signal can communicate *independently* with the environment as an input (the g transition into the NOR gate) or an output ($\overline{g}$ out of the AND gate), or an internal communication between the AND and NOR gate can occur resulting in a τ transition with no interaction with the environment. When restricted with the Res transition rule, the first two actions are eliminated.

There is no possibility of the physical AND or NOR gates communicating independently with the environment as the Com_1 and Com_2 transition rules allow. In particular, if the AND gate is the only gate driving the g wire, then an independent input action on the g signal should not occur in absence of crosstalk and other circuit failures. All external communications from this module on wire g should also be outputs ($\overline{g}$), and will also communicate with the NOR gate. The same sort of reasoning exists for the SI or burst-mode processing of the inputs a and b, which must communicate jointly when being driven as an input to the circuit. The required transitional behavior for modeling this simple circuit under a speed-independent or burst-mode model is not possible under the CCS Com transition rules – hence new communication transition rules are necessary. Standard CCS syntax would require adding a FORK component on each of the wires a, b, and g rather than as a single wire with three connections. This represents the delay-insensitive hazard model.

New communication transition rules must allow communication between an arbitrary number of agents to model isochronous forks, speed-independent, and burst-mode hazard models. This requires a *conjunctive* parallel communication operator, that allows broadcast-like communication.

There are two methods for formulating broadcast, or conjunctive synchronization. One method is to define a reserved atomic action in which many agents can participate. This is the definition used in CCS, allowing dual-agent communications via the atomic action τ by synchronizing l and $\overline{l}$ as an inseparable action. A second approach is to define every action to be composed of a finite set of inseparable atomic actions. As the set of actions are inseparable, they can be considered a single event that is indivisible in time. Circal takes the latter approach where every action is defined in terms of finite sets of labels, and where the combinators dictate which action sets can be synchronized [Mil85].

The desired conjunctive semantics are similar to that of Hoare's $P \parallel Q$ combinator in CSP, which depends on the explicitly supplied sorts of P and Q, which Hoare calls the *alphabet* of the agents [Hoa85]. This is an alternative method of parallel composition which could have been selected as the CCS parallel composition operator, but it is more difficult to implement because the dependency on sorts effectively results in an infinite family of operators. Hoare's '$\|$' combinator also does not use the notion of names and conames (inputs and outputs) for communication – any set of identical characters in the alphabet can communicate. The final significant difference is that it is more natural to utilize a *hiding* operator (one which changes an externally observable action into an invisible action) with the '$\|$' combinator, rather than the restriction operator of CCS (which prevents an action from occurring). The dependency on sorts can "restrict" the undesirable independent behaviors from occurring when using conjunctive communication.

A new transition rule called **Conjunction** ('$|_c$') is introduced that forces synchronization amongst a set of agents with similar sort labels. The transition rules are defined by inference, and are similar to the CSP operator. The dependency of one action being from the set of names $\mathcal{A}$ and the other from the set of conames

$\overline{\mathcal{A}}$ as in the Com transition rule is not required, but the input-output sense of the transition is preserved, which is not possible under the Com rules.

$$\textbf{Conj}_1 \quad \frac{E \xrightarrow{\alpha} E'}{E \mid_c F \xrightarrow{\alpha} E' \mid_c F} \quad (\alpha, \overline{\alpha} \notin \mathcal{L}(F))$$

$$\textbf{Conj}_2 \quad \frac{F \xrightarrow{\alpha} F'}{E \mid_c F \xrightarrow{\alpha} E \mid_c F'} \quad (\alpha, \overline{\alpha} \notin \mathcal{L}(E))$$

$$\textbf{Conj}_3 \quad \frac{E \xrightarrow{\alpha} E' \qquad F \xrightarrow{\alpha} F'}{E \mid_c F \xrightarrow{\alpha} E' \mid_c F'} \quad (\alpha \in \mathcal{L}(E) \cap \mathcal{L}(F))$$

$$\textbf{Conj}_4 \quad \frac{E \xrightarrow{l} E' \qquad F \xrightarrow{\overline{l}} F'}{E \mid_c F \xrightarrow{\overline{l}} E' \mid_c F'} \quad (l \in \mathcal{L}(E) \wedge \overline{l} \in \mathcal{L}(F))$$

$$\textbf{Conj}_5 \quad \frac{E \xrightarrow{\overline{l}} E' \qquad F \xrightarrow{l} F'}{E \mid_c F \xrightarrow{\overline{l}} E' \mid_c F'} \quad (\overline{l} \in \mathcal{L}(E) \wedge l \in \mathcal{L}(F))$$

Table 7.2: Parallel Conjunction Transition Rules

Note that the side conditions to Conj_1 and Conj_2 in Table 7.2 have provided the desired "restriction" operation for interconnected hardware components. They do not allow agents to evolve independently when composed in parallel if a matching name or coname appears in a parallel agent. Conj_3, Conj_4, and Conj_5 create the conjunctive communication. Conj_3 is applied when all labels are either inputs or outputs. Conj_4, and Conj_5 are applied when there is a mixture of input or output labels, and the resulting transition will use an output label.

Correctly modeling hierarchical agents and observational equivalence with τ transitions must be possible when the names and/or conames are not externally accessible. The operation of creating τ transitions with conjunctive communication is one of "localizing" the interconnection between the parallel agents. Removing access to the signal from outside the current block is achieved by removing the name from the sort while allowing the effect of the action to proceed unconstrained. This new *hiding* operation of Table 7.3 replaces the Res operator when used with Conj.

Hide$_1$	$\dfrac{E \xrightarrow{\alpha} E'}{E \backslash L \xrightarrow{\alpha} E' \backslash L}$	$(\alpha, \overline{\alpha} \notin L)$
Hide$_2$	$\dfrac{E \xrightarrow{\alpha} E'}{E \backslash L \xrightarrow{\tau} E' \backslash L}$	$(\alpha \in L \vee \overline{\alpha} \in L)$

Table 7.3: Hiding Transition Rules

When hiding, L is the set of labels that are being hidden from outside the agent. Intuitively, where restriction disallows the occurrence of an action, hiding disallows the action from interacting with the environment – localizing it to the current agent.

The primary advantage of Conj over Com is the ability to synchronize multiple agents on a single event, an operation that is necessary for speed-independent, burst-mode and isochronous fork evaluation. For instance, the agents $(P \mid_c Q \mid_c R)$ with sorts $\mathcal{L}(P) = K$, $\mathcal{L}(Q) = L$, and $\mathcal{L}(R) = M$ will result in a three-agent synchronization when the signal l as either an input or output is in the label sets K, L, and M. The signals a and b in the example from Figure 7.1 and Table 7.1 synchronize the AND and NOR gates with signals being driven from the environment, and the $\overline{g}$ signal synchronizes the AND and the other NOR gate and drives the $\overline{g}$ signal to the environment. The C-element of Figure 3.6 on Page 63 works the same way; two AND gates, one NOR, and the environment all synchronize on the output $\overline{c}$. This new combinator has the effect of reducing the behavior of agents further than the Com transition rule. The rest of the calculus and proof system remains the same.

7.2.2 Analyze Parsing

To simplify the specification of agents, the parallel conjunction operator is textually specified with the same symbol, '$|_c$', as the standard CCS parallel composition operator. The evaluation mode of Analyze will determine whether the operator uses composition ('$|$') or conjunctive ('$|_c$') transition rules.

The binding power of the CCS operators place Restriction and Relabeling as the tightest binding, followed by Prefix, Composition, and finally Summation. This binding order can result in problems with a circuit definition. For example, the definition $E10 \stackrel{\text{def}}{=} a.E10' \backslash \{a\}$ will result in an agent $E10$ having a sort that includes the name $a \in \mathcal{L}(E10)$. From a hardware perspective, this is an invalid sort, because the a wire will interact with the environment only outside of agent $E10'$, while within the agent events on the wire will not propagate outside the circuit. Therefore the current version of Analyze makes a slight modification to the binding precedence of CCS to simplify the language for hardware designers by binding Prefix tightest, followed by Restriction and Relabeling, Composition, and Summation. Hence $a \notin \mathcal{L}(E10)$ in the definition of agent $E10$ when $E10 \stackrel{\text{def}}{=} a.E10' \backslash \{a\}$. Parentheses should always be used to clarify any unobvious bindings. For compatibility with the CWB and CCS, future versions of the Analyze prototype will use the standard CCS binding order but will issue a warning on odd usage as listed above.

7.2.3 Circuit Connections

The general flexibility of CCS specifications permits a number of constructions that, although useful for specifications, result in errors in an implementation. The static analysis of Analyze will prove that the circuit specification does not violate physical properties of the circuit.

The following checks are applied when the conjunctive communication operator is used to ensure that the interconnections are physically correct. The sort of the parallel agents, required for conjunctive communication, is also required by these interconnection checks.

1. When all signals α in the conjunctive communication are inputs, $\alpha \in \mathcal{A}$, all of the agents evolve in parallel according to Conj_3 on a single input transition $\xrightarrow{\alpha}$.

   Under speed-independent or burst-mode analysis, multi-way input connections are assumed correct. A warning is printed out that this particular wire is interconnected using the isochronous fork assumption – hence the analysis is quasi delay-insensitive – when analyzing a circuit in the delay-insensitive mode. The designer must make sure that the isochronous fork is necessary for the implementation.

2. More than one conjunctive communication label is an output. This is an illegal circuit interconnection because two actively driven output signals cannot handshake. This is allowed by the Conj_3, Conj_4, and Conj_5 transition rules and is checked upon verification for correct interconnection. A diagnostic error message is printed and the verification aborted when this interconnection occurs.

The current version of the Analyze prototype cannot model tristate signals as distributed agents requiring a conjunction of output drivers.

