THE UNIVERSITY OF CALGARY

REASONING ABOUT ASYNCHRONOUS DESIGNS IN CCS

BY

YING LIU

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER

ENGINEERING

CALGARY, ALBERTA

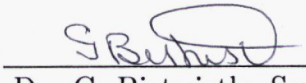
OCTOBER, 1992

ISBN 0-315-83194-4

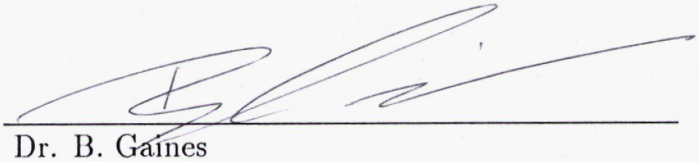
THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

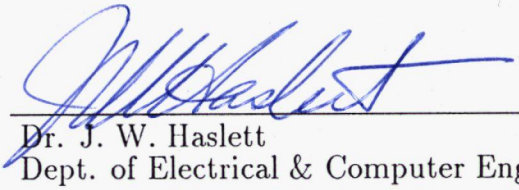
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "REASONING ABOUT ASYNCHRONOUS DESIGNS IN CCS", submitted by YING LIU in partial fulfillment of the requirements for the degree of Master of Science.



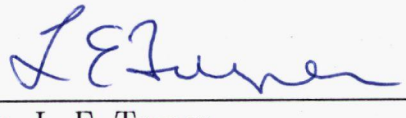
Dr. G. Birtwistle, Superviser & Chairman
Department of Computer Science



Dr. B. Gaines
Department of Computer Science



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



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

Date: October 21, 1992

Abstract

With VLSI technology advancing rapidly, synchronous designers are finding it difficult to distribute clock signals and maintain functionality as more circuitry is packed onto chips. Ways out of the dilemma are to raise the level of abstraction and to use simple and standard rules of composition. These are amongst the advantages offered by asynchronous design.

For years, designs have been "verified" via simulation at various levels. Attention is now being paid to formal methods which use induction proofs over regular structures in two steps and give full coverage over all input/output sequences.

This thesis brings the formal methods to bear on the asynchronous hardware design style. A parallel specification style is developed which scales well when the number of inputs to a system increases and a testing style based upon the modal μ-calculus is proposed to test the consequences of specifications. Several non-trivial designs are evaluated by this methodology.

Acknowledgements

I would like to express the deepest appreciation to my supervisor, Dr. Graham Birtwistle: for his willingness to accept me as one of his students when I was lost; for his patience in leading me into the world of formal methods which was completely new to me; for his enthusiastic encouragement and unfailing support all the time; and for his thoughtfulness and help in every small aspect. Without his support, the thesis would not have been possible.

Thanks to Shiu Kai Chin, Jo Ebergen, Doug Edwards, Brian Gaines, Brian Graham, Carl McCrosky, and Faron Moller who carefully read the first draft of my thesis. Their insights and suggestions helped me greatly in improving the quality of this thesis. Thanks also to Mantis Cheng, Ganesh Gopalkrishnan, Chris Tofts, and David Walker for beneficial discussions.

The Formal Method Group at the University of Calgary is a wonderful place to do research. John Aldwinckle's incisive understanding in modal logics and Ken Stevens' valuable experience in asynchronous design were a great help in carrying out this thesis research. Thanks also to Robin Cockett, Tom Fukushima, Mike Hermann, Donald Kuzenko, Bruce MacDonald, Rajagopal Nagarajan, Cameron Patterson, Todd Simpson, Konrad Slind, Dave Spooner and Sue Stodart who were always helpful and supportive.

This work could not have been completed without the support of the Alberta Microelectronics Center and the University of Calgary.

Finally, special thanks to my parents for their long lasting support and encouragement when I was with them and when I am half the earth away.

Contents

List of Figures

Chapter 1

Introduction

With the rapid advancement in chip fabrication technology, VLSI circuits are becoming smaller, denser and faster. To retain their market advantage, manufacturers like to keep the design-verification-fabrication-test window small. This is becoming increasingly difficult with the standard synchronous design style, and so researchers and manufacturers are re-examining old decisions to see if neglected design styles can be resurrected.

Two important requirements for a design style are composability and amenability to verification.

Composability means there are simple and consistent rules for joining circuits and sub-systems together. It may also mean that if all constituents of a design share a property, then so will the complete design (e.g. delay insensitivity, or strong inputs and strong outputs).