3. There is one and only one label in the conjunctive communication which is an output. In this case, the rules Conj_4 and Conj_5 apply, and the signal $\overline{l}$ offered to the environment is an output, $\overline{l} \in \overline{\mathcal{A}}$.

   The current version of Analyze prints a warning when an unrestricted (unhidden) conjunction contains one or more input signals and one output label. This is a potential site for errors in a circuit as pointed out in Section 3.3.6 because non-local delay analysis of the implementation *and* its environment becomes necessary to assure a hazard free implementation of such an agent. Therefore this information must be passed on to the physical layout stage.

7.2.4 Restriction and Relabeling

The following static checks of Analyze are associated with the Restriction and Relabeling transition rules. Although restriction and relabeling are not necessarily associated with the composition or conjunction operators, good design practice will relabel and restrict signals as soon as possible to avoid confusion and computational complexity. When restrictions are associated directly with a set of parallel compositions, Analyze can create and analyze the specification compositionally, which can reduce the time and memory complexity by an order of magnitude and more.

1. Restricted signals can result in deadlock. For example, if the signal b is restricted from agent E11 where E11 $\stackrel{\mathrm{def}}{=}$ $a.b.$E11, a deadlock will occur after the a transition has occurred because the resulting behavior is equivalent to the

process a.Nil. The deadlock will not occur using the semantics of hiding, and E11 $\approx$ E12 where E12 $\stackrel{\text{def}}{=}$ a.E12. Hiding is a more natural use for hardware specifications as it allows actions to occur uninhibited by the environment, whereas there is really no way to *prevent* a signal occurrence as with the restriction semantics.

Sometimes a component will contain a redundant or optional input or output. Simply leaving the signal unconnected results in confusion for several reasons. First, it is not clear if the module has been incompletely interconnected or if the signal is to be ignored. Secondly, different behaviors arise if the signal is removed from environment interaction with restriction or hiding. The correct way to remove the signal from consideration is to first connect the unused signal to a "signal sink" and then restrict or hide the signal from the environment. A signal sink will accept an unbounded number of signal transitions, and is sometimes called a "block of wood" with a definition WOOD $\stackrel{\text{def}}{=}$ a.WOOD. This makes the semantics of composition and restriction or conjunction and hiding the same, and there is no longer any confusion about the completeness of the design. Therefore, whenever a signal is restricted or hidden and it does not communicate with another agent, a warning is issued by Analyze.

2. A warning is issued if any of the labels in a restriction (hiding) or relabeling set are not in the sort of the bound agent. This occurrence typically results in circuit failures caused by a typo or an incorrect label set.

3. Verification will usually fail if the sort of the specification and the implementation are different, because extra behaviors exist in one component that do

not in the other. When verifying an implementation against a specification, a warning is issued when the sorts are incompatible.

7.3 Computation Interference

Pure CCS implements compositional communication using *handshake synchronization*. If two agents are composed and restricted on α, then the internal τ transition can only occur when one agent offers an α transition and the other offers an $\overline{\alpha}$ transition. If only one of the two labels are offered, then the τ transition will not occur. This type of handshake synchronization is extremely useful for simplifying the specification of complex parallel processes [SABL93]. However, it does not model hardware well because this handshake synchronization will assume that an output will not be driven until it can be accepted by an input!

Definition 53 *Computation interference exists between composed agents if*

$\overline{\alpha}.P \mid Q$ *where* $\alpha \in \mathcal{L}(Q)$ *and* $Q \stackrel{\alpha}{\not\rightarrow}$

Computation interference exists in a state where an output can fire before its corresponding input agents are prepared to synchronize with the output. Any circuit output whose firing is disabled by the handshake communication of CCS should result in an error.

When an agent is prepared to accept an input signal a label $\alpha \in \mathcal{A}$ is offered. No interference occurs when the label is not matched by a colabel (output) from a parallel agent; the agent offering the input idly waits for the input to occur.

The compositional design of Analyze retains information about the parallel structure of agents and can detect when computation interference occurs in a circuit.

There are three types of computation interference which require different responses from the synthesis and verification procedures.

7.3.1 Interference in a Specification

High level specifications commonly contain synchronizations that contain computation interference. At this level, interference is not an error but directs the designer to synchronizations in the design that require further implementation detail. These synchronizations must be split into agents that implement the synchronization without interference. Directed hierarchical verification and refinement is partially based on the occurrence of computation interference, as discussed further in Section 7.6. This type of synchronization must not be disregarded in hierarchical decomposition by splitting it into two separate high-level specifications, or an unfaithful implementation may result.

7.3.2 Implementation Interference on an Output

Whenever computation interference occurs in processes modeling hardware components an unrecoverable error has occurred. The fault is typically the result of a hazard in the circuit, but may also be the result of the behavioral faults of an incorrect design. If the interference is caused by a design fault the circuit must be redesigned to remove the interference. If the interference is caused by a hazard in the circuit then the circuit must be redesigned, or this information must be passed on to the implementation phase so that the occurrence of the hazard can be avoided through layout engineering.

Without the ability to detect interference, hazards and behavioral faults can be hidden by the handshake synchronization of CCS. For example, the static 1 hazards revealed using speed-independent analysis of the C-element of Figure 3.6 in Section 3.3.5 can only be detected as computation interference. Without detecting interference, the unstable AND gate would block the output that also feeds back as an input until the AND gate stabilizes. This causes the OR gate to stabilize, hiding the hazard.

7.3.3 Implementation Interference on a Restricted Signal

If the implementation is a flat leaf cell, then computation interference on internal (restricted or hidden) signals is an error just as though it were an output. However, if a specification is hierarchical and the interference is caused by a hardware sub-agent, the error may be due to the unrestricted behavior of the sub-agent. Replacing the hardware sub-agent with its specification will remove the computation interference in a well designed circuit.

The crossing example in Section 7.5 uses a two level refinement because the two halves of the definition are identical. Conformance will point out interference violations in the "vehicle" specifications because the environmental restrictions have not been specified. Using the specification at that level, or a flat implementation, removes the interference.

7.4 Bisimulation and Minimization

Several aspects must be considered when designing a synthesis and verification system for asynchronous circuits, including the time and memory complexity of the algorithms, and how accurately the physical circuits are modeled. Trace based systems were the first to reason about asynchronous systems. These include systems based on CSP [Ebe88, Udd84] or variants of trace theory [Dil89]. The most appropriate equality for hardware verification should be chosen when significant advantages are apparent.

One of the features of CCS is the unique canonical representation of agents based on bisimulation semantics, achieved by merging all indistinguishable states. This merging, also called *minimization*, has a side effect of losing the structure of parallel and hierarchical agents in creating the canonical representation.

Minimization can be used to prove bisimilarity between agents when all of the states of the agents are minimized together. If the minimized states contain at least one state from each of the agents, then they are bisimilar. If the agents are not bisimilar, then all of the minimized states from each agent will be mutually exclusive of states from the other agent (except the special state Nil).

7.4.1 Minimization and Equivalences

Two processes are bisimilar when they are trace equivalent and determinate as shown in Proposition 7. Hence trace equivalence and minimization are sufficient to show bisimilarity amongst determinate processes. Further, two determinate processes that are trace conformant are also logic conformant.

Proposition 15 *If I and S are determinate, then $S \succeq_l I$ if $S \succeq_t I$.*

Proof Going from left to right (if $S \succeq_l I$ then $S \succeq_t I$) can be proven using the trace and logic conformance definitions of Definition 25 and 27. Going the other way, it is sufficient to show that

$$\mathcal{LC} \stackrel{\text{def}}{=} \{(S, I) : S \succeq_t I \text{ and } S, I \text{ are determinate}\}$$

is a logic conformance up to $\succeq_l$. Since all s-derivatives of a determinate agent are bisimilar from Definition 23, by Proposition 14 and Definition 27 it can be shown that the above relation holds. By Definitions 15 and 16 the s-derivatives of a determinate agent maps to $S \stackrel{\alpha_1}{\rightarrow} \stackrel{\alpha_2}{\rightarrow} \cdots \stackrel{\alpha_n}{\rightarrow} S'$. Since $\mathcal{LC}$ is a logic conformation, we also have $I \stackrel{\widehat{\alpha_1}}{\Rightarrow} \stackrel{\widehat{\alpha_2}}{\Rightarrow} \cdots \stackrel{\widehat{\alpha_3}}{\Rightarrow} I'$ and $(S, I) \in \mathcal{LC}$. $\square$

If the implementation or specification are *not* determinate then Logic Conformance is not implied by Trace Conformance. Typical verifications will require logic conformance, which can be calculated more efficiently than trace conformance for nondeterminate agents.

7.4.2 Minimization Algorithm

The worst case complexity of a set of parallel agents is the product of the state space of each of the agents. If the state space of the agents can be reduced, such as through minimization, then the complexity of the parallel composition can be greatly reduced as well. Minimization also has useful applications for bisimulation and conformance as shown in the previous section.

Fernandez implemented an efficient algorithm for minimizing agents using bisimulation [Fer90, PT87]. A different algorithm based on branching time bisimulation and used in Analyze is described here. The concept for minimization is to equate all states as if they were all bisimilar, and then separate any states that can be distinguished by their transitions. States with distinguishable transitions are separated out until no more distinguishing transitions can split states apart.

The minimization algorithm is calculated as follows:

1. First each state is marked with the set of $\stackrel{\widehat{\alpha}}{\Rightarrow}$ transitions that are possible from each state by walking all states. All possible $\stackrel{\widehat{\alpha}}{\Rightarrow}$ transitions from any state are fully specified by the state's transition set when there are no τ transitions from the state, and this set is recorded and stored. When a state contains τ transitions they are followed until a leaf node is reached where no τ transitions are possible, or until a state is reached that has already been completed or touched in this walk. The union of the set of possible transitions is returned and stored for the source state.

2. All states are initially placed into the same "bin". States that are not bisimilar from their transition sets are then split into different bin. For instance, all states that can only do an a transition are placed in one bin, all that can do an a and b transition are placed into another bin, etc.

   This split will result in a maximum of 2^n bins, where n is the number of labels in the agent's sort. As an example, assume that an agent has the sort $\{a, b, c\}$. Each bin is split by the property of having each transition label in it. Those states which have the transition go in one bin, those that do not into the another. Figure 7.2 conceptually shows how this split works. The input or output sense of each signal is considered significant, so transitions z and $\overline{z}$ are separate, distinguishable transitions placed in different bins.

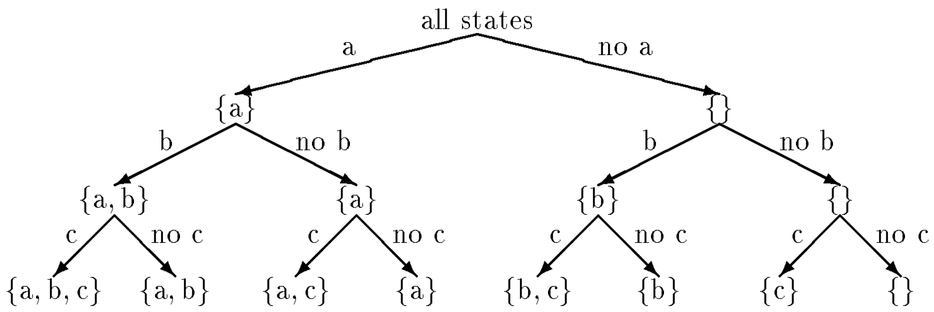


Figure 7.2: Initial Bin Split for Minimization

If a bin contains no states, it is removed or not created. Hence the initial partition may contain significantly less than the 2^n bins.

3. Following the initial split, this algorithm iteratively splits these bins using branching time bisimulation until no more splits are possible.

   (a) The following steps are repeated for each bin, until all bins have been examined and no extra bins were created.

   (b) Each state in the bin is walked, and a data structure is created which indicates the destination bins for each transition in the state. For example, two a transitions that go to bins 1 and 2 are recorded in an association list as (a (1 2)). The τ transitions must be walked just as in step 1 above.

   (c) Each transition label is then examined. All states in the bin must have an identical set of destination bins for each transition or the current bin must be split into separate bins distinguishable by the different destination possibilities. If all association lists are identical for each transition label, the transitions in this bin cannot be distinguished and the bin is not split. The association lists must be recalculated for those transitions that have the current bin as a destination whenever a split occurs to assure that the referenced states have not been split into another bin. For efficiency, each bin is iteratively examined until there no more splits can be made.

7.5 Analyze Usage Example

This section briefly shows a verification session using Analyze. Analyze is a prototype verification and synthesis tool written in Common Lisp. The user interface is designed as a set of functions that can be called after the software is loaded.

Following is a small example circuit based on a railroad crossing proposed by Bradfield and Stirling [BS90]. In this example, a car and train must not be allowed to cross the intersection at the same time. This model is based on transitional semantics, so the crossing will be the transition of a label. The specification is derived using three parallel processes – one for the car, one for the train, and one for the semaphore. The semaphore acts like a "stop light", only allowing either the car or train to enter the intersection at a time. Following is the definition of the specification using the notation accepted by the CWB and Analyze ("bi" is used to name a process using the Con rule, and label complementation is represented using the quotation `'g` rather than an overbar $\overline{g}$).

```
bi CAR-TRAIN-SPEC
(CAR | TRAIN | CTSEM) \{g,p}


bi CAR car.g.'ccross.'p.CAR
```

```
bi TRAIN train.g.'tcross.'p.TRAIN


bi CTSEM 'g.p.CTSEM
```

This specification can be tested behaviorally and for invariant properties. The following tests were made in the Concurrency Workbench. The `if` command reads a file into the CWB, and the `cp` command is used to see if a process satisfies the formula.

```
if car-train.ccs


cp CAR-TRAIN-SPEC
LIVE car
*** true


cp CAR-TRAIN-SPEC
LIVE train
*** true


cp CAR-TRAIN-SPEC
LIVE 'ccross
*** true


cp CAR-TRAIN-SPEC
LIVE 'tcross
```

```
*** true


cp CAR-TRAIN-SPEC
CONCURRENT car train
*** true


cp CAR-TRAIN-SPEC
MUTEX2 'ccross 'tcross
*** true


cp CAR-TRAIN-SPEC
HANDSHAKE car 'ccross
*** true


cp CAR-TRAIN-SPEC
HANDSHAKE train 'tcross
*** true
```

The specification is live, the outputs are mutually exclusive for concurrent arrivals, and the handshake protocols are obeyed. The specification is not a valid implementation because there is computation interference on the $\overline{g}$ signal and there are multiple output drivers on the $\overline{p}$ signal.

The specification can be decomposed using a number of approaches. For the first pass, let's assume that the specification is to be decomposed into specifications of macro module components, using inverters, TOGGLEs, mutual exclusion elements (MEs), MERGE gates, and signal sinks (WOOD). The analog device for creating mutually exclusive signals is the ME element, defined in Table 6.5 on Page 164. Note that the ME device is a level-sensitive circuit. This means that the interface for this circuit must convert from transition to four-cycle logic to access the ME element. The initial pass of a decomposition appears in Figure 7.3.

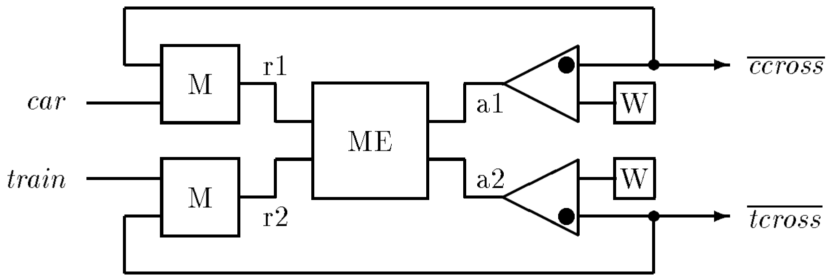


Figure 7.3: Initial Crossing Decomposition

This decomposition is specified in CCS as "CTImpl-Fails" by the following statements:

```
bi CTImpl-Fails
(  VehicleObv[car/in, r1/r, a1/a, ccross/out]      \
 | VehicleObv[train/in, r2/r, a2/a, tcross/out]    \
 | MESPEC                                          \
) \{ r1, a1, r2, a2 }


bi VehicleObv
(  MERGE[in/a, fb/b, r/c]       \
 | TOGGLE                       \
```

```
 | WOOD[c/a]                    \
 | IFORK[b/a, out/b, fb/c]      \
) \{ fb, b, c }
```

Since the behavior of the car and the train are the same as can be seen from closer examination of the specification Figure 7.3, the same definition can be shared. Relabeling is used to change the names for the correct communication interaction.

This implementation is tested by Analyze, which points out some errors. Analyze is written in Common Lisp, so commands must be parenthesized. Analyze is loaded with the `load-analyze` function, a file of CCS agents is read with the `parse-agents` command, and verification is carried out with the `analyze` function. The default mode uses the unbounded delay speed-independent model.

```
> (load "load-analyze")
> (parse-agents "car-train.ccs")
> (delay-insensitive-mode)
> (analyze '|CTImpl-Fails| 'car-train-spec)


;;;
;;; Parameters set for delay-insensitive analysis with trace verification
;;;
;;; Generating a trace-determinate specification from CAR-TRAIN-SPEC ...
;;;   ... minimized spec contains 8 states
;;;   ... successfully generated TD specification with 4 states
;;; Generating and trace verifying CTImpl-Fails against CAR-TRAIN-SPEC ...
;;;
```

```
;;; ERROR! Computation interference encountered!
;;;     Signal  'r2  in agent CTImpl-Fails*
;;;   Trace: (car 'r1 train 'r2 t 'a2 'tcross ('fb) 'r2 train 'r2)



;;;
;;; The top-level agent contains 80 unminimized states
;;;
;;;
;;;   The following are warnings or errors detected during analysis:
;;;        - The agent contained computation interference.
;;;          If this is an implementation it is NOT conformant to
;;;          a specification.  Otherwise synchronizations are incomplete
;;;          to implement the specification.
;;;        - This agent contains computation interferences in some
;;;          internal subcells.  This can cause an error unless it
;;;          occurs exclusively in unreachable states.
;;;        - Duplicate error type messages were suppressed.
;;;
Warning:

         11 errors encountered during creation of agent CTImpl-Fails*-0.
              Do NOT trust the behavior of this agent!
```

Analyze points out a violation in this implementation that is due to a race between the inputs into the MERGE element. In the trace above, the feedback signal from the output of the TOGGLE and the *train* input can flip the input to the ME element before it has responded. This could result in a deadlock or a runt pulse on the output. A modification of this specification is shown in Figure 7.4.

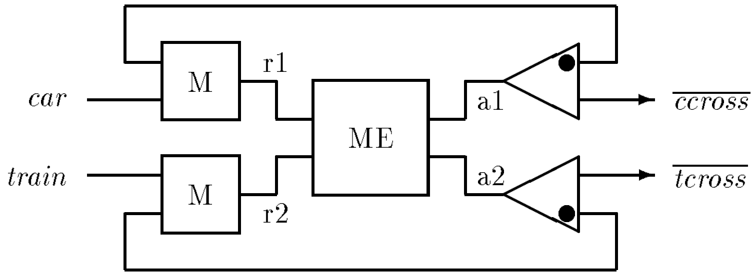


Figure 7.4: Trace Crossing Decomposition

This implementation verifies using trace conformance but not logic conformance (bisimulation). Both $\overline{ccross}$ and $\overline{tcross}$ can transition concurrently, yet trace conformance cannot detect this because all the correct traces are generated. The final attempt is shown in Figure 7.5. This decomposition is trace and logic conformant to the specification.

Another method of implementing this circuit is to decompose the initial specification into burst-mode specifications which conforms to the specification. Figure 7.6 is a "transition" burst-mode definition of the circuit that interfaces the environment with the ME element. This specification, when composed with the ME, creates a circuit that is verified with logic conformance against the original specification CAR-TRAIN-SPEC. This burst-mode specification can be directly implemented using MEAT. The interfaces from Figures 7.3 through 7.5 can be verified against this specification for conformance. Only the circuit of Figure 7.5 conforms to this graph.

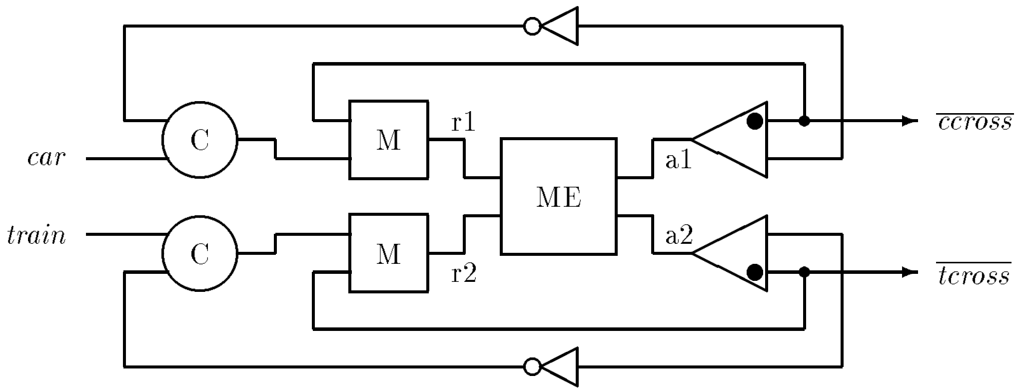


Figure 7.5: Correct Crossing Decomposition

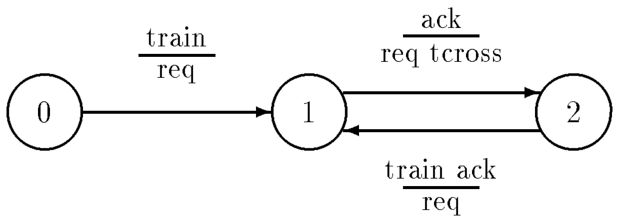


Figure 7.6: Burst-mode Transition Graph for Train

7.6 High Level Synthesis

This section describes a method for high level synthesis of verified circuits. This hierarchical synthesis starts with a specification, and applies conformance at all levels in the hierarchy in a top down fashion to verify the final circuit implementation conforms to the specification.

Unfortunately conformance is *not* a congruence, which could invalidate some results. A congruence does not hold when the *initial* state of the system contains an unstable summation. A summation is unstable when a τ transition is a possible action in the summation. The congruence does not hold on the summation because the τ transition can arbitrarily change the derivations of the initial state. Fortunately, it is a simple test to assure that the verification is a congruence by checking the stability of the initial state of the specification and implementation as a side condition for verifying conformance.

The high level synthesis method presented here is targeted for high performance VLSI implementations. This is a *directed* synthesis system that requires the talent of an engineer to choose amongst the myriad of architectural choices such as parallel versus serial, area and time tradeoffs, etc. A major advantage of this system over other directed synthesis systems is the ability to *verify* correct implementations rather than just assume that the implementations are correct by construction or by simulations. This would have detected the deadlock and other problems that existed in the initial Post Office design.

Figure 7.7 shows the method for high level synthesis that is supported by Analyze. This synthesis system produces a set of communicating burst-mode state machines as leaf nodes. Software tools are named in parenthesis that support the labeled operation.

The first synthesis step requires an informal description to be transformed into a formal specification such as CCS. The informal definition is typically an informal natural language description of the interface or circuit behavior, or an informal description using block diagrams, state diagrams, or timing diagrams. An object oriented style of formalizing parallel specifications is described in [SABL93].

The formal description should be tested to ensure that it is correct before embarking on an implementation, as described in [Liu92]. If the implementation fails the property or behavioral tests, the formal description should be modified and retested. The design should be made as *loose* as possible such that it does not violate

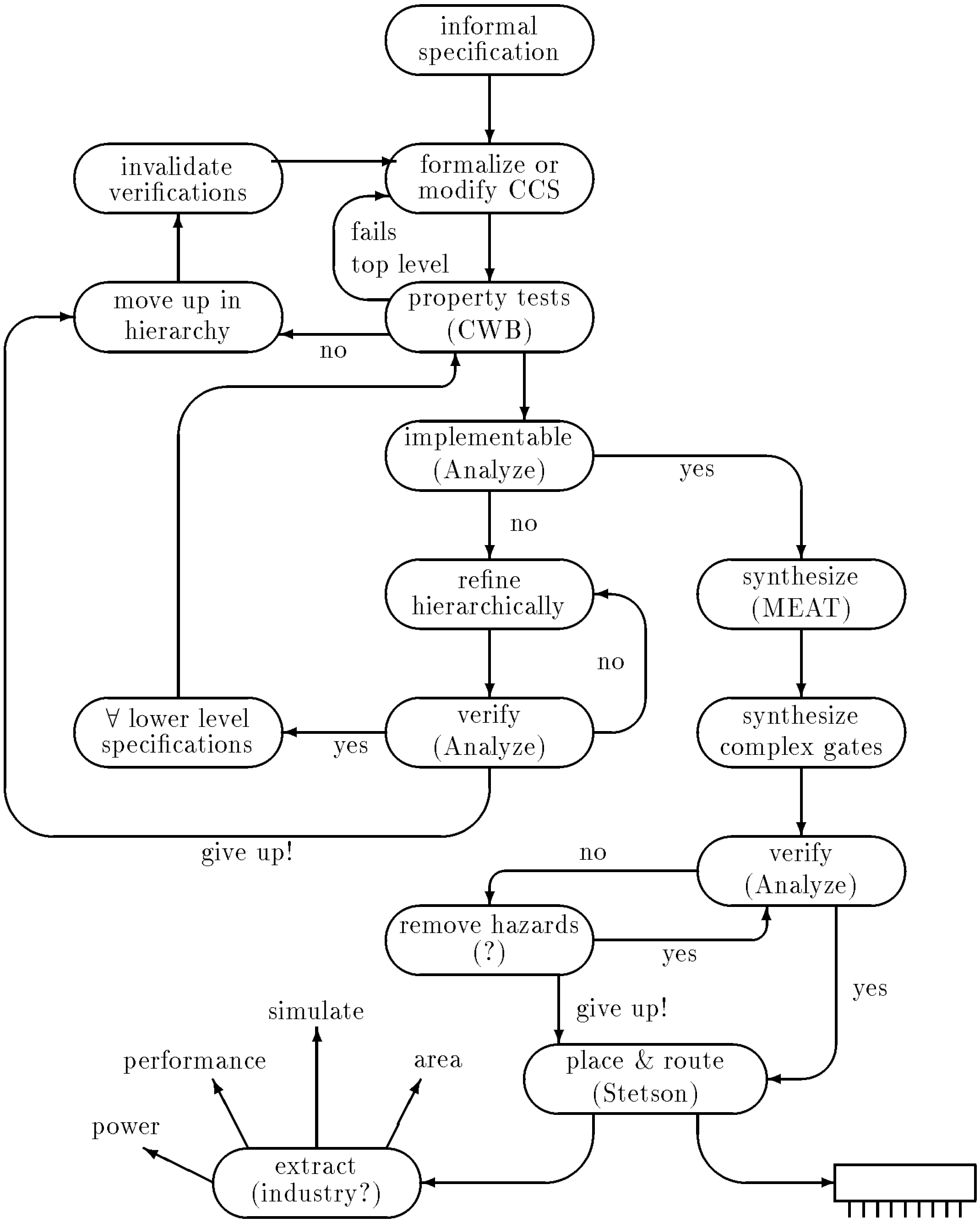


Figure 7.7: Synthesis Procedure

design constraints, yet leaves as much flexibility as possible for design decisions.

When the designer is satisfied that the loose specification meets the design constraints, the specification must be refined into an implementation. A specification must be refined under three circumstances:

1. The implementation has hidden the complexity of an internal behavior.
2. There is a communication which cannot be implemented as specified.
3. The specification cannot be implemented as a burst-mode state machine.

If complexity has been hidden – such as modeling an addition operation with the communication label 'add' – then the implementation details of the operation will need to be specified. The second cause for refinement is an illegal communication which cannot be implemented. These illegal communications occur in a circuit as computation interference, described in Section 7.3. Finally, the implementation can be tested as a valid burst-mode state machine. If the implementation verifies as a valid burst-mode specification, a circuit can be synthesized directly from the specification.

If specification is not a valid burst-mode state machine, then it will need to be refined into a set of parallel subspecifications. These refined subspecifications will then be tested for conformance against the specification. If the parallel subspecifications do not conform to the specification, the subspecifications will need to be modified until they are correct. If the designer cannot find any way to correctly implement the specification, the designer will need to move up one level in the hierarchy, and create a new refinement for that level. All verifications that were dependent on the modified refinement must be invalidated and reverified against the new agents. This backtracking can also be used to investigate alternate design approaches.

When the specification is a valid burst-mode definition, the circuit should be synthesized and then verified for hazards. Typically the hazards can be removed by the techniques presented in Section 3.7. If no hazard free implementation can be created, the hazard must be controlled in the layout, place, and route steps. Once the circuit has been placed and routed, layout information should be passed up through the system as back annotations. The Analyze system cannot currently utilize back annotated information such as delays, performance, or power consumption figures.

7.7 Burst-mode State Machine Verification

Decomposition is necessary under three circumstances as indicated on Page 206. Computation interference points out unimplementable handshakes, and the designer should be aware of when a complex operation has been modeled abstractly. The third condition for decomposition occurs when an agent is not a valid burst-mode specification. Terminal burst-mode specifications can be directly implemented if they conform to the rules from Section 4. The following steps are necessary enhancements to Analyze to verify that an agent is a legal burst-mode specification.

1. Inputs and outputs cannot be concurrently enabled (Rule 1).

2. Input bursts must be confluent (see Definition 24) and may not be empty (Definition 1).

3. Output bursts may be empty and need not be confluent (Definition 2).

4. There must be an even number of transitions for each label in the sort of the agent (Rule 7).

5. The AFSM is closed and determinate (Rule 8).

6. The environment will enable only a single input burst from any state (Rule 9).

The first two steps ensure that inputs and outputs are segregated into bursts, and that the bursts are semi-modular. The first step can easily be checked mechanically for any agent, independent of its sort. However, it is very difficult to determine from an arbitrary agent specification what constitutes a valid input burst, particularly when output bursts may be empty. By adhering to the burst-mode notational extension to CCS presented in Table 4.1, bursts can easily be specified and Analyze guarantees that the transition is confluent and semi-modular by generating all the necessary interleavings. Verifying the confluence of input bursts is implemented as part of Analyze, so it is the responsibility of the designer to use the burst-mode notation or assure that the interleavings are correctly specified such that burst-mode is obeyed.

According to Definition 2, the output burst need not be confluent. Since all input bursts must be confluent, all interleavings of outputs can be accepted. If a circuit only generates a subset of those interleavings, the system will continue to operate properly. This makes specifications much looser resulting in increased design freedom. However, this presents a problem for verification. Specifications will typically generate all output burst interleavings. If the implementation does not produce all these interleavings, it will not conform to the specification according to conformance (Definitions 25 and 30). Currently the specification must be modified to reflect the subset of the interleavings that the implementation will produce. Further research is required to allow logic conformance to automatically verify circuits where the interleavings of output bursts are subsets of the specification.

Current technology requires that burst-mode state machines are deterministic. Arbitration must be accomplished with an arbiter or mutual exclusion element.

Proposition 16 *A determinate agent will obey Rule 8 in Section 4.6, which guarantees that from any given state, there will not be two burst-mode transitions where the labels of one transition are a subset of the other.*

Proof By Definition 23, any s-derivative must result in bisimilar states. By Definition 1, if one burst is a subset of another then the same sequence s of observable actions must be possible in both bursts. If the agent is determinate, then these sequences $P \overset{s}{\Rightarrow} P'$ and $P \overset{s}{\Rightarrow} P''$ must arrive at the same state since $P' \approx P''$. Hence Rule 8 must hold as the subsequence can only be part of the longer input burst. □

Therefore, verifying that a circuit specification is determinate is sufficient to assure Rule 8 holds.

CCS is a transition based protocol, whereas digital logic is bistable. Therefore some preprocessing, mentioned in step 4 (from Rule 7 on Page 93), may be necessary to create a specification that can be directly synthesized using MEAT. This rule assures that the correct rising and falling of voltage levels is specified.

The final requirement for burst-mode state machines assures that, when there are multiple input bursts available from a single state, the bursts are driven by the environment in a mutually exclusive fashion. This cannot be verified by examining the state machine independently, but requires analysis of the environment of the state machine. The mutual exclusivity of bursts will automatically be detected when verifying the circuit in its environment *if and only if the signal transitions in all other bursts are not enabled in the destination states.* Otherwise Analyze cannot automatically verify the mutual exclusive burst requirement. For example, if the state fragment in Figure 7.8(a) is composed into a circuit that produces $\overline{a}$ and $\overline{b}$ concurrently, then computation interference will occur in states 1 and 2. Analyze cannot automatically verify mutual exclusivity on $\overline{a}$ and $\overline{b}$ provided by the environment to fragment Figure 7.8(b) because state 1 can make a b transition and state 2 can make an a transition.

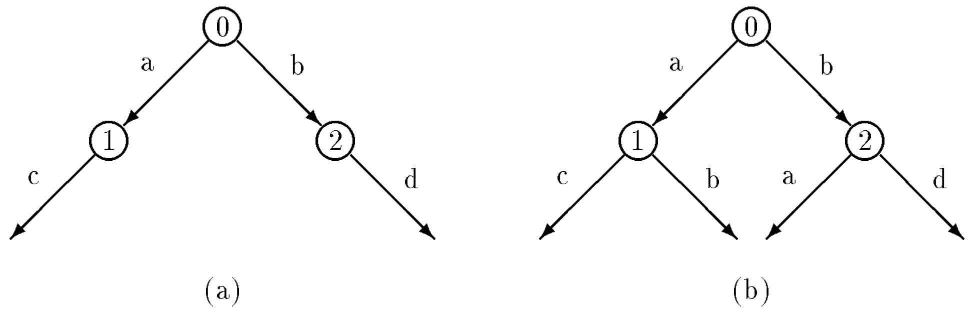


Figure 7.8: Environmental Burst Constraints

The MUTEX2 macro of the Modal-μ calculus defined in Chapter 6 can be used when conformance cannot determine mutual exclusivity of the bursts. Unfortunately the specifications need to be modified for this test. The completion of the bursts in question will each signal completion to the TEE component, and the $\overline{c}$ output of the TEE will be connected following the final signal in the burst. The $\overline{b}$ signal indicates completion of the burst. The component is then placed into the circuit environment, and if the $\overline{b}$ signals of the bursts in question are not mutually exclusive, then the environment supplies the bursts concurrently.

$$\text{TEE} \stackrel{\text{def}}{=} a.\overline{b}.\overline{c}.\text{TEE} \tag{7.1}$$

For example, the burst-mode specification of the circuit in Figure 4.4 on Page 97 does not require the application of modal-μ formulae to its environment. The only state with a choice of transition bursts is state 4. Since state 5 cannot accept a transition on *deliver*, concurrently presenting *deliver* and *ack-send* will result in computation interference on *deliver* in state 5. However, the nonblocking arbiter specification of Figure 4.3 on Page 94 will *not* show computation interference because all inputs are valid transitions in *every* state. Compose a TEE between states 0 and 1 and states 0 and 4 and compose the circuit with its environment. The MUTEX2 formula will not be satisfied by the resulting circuit, showing that the environment is not well behaved for this AFSM. Therefore, as discussed in Section 4.5, the SEQUENCER circuit needs to shield this AFSM from concurrent input changes supplied by the environment.

7.8 Summary

The CCS calculus is an extremely concise and useful language for specifying parallel asynchronous circuits, particularly when the specifications are created in an object-oriented style based on the parallel composition operator. CCS is also amenable to automatic proof systems due to the succinctness of the language, and proof systems are implemented in the Concurrency Workbench and the Analyze tool of this thesis. The attention to structural aspects of concurrent design support accurate hierarchical circuit synthesis.

Pure CCS cannot modeling and reasoning about circuits specifications at the device level due handshake communication. These problems were discussed, and solutions were proposed and implemented in a software prototype CAD tool called Analyze. Unfortunately some of these solutions required slight changes to the transitional rules of CCS, in particular the parallel composition operator was redefined in terms of a parallel conjunctive composition. The changes impact the basic CCS as minimally as possible, solely changing aspects where accuracy or simplicity of modeling hardware systems would be compromised. These modifications were all made as transparently as possible, by pushing the complexity of the changes, such as in the composition operator, into the Analyze tool rather than burdening the designer with more additional notation, etc.

These modifications support a powerful synthesis and proof tool for asynchronous circuits. Invariant analysis of specifications was presented. These are divided into sets which can be carried out independent of a specification's structure, and those that are dependent upon the behavior and component interactions. Definitions and formulae were presented for the hierarchical verification of circuits based on invariant analysis. The computational speed of Analyze is much improved over the Concurrency Workbench, as many common applications including minimization require an order of magnitude less CPU time to complete. Analyze attains this performance advantage through compositional algorithms and by a more restricted applicability.

A high level synthesis procedure based on Analyze is presented that can synthesize verified circuit implementations in a top-down fashion. This procedure is targeted at implementations comprising communicating burst-mode state machines. The steps for validating a correct burst-mode specification are also described.

The major contribution of this chapter is the software prototype CAD tool developed for the verification and synthesis of asynchronous circuits. Although there is still work to do, such as supporting hierarchical burst-mode verification and improving performance, this tool has proven very useful to myself and others who have exercised its capabilities [vG94]. This tool, along with a short user manual and set of examples, has been made publicly available via ftp.

The remainder of this chapter notes some of the contributions that are part of the Analyze tool and synthesis philosophy.

1. A prototype tool has been developed for supporting the hierarchical, top-down synthesis and verification of asynchronous systems. The hazard modeling of Analyze is more rigorous than in other tools.

2. Analyze includes all of the common delay models: delay-insensitive, quasi delay-insensitive, speed-independent, and burst-mode. When a violation occurs, signal backtraces are included to aid the designer in determining the cause of the fault.

3. Analyze includes multiple equivalences. Currently, complete trace semantics and branching time bisimulation semantics have been defined as "conformances".

4. A designer directed hierarchical top-down synthesis methodology has been developed.

5. A new parallel composition operator, called parallel conjunction, has been defined and implemented. The restriction operator for parallel composition

has been changed to use hiding semantics for the conjunction operator as this operator restricts independent actions as a side condition.

6. The definition of computation interference is created for labeled transition systems. This is an extension of CCS transition semantics, and is the backbone of the verification and synthesis systems developed for this thesis.

7. The steps required for the verification of correct burst-mode specifications is developed and spelled out. This is necessary for the top-down synthesis methodology.

8. An efficient algorithm for state minimization and branching time bisimulation is presented in this chapter, and implemented in Analyze.

Chapter 8

Conclusions

"It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and hardware design, is unsound and clumsy because the people who do it do not have any clear understanding of the fundamental principles underlying their work. Most of the abstract mathematics and theoretical work is sterile because it has no contact with the real computing. One of the central aims of the PRG, as a teaching and research group, has been to set up an atmosphere in which this separation cannot happen."

C Strachey 1974

Perhaps the most significant result of this thesis has been my metamorphosis from a "devil's advocate" railing against the lack of utility and maturity of formal methods for circuit design into a devotee. The simplicity of CCS syntax and semantics are the foundation of my newly acquired interest. It is a specification notation that is natural for designers and engineers and is one with which formal proofs can be carried out automatically.

Chills go down my spine when I recall the "old ways" of simulation and the months I spent modifying simulation sequences in the Post Office and manually inspecting state machines in an attempt to discover the cause of a deadlock. However, the most significant advantages of verification have yet to move from the conceptual, intellectual, and theoretical domains into the labs of engineers. The success of this work is, in a way, battling a two headed dragon as both formal methods *and* asynchronous design must become mainstream for this to happen.

If the momentum behind the work in simulation and clocked systems is to be stemmed and turned, significant advantages of other techniques must become apparent. The conceptual advantages of verified asynchronous systems have been espoused here and in other works. The momentum will change only if *practical* solutions to everyday engineering problems are available. These practical solutions will only come about today through the synergy of merging theory, software engineering, and circuit design in the form of a toolkit. The prototype Analyze tool of this thesis is a first stab at a practical workbench for the synthesis and verification of industrial strength asynchronous integrated circuits.

8.1 Challenges

Significant challenges must be addressed before verification becomes widely used. This section quickly covers some of the areas where further work is required to facilitate these concepts.

8.1.1 Complexity

Invariant analysis inevitably requires that all states be examined. The performance of verification techniques is through the complexity of the algorithms directly proportional to the size of the agents being examined. Further, the state space of parallel systems grows as the product of the parallel terms. The difficulty of controlling such consuming complexity becomes apparent when one considers a simple example. The nacking (or blocking) arbiter implementation of the Post Office requires a sequencer – a library component for many of the macro module based asynchronous synthesis systems. My implementation contains four ME's, 12 2-input NAND gates, six inverters, and two set-reset flip flops. The overall complexity of such a rather simple circuit is $16^4 \times 8^{12} \times 2^6 \times 8^2 = 18,446,744,073,709,551,616$ states (which exceeds the address size of a 64 bit architecture). This is typically referred to as the "state explosion problem". Some approaches to reducing the complexity are discussed here.

Compositional Tools

Significant improvements in performance can be achieved with compositional algorithms. Much of the performance gains of Analyze over the Concurrency Workbench can be attributed to its compositional style.

For example, let's review the comparison between Analyze and the CWB for the simple C-element circuit from Section 6.5. This design contains three 2-input AND gates and a three-input OR gate (or in a CMOS implementation three 2-input NAND gates and one 3-input NAND gate). This parallel implementation is bounded by $8^3 \times 16 = 8192$ states. The workbench's non-compositional algorithms create all 8192 states in parsing the circuit, expending 193 CPU seconds. Analyze, which creates the circuit compositionally, only creates the 36 restricted states, requiring 0.6 seconds of computation time. Verification is also compositional in Analyze, and is more efficient than parsing. The verification of the C-element using the burst-mode hazard model takes 0.2 seconds of CPU time and touches 23 states as these states conform to the four states of the specification.

Hierarchy and Other Mechanisms

Optimally there would be some means of hiding the complexity of parallelism from engineers to the extent that large systems could be verified efficiently. Other simple and obvious techniques have been used by the Analyze prototype, such as hashing. More complicated techniques, such as applying induction techniques to regularly interconnected arrays of components, BDD type representations, and more efficient algorithms may achieve some limited success for controlling complexity.

However, due to the inherent exponential state explosion of parallel components, careful hierarchical decomposition will always remain a fact of life in highly parallel architectures. Although CCS facilitates this by accurately modeling the observable effects of components throughout all levels in the hierarchy, controlling complexity through this process of decomposition can be very challenging, particularly when the obvious partitions are just too large to verify. For example, even verifying the implementation of the sequencer referred to above requires hierarchical modeling.

8.1.2 Tool Support

Unfortunately, designers of industrial strength asynchronous VLSI circuits have been forced to produce chips without sufficient tool support. Without such assistance, the cost and possibility of errors is too great for asynchronous circuit design become mainstream.

An applicable theory that has not been put to work simplifying or solving problems has been squandered. The practical application of the theories presented here, that are in themselves based on the founding work of others, is embodied in a software prototype that is freely available via anonymous ftp. The work herein is a first step toward the ultimate application of these principles in the rapid creation of verified integrated circuits form an asynchronous designers workbench of tools.

8.2 Analyze Critique

The application of the principles in this thesis are somewhat limited in scope. Value passing has not been implemented as part of the core CCS transition rules. This results in a dichotomy of effective applicability of CCS toward circuit design. It can be very efficient for verifying control, yet quite inefficient for datapath logic. Fortunately this melds well with the asynchronous design style I have developed over the years. Regular datapath logic is fairly easy to design correctly and is of universal applicability, whereas correctly designing custom control can be very challenging. Therefore this method has only been used to verify control and the datapath interfaces.

Complete verification currently requires the use of the Concurrency Workbench as well as Analyze. Whenever the satisfaction of application specific temporal equations is required, as in certain cases to verify mutually exclusive environmental behavior of some input bursts, the CWB must be used. The current Analyze prototype does not yet implement the universal invariant tests such as liveness. It would be convenient to add the capabilities of verifying liveness and the process logics presented in this thesis to the Analyze tool.

The multi-way synchronization of the conjunction operator, trace conformance, and computation interference capabilities of Analyze allow it to couple the simplicity of the CCS syntax with the ability to accurately model and verify circuits.

The textual and function based user interface of Analyze is rather weak. There are no programs that can generate textual CCS descriptions from schematic drawing tools. The tool also contains some theoretical flaws from the (too) early implementation of logic conformance that result in erroneous results under certain conditions. There are also some inflexibilities forced upon specifications due to the parser.

One of the largest deficiencies lies in the incompleteness of some essential aspects of the tool. Trace conformance is computed directly but logic conformance is not. Checking when an agent definition is a valid burst-mode specification has not yet been implemented. Hierarchical verification of burst-mode controllers has not yet been implemented, but the theory is complete. Unfortunately this has postponed a goal of verifying major portions of the Post Office implementation. The synthesis procedure is not fully supported as there is no bookkeeping method for tracking the validity of verifications over many hierarchical levels, and annotation is neither stored nor used. A more automated system of estimating complexity to aid in the directed decomposition would greatly reduce designer's workload. Transistor-level and tristate models have not been developed. Hence complex gates must be characterized externally and their behavior imported into Analyze before they can be used.

8.3 Future Directions

My assessment is that CCS is a very promising foundation for the formal verification and synthesis of asynchronous circuits. The simplicity of the model is a feature and impediment. Specifications can be concise, parallel, and "object oriented". The conjunctive transition rules allow most hardware components to be directly modeled. However, datapath logic is not modeled efficiently. Verification of regular datapath processes such as RAMS, multipliers, registers, etc. can be carried out much more efficiently with HOL or other inductive proof techniques. The CCS formalism is very similar to the asynchronous design style in that it seems to vastly simplify the difficult aspects of design (control verification), whereas the easier tasks such as datapath module verification are neither easily nor efficiently automated.

Typical tradeoffs exist between a simple, efficient, restricted model versus a broadly applicable, unwieldy, and less efficient one. The ramifications of such tradeoffs seem to have greater impact on performance and the ability to achieve the desired goals (proof automation) than with other tools such as programming languages and simulation based circuit models. Most tools based on labeled transition systems that have been broadened in scope seem to lose the clarity and simplicity of the underlying proof system without acquiring offsetting benefits when applied to asynchronous systems. Such broader systems may not be simple or powerful enough to rival other more complex logic theorem proving systems such as HOL.

A better approach may be to make CCS and labeled transition systems such as Analyze companions to HOL or VHDL, applying each method to solve problems in their particular area of expertise. The inductive abilities of HOL can rapidly prove the correctness of datapath logic, whereas Analyze is more amenable to proving the correctness of control circuitry. VHDL could be used as the back end for circuit simulation and as the specification language for automatic synthesis, as well as to interface with place and route software available from vendors today. Contact has already been made between VHDL and HOL [vT93]. Unfortunately such a cooperation would most likely be difficult and only operate on a subset of the syntax of the more general systems. I would very much like to investigate the feasibility of such a cooperative tool.

This first prototype has proven its worth in the small set of applications to which it has been applied. Hopefully it will serve as a stepping stone for a second generation prototype. I would like to continue my work on the software engineering in this tool, fix the faults, improve the algorithms and runtime performance, and complete the open areas. The real test will come when this methodology is applied to an industrial strength design such as the Post Office, which I hope to do!

Bibliography

[AG92] Venkatesh Akella and Ganesh Gopalakrishnan. SHILPA: A High-Level Synthesis System for Self-Timed Circuits. In *International Conference on Computer-Aided Design (ICCAD-92)*, pages 587–591, 1992.

[And93] Henrik Reif Andersen. *Verification of Temporal Properties of Concurrent Systems*. PhD thesis, Aarhus University, Denmark, June 1993.

[Bai94] Andrew Michael Bailey. *Modeling, Design and Analysis of Digital Circuits Using Circal*. PhD thesis, University of Strathclyde, September 1994.

[Bak90] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley, 1990.

[BE92] J. A. Brzozowski and J. C. Ebergen. On the Delay-Sensitivity of Gate Networks. *IEEE Transactions on Computers*, 41(11):1349–1360, November 1992.

[BM92] P. Beerel and T.H.-Y. Meng. Automatic Gate-Level Synthesis of Speed-Independent Circuits. In *International Conference on Computer-Aided Design (ICCAD-92)*. IEEE Computer Society Press, November 1992.

[Bre90] G. Brebner. A CCS-based Investigation of Deadlock in a Multi-process Electronic Mail System. Technical Report, University of Edinburgh, 1990.

[Bru91] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[Bru92] G. Bruns. A Case Study in Safety-Critical Design. Technical Report ECS–LFCS–92–239, Edinburgh University, 1992.

[Bru93a] Erik Brunvand. Designing Self-Timed Systems using Concurrent Programs. *Journal of VLSI Signal Processing*, 7, 1993. Special issue on asynchronous circuits.

[Bru93b] Erik Brunvand. The NSR Processor. In *Proceedings of the 26th International Conference on System Sciences*, Maui, Hawaii, January 1993.

[BS89] E. Brunvand and R. F. Sproull. Translating Concurrent Communicating Programs into Delay-Insensitive Circuits. In Randall Bryant, editor, *International Conference on Computer-Aided Design (ICCAD-89)*, pages 262–265. IEEE Computer Science Press, 1989.

[BS90] J. Bradfield and C. Stirling. Verifying Temporal Properties of Processes. In J.C.M. Baeten and J.W. Klop, editors, *Concur 90: Theories of Concurrency, Unification, and Extension*, number 458 in Lecture Notes in Computer Science, pages 115–125. Springer Verlag, 1990.

[Cam91] Juanito Camilleri. *Priority in Process Calculi.* PhD thesis, University of Cambridge, March 1991.

[Car] Carnegie-Mellon University. *User's Guide to COSMOS.*

[CD73] Henry Y. H. Chuang and Santanu Das. Synthesis of Multiple-Input Change Asynchronous Machines using Controlled Excitation and Flip-Flops. *IEEE Transactions on Computers*, C-22(12):1103–1109, December 1973.

[CDS93a] W. S. Coates, A. L. Davis, and K. S. Stevens. Automatic Synthesis of Fast Compact Self-Timed Control Circuits. In *IFIP Working Conference on Design Methodologies*, pages 193–208, April 1993.

[CDS93b] W. S. Coates, A. L. Davis, and K. S. Stevens. The Post Office Experience: Designing a Large Asynchronous Chip. *Integration, the VLSI Journal*, 15(3):341–366, October 1993. Special issue on asynchronous systems.

[Chu87] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits From Graph-Theoretic Specifications.* PhD thesis, Massachusetts Institute of Technology, September 1987.

[Chu93] Tam-Anh. Chu. CLASS: A CAD System for Automatic Synthesis and Verification of Asynchronous Finite State Machines. *Integration, the VLSI Journal*, 15(3):263–289, October 1993. Special issue on asynchronous systems.

[CM72] W.A. Clark and C.E. Molnar. The promise of macromodular computer systems. In *Sixth IEEE Computer Conference*, pages 309–312, September 1972.

[Coh88] A. J. Cohn. A Proof of Correctness of the VIPER Microprocessor: The First Level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71, Norwell, Massachusetts, 1988. Kluwer. Proceedings from the Calgary workshop.

[Com92] R. Comerford. How DEC Developed Alpha. *IEEE Spectrum*, pages 26–31, July 1992.

[CS90] Rance Cleaveland and Bernhard Steffen. A Preorder for Partial Process Specifications. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of ConCur '90*, number 458 in LNCS, pages 141–151. Springer Verlag, 1990.

[Dav77] A. L. Davis. The Architecture of DDM1: A Recursively Structured Data-Driven Machine. Technical Report UUCS-77-113, University of Utah, Computer Science Dept, 1977.

[Dav92] A. L. Davis. Mayfly: A General-Purpose, Scalable, Parallel Processing Architecture. *Lisp and Symbolic Computation*, 5(1/2):7–47, May 1992.

[DCH+89] Al Davis, Bill Coates, Robin Hodgson, Richard Schediwy, and Ken Stevens. Mayfly System Hardware. Technical Report HPL-SAL-89-23, Hewlett Packard Company, April 1989.

[DHWT91] David L. Dill, Alan J. Hu, and Howard Wong-Toi. Checking for Language Inclusion Using Simulation Preorders. In K. G. Larsen and A. Skou, editors, *Proceedings of CAV'91*, number 575 in LNCS, pages 255–265, 1991.

[Dil89] David L. Dill. *Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1989.

[dNH83] R. de Nicola and M. C. Hennessy. Testing Equivalence for Processes. *Journal of Theoretical Computer Science*, 34:83–133, 1983.

[DNS92] D. Dill, S. Nowick, and R. F. Sproull. Specification and Automatic Verification of Self-timed Queues. *Formal Methods in Systems Design*, 1(1):30–60, 1992.

[DS87] William J. Dally and Paul Song. Design of a Self-Timed VLSI Multicomputer Communication Controller. In *Proceedings of the International Conference on Computer Design*, pages 230–234. IEEE Computer Society Press, 1987.

[Ebe88] Jo C. Ebergen. A Formal Approach to Designing Delay-Insensitive Circuits. Technical Report Computing Science Note 88/10, Eindhoven University of Technology, May 1988.

[Ebe91] Jo C. Ebergen. A Formal Approach to Designing Delay-Insensitive Circuits. *Distributed Computing*, 3(5):447–450, 1991.

[EG93] Jo Ebergen and Sylvain Gingras. A Verifier for Network Decompositions of Command-Based Specifications. In *Proceedings of the 26th International Conference on System Sciences*, Maui, Hawaii, January 1993. IEEE Computer Society Press.

[EP92] J. C. Ebergen and A. M. G. Peeters. Modulo-N Counters: Design and Analysis of Delay-Insensitive Circuits. In J. Staunstrup and R. Sharp, editors, *2nd Workshop on Designing Correct Circuits*, pages 27–46. Elsevier Science Publishers, June 1992.

[FDG+93] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A Micropipelined ARM. In *Proceedings of VLSI '93*, Grenoble, France, September 1993. Best paper award.

[Fer90] Jean-Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219 – 236, 1990.

[FM91] Jean-Claude Fernandez and Laurent Mounier. "On the Fly" Verification of Behavioral Equivalences and Preorders. In K. G. Larsen and A. Skou, editors, *Proceedings of CAV'91*, number 575 in LNCS, pages 181–191, 1991.

[GA93] G. Gopalakrishnan and V. Akella. Specification, Simulation, and Synthesis of Self-Timed Circuits. In *Proceedings of the 26th Hawaii International Conference on System Sciences*. IEEE Computer Society Press, January 1993.

[Geo68] George H. Barnes et al. The ILLIAC IV Computer. *IEEE Transactions on Computers*, 8(C-17):746–757, August 1968.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, Cambridge, 1993.

[Gra92] Brian T. Graham. *The SECD Microprocessor, A Verification Case Study.* Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, 1992.

[Hay81] A. B. Hayes. Stored State Asynchronous Sequential Circuits. *IEEE Transactions on Computers*, C-30(8), August 1981.

[Hay83] A. B. Hayes. Self-Timed IC Design with PPL's. In R. E. Bryant, editor, *Third Caltech Conference on Very Large Scale Integration*, pages 257–274, Rockville, Maryland, 1983. Computer Science Press, Inc.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, London, 1985.

[Hol82] Lee Hollaar. Direct Implementation of Asynchronous Control Units. *IEEE Transactions on Computers*, C-31(2), February 1982.

[Huf57] David A. Huffman. The Design and Use of Hazard-free Switching Networks. *Journal of the Association for Computing Machinery*, 4(41):47–62, January 1957.

[Hun86] W. A. Hunt. *FM8501, A Verified Microprocessor.* PhD thesis, Institute for Computing Science, University of Texas, Austin, February 1986.

[JU90] 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, *Proceedings of ConCur '90*, number 458 in LNCS, pages 342–466. Springer Verlag, 1990.

[KK79] P. Kermani and L. Kleinrock. Virtual Cut-Through: A New Computer Communication Switching Technique. *Computer Networks*, 3:267–286, 1979.

[Kra85] Glenn A. Kramer. Helios Design Consultant System. *SIGART Newsletter*, (92):92–93, April 1985.

[LABS93] Y. Liu, J. Aldwinckle, G. Birtwistle, and K. S. Stevens. Testing the Consequences of Specifications in Modal μ. In *1993 Canadian Conference on Electrical and Computer Engineering*, number Vol II, pages 987–990, Vancouver, Canada, September 1993.

[Lar89] K.G. Larsen. Modal specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, number 407 in LNCS, pages 232 – 246. Springer-Verlag, 1989.

[LBP94] Y. Liu, G. Birtwistle, and N. Paver. Specification of the Manchester Amulet 1: Execution Pipeline. Computer Science Department Technical Report, University of Calgary, June, 1994.

[Liu92] Ying Liu. Reasoning about Asynchronous Designs in CCS. Master's thesis, University of Calgary, 1992.

[LKSV90] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Synthesis of Verifiably Hazard-Free Asynchronous Control Circuits. Technical Report UCB/ERL M90/99, University of California at Berkeley, November 1990.

[LKSV91] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Synthesis of Verifiably Hazard-Free Asynchronous Control Circuits. In Carlo H. Sequin, editor, *Proceedings of the 13th Conference on Advanced Research in VLSI*, UC Santa Cruz, March 1991.

[Mar89] Alain J. Martin. The Design of a Delay-Insensitive Microprocessor: An Example of Circuit Synthesis by Program Transformation. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 244–259, 1989.

[Mar90] Alain J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In W.J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[Mar91] Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. Technical report, California Institute of Technology, Department of Computer Science, Pasadena, CA 91125, August 1991.

[MBL+89] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. The Design of an Asynchronous Microprocessor. In C.L. Seitz, editor, *Decennial Caltech Conference on VLSI*, pages 251–273. MIT Press, 1989.

[MC80] C. Mead and L. Conway. *Introduction to VLSI Systems*, chapter "System Timing". Computer Science. Addison Wesley, 1980. This chapter written by Charles L. Seitz.

[McC86] Edward J. McCluskey. *Logic Design Principles with Emphasis on Testable Semicustom Circuits.* Prentice Hall International Editors, 1986.

[MCS94] Alan Marshall, Bill Coates, and Polly Siegel. Designing an Asynchronous Communications Chip. *IEEE Design & Test of Computers*, 11(2):8–21, summer 1994.

[Mel88] T. F. Melham. Abstraction Mechanisms for Hardware Verification. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–291, Norwell, Massachusetts, 1988. Kluwer.

[MFR85] C. E. Molnar, T. P. Fang, and F. U. Rosenberger. Synthesis of Delay-insensitive Modules. In H. Fuchs, editor, *Chapel Hill Conference on VLSI*, pages 67–86, Rockville, MD, 1985. Computer Science Press.

[Mil65] R. E. Miller. *Switching Theory*, volume 2. Wiley, New York, New York, 1965. Chapter 10 is a review of Muller's work on speed independent circuits.

[Mil85] G. J. Milne. Circal and the Representation of Communication and Concurrency. *ACM Transactions on Programming Languages and Systems*, 1985.

[Mil89] Robin Milner. *Communication and Concurrency*. Computer Science. Prentice Hall International, London, 1989.

[MM91] Wenbo Mao and George J. Milne. An Automated Proof Technique for Finite-State Machine Equivalence. In K. G. Larsen and A. Skou, editors, *Proceedings of CAV'91*, number 575 in LNCS, pages 233–243, 1991.

[MM92] G. A. McCaskill and G. J. Milne. Hardware description and verification using the Circal-System. Technical Report HDV-24-92, University of Strathclyde, Department of Computer Science, Glasgow, Scotland, June 1992.

[Mol91] Faron Moller. *The Edinburgh Concurrency Workbench (Version 6.0)*. University of Edinburgh, August 1991.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: specification*. Springer-Verlag, New York, 1992.

[MT90] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of ConCur '90*, number 458 in LNCS, pages 401–415. Springer Verlag, 1990.

[ND89] Steven M. Nowick and David L. Dill. Practicality of State Machine Verification of Speed-Independent Circuits. In *1989 International Conference on Computer-Aided Design (ICCAD-89)*. IEEE Computer Society, 1989.

[ND91a] S. M. Nowick and D. L. Dill. Synthesis of Asynchronous State Machines Using a Local Clock. In *1991 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE Computer Society, 1991.

[ND91b] Steven M. Nowick and David L. Dill. Automatic Synthesis of Locally-Clocked Asynchronous State Machines. In *IEEE International Conference on Computer-Aided Design (ICCAD-91)*. IEEE Computer Society, 1991.

[ND92] Steven M. Nowick and David L. Dill. Exact Two-Level Minimization of Hazard-Free Logic with Multiple-Input Changes. In *International Conference on Computer-Aided Design (ICCAD-92)*. IEEE Computer Society, 1992.

[NDDH93] S. M. Nowick, M. E. Dean, D. L. Dill, and M. Horowitz. The Design of a High-Performance Cache Controller: A Case Study in Asynchronous Synthesis. *Integration, the VLSI Journal*, 15(3):241–262, October 1993. Special issue on asynchronous systems.

[Par81] D.M.R. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of 5th G. I. Conference on Theoretical Computer Science*, number 104 in Lecture Notes in Computer Science, pages 167–183. Springer Verlag, 1981.

[Par87] J. Parrow. Verifying a CSMA/CD-protocol with CCS. Technical Report ECS–LFCS–87–18, University of Edinburgh, 1987.

[Pav94] Nigel Charles Paver. *Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, University of Manchester, 1994.

[PDF$^+$92] N. Paver, P. Day, S. B. Furber, J. D. Garside, and J. V. Woods. Register Locking in an Asynchronous Microprocessor. In *IEEE International Conference on Computer Design (ICCD-92)*, pages 351–355, October 1992.

[Pet81] J. Peterson. *Petri Net Theory and Modeling of Systems*. Prentice Hall, 1981.

[PT87] Robert Paige and Robert Tarjan. Three partition refinement algorithms. *SIAM Journal of Computation*, 16(6):973–989, 1987.

[Rob61] J. E. Robertson. Problems in the Physical Realization of Speed Independent Circuits. In *Proceedings of the 2nd AIEE Symposium on Switching Circuit Theory and Logical Design*, pages 106–108, Detroit, MI, October 1961. Early treatment of the isochronous fork problem.

[Rub87] Steven M. Rubin. *Computer Aids for VLSI Design.* VLSI Systems. Addison-Wesley, 1987.

[SABL93] K. S. Stevens, J. Aldwinckle, G. Birtwistle, and Y. Liu. Designing Parallel Specifications in CCS. In *1993 Canadian Conference on Electrical and Computer Engineering*, number Vol II, pages 983–986, Vancouver, Canada, September 1993.

[SDC93] K. S. Stevens, A .L. Davis, and W. S. Coates. The Post Office Experience: Designing a Large Asynchronous Chip. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, pages 409–418, January 1993.

[SMD93] Polly Siegel, Giovanni De Micheli, and David Dill. Automatic Technology Mapping for Generalized Fundamental-Mode Asynchronous Designs. In *30th ACM/IEEE Design Automation Conference*, pages 61–67, 1993.

[SMSM79] Ivan E. Sutherland, Charles E. Molnar, Robert F. Sproull, and J. Craig Mudge. The TRIMOSBUS. In Charles L. Seitz, editor, *Proceedings of the Caltech Conference on Very Large Scale Integration*, pages 395–426, January 1979.

[SRD86] Kenneth S. Stevens, Shane V Robison, and A.L. Davis. The Post Office – Communication Support for Distributed Ensemble Architectures. In *Proceedings of 6th International Conference on Distributed Computing Systems*, pages 160 – 166, May 1986. Best paper award.

[Ste84] Kenneth S. Stevens. "The Soft Controller". Master's thesis, University of Utah, October 1984.

[Ste86] Kenneth S. Stevens. The Communications Framework for a Distributed Ensemble Architecture. Technical Report 47, Schlumberger Palo Alto Research Center, 3340 Hillview Ave, Palo Alto, Ca. 94304, February 1986.

[Ste92] Kenneth S. Stevens. Automatic Synthesis of Fast, Compact Self-Timed State Machines. Technical Report 92/495/33, University of Calgary, Computer Science Department, December 1992.

[Sti91] Colin Stirling. An Introduction to Modal and Temporal Logics for CCS. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, number 491 in LNCS, pages 2–20. Springer-Verlag, 1991.

[Sti92] Colin Stirling. Modal and Temporal Logics for Processes. Tech Report ECS-LFCS-92-221, Laboratory for the Foundations of Computer Science, Computer Science, University of Edinburgh, 1992.

[Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. Turing Award Lecture.

[Sys89] The HOL System. Reference Manual. Technical report, Cambridge Research Center, SRI International under contract to DSTO Australia, Cambridge, England, 1989.

[Tar55] A. Tarski. A Lattice Theoretic Fixpoint Theorem and its applications. *Pacific Journal of Mathematics*, 5, 1955.

[TF92] Y. Tamir and G. L. Frazier. Dynamically-Allocated Multi-Queue Buffers for VLSI Communication Switches. *IEEE Transactions on Computers*, 41(6):725–737, June 1992.

[Udd84] Jan Tijmen Udding. *Classification and Composition of Delay-insensitive Circuits*. PhD thesis, Technical University of Eindhoven, 1984.

[Ung69] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, New York, New York, 1969.

[vB92a] C. H. van Berkel. Beware the Isochronic Fork. *Integration, the VLSI Journal*, 13(2):103–128, 1992.

[vB92b] C. H. (Kees) van Berkel. *Handshake Circuits: an Intermediary Between Communicating Processes and VLSI*. PhD thesis, Technical University of Eindhoven, May 1992.

[vBBK+94] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters. Asynchronous Circuits for Low Power: A DCC Error Corrector. *IEEE Design & Test of Computers*, 11(2):22–32, summer 1994.

[vBKR+91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald W.J.J. Saeijs, and Frits Schalij. The VLSI programming language Tangram and its translation into handshake circuits. In *Proceedings of the European Design Automation Conference*, pages 384–389, 1991.

[vG90a] R. J. van Glabbeek. The Linear Time - Branching Time Spectrum. Technical Report CS-R9029, Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, 1990.

[vG90b] R. J. van Glabbeek. The Linear Time - Branching Time Spectrum (extended abstract). In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of ConCur '90*, number 458 in LNCS, pages 279–297. Springer Verlag, 1990.

[vG94] Hans van Gageldonk. The Asynchronous Move Machine: Verification using CCS. Master's thesis, University of Eindhoven, August 1994.

[vT93] John P. van Tassel. *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant.* PhD thesis, University of Cambridge, July 1993. Available as Technical Report 317.

[WE85] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective.* VLSI Systems. Addison-Wesley, Menlo Park, CA, 1985.

[Xin92] Liu Xinxin. *Specification and Decomposition in Concurrency.* PhD thesis, University of Aalborg, April 1992.

[YD92] K. Y. Yun and D. L. Dill. Automatic Synthesis of 3D Asynchronous Finite-State Machines. In *International Conference on Computer Aided Design, ICCAD-92*, pages 576–580, Los Alamitos, Calif., November 1992. IEEE Computer Science Press.