Traditionally, verification has been concerned with checking whether an implementation conforms to its specification. More attention is now being paid to formal verification since the simulation of hardware systems is exponential (in time) in its inputs, whereas formal proofs (which can abstract n-bit values on buses to a single function and prove regular designs in two steps by induction) tend not to balloon in size as we go up the design hierarchy. With formal techniques, we may also check the consequences of specifications and answer such questions as "Can the system deadlock? Is it safe? Live?". It is hard to see how these checks can be done satisfactorily

using simulation.

The asynchronous design style has been neglected for a long time, perhaps mainly due to problems associated with producing hazard free designs and its extra logic requirements. But it has some positive features, amongst which we mention:

- Since asynchronous design islands can work at their own rates, we may join asynchronous components together easy in the knowledge that this will not disturb other components, provided that suitable interfacing standards are adhered to.

- Languages like CCS and CSP have been used to specify asynchronous designs. Since these languages have been given proper semantics, the way is now open to reason about the designs they represent. CCS has the added advantage of mechanized support.

1.1 Asynchronous Circuit Design Style

The major difference between synchronous design and asynchronous design is that asynchronous circuits do not use a global clock. However, with advances in VLSI technology, hard problems associated with synchronous design are arising [Sei80], e.g. power, routing, clock distribution, and clock skew.

These problems do not arise in asynchronous design, since attention is focussed upon letting components work at their own rate and finding standard ways of joining them together. Thus it seems worthwhile to re-examine the pros and cons of asynchronous design.

Pros

- **Simple Standard Interfacing.** Asynchronous design is quite different from conventional synchronous design in its signaling conventions. The signaling protocols of asynchronous systems only rely on the ordering of signals; they make no assumptions about signal speed.

- **Separation of Timing and Functionality.** With its simple and standard signaling protocols, an asynchronous system can be designed as a set of separate subsystems with standard communications amongst them. The partition of subsystems is based only on functionality; speed is a performance issue to be handled separately. Thus we avoid problems with clock distribution and clock skew which may become severe when we compose synchronous systems.

- **Composability.** Because asynchronous design enhances separating timing from functionality, large asynchronous systems can be composed of subsystems operating at widely different speeds, taking advantage of the maximum speed available from each of its subsystems. This leads to the most important feature of asynchronous design: composability. The composability of asynchronous design not only provides a simple way of building larger structures hierarchically, but also makes it easy for system upgrading when improved circuitry becomes available.

- **Testability.** As far as the correctness of a design is concerned, the composability of asynchronous design also makes it easy to test. Each subsystem is first tested independently. When the whole system is composed adhering to a

self-timed communication protocol, it is easy to test the system without even knowing the details in each subsystem.

- **Layout.** In synchronous design, more attention is paid to placement and routing in the effort to achieve a functionally correct circuit. This is no longer the case in asynchronous design, since delays caused by wire length only affect performance but have nothing to do with functionality.

- **System Performance.** Besides its advantages mentioned above, asynchronous design may be superior to synchronous design in system performance. Generally speaking, synchronous systems tend to reflect a worst-case behavior, while asynchronous systems tend to reflect an average-case behavior. This is because the clock frequency in a synchronous circuit has to be set according to the worst-case delay in any of its subsystems, while the asynchronous system starts a new computation immediately after the previous data has been computed and the new data is available. Hence, provided the handshaking circuitry is not too slow or cumbersome, an asynchronous system should run faster than the worst case.

Cons

- **Lack of experience.** For many years, removing hazards from asynchronous design has been considered very difficult, and most circuit designers have shied away from asynchronous design.

- **Basic Modules are Difficult to Design.** Designing basic asynchronous building blocks is very difficult. But, over the years, the problem has been

solved, and several libraries now exist [Bru87, Bru91b, MFR85]. In contrast, composing asynchronous systems from library components is very easy, so once a tried and trusted library is to hand, hierarchical design is straightforward.

- **Larger Circuit Area.** One of the negative aspects of asynchronous design is that an asynchronous circuit is usually larger than its synchronous counterpart due to its extra logic requirements. However, it is interesting to note that the new (synchronous) DEC Alpha chip [Com92] not only has 30% of its area devoted to clocking circuitry, but also consumes 30 watts (17 by the clocks and probably 5–10 by the pads!). Perhaps these old ideas need re-evaluating too.

- **Lack of Supporting CAD Systems.** A long history of designing synchronous designs has resulted in a large variety of CAD systems available for supporting synchronous design, e.g. VTI, Cadence, GDT and etc. Although some CAD systems for asynchronous design do exist [Bru91c, Mar90b] they are still prototypes and do not support verification. Individual verification tools [Dil89, Mol91] have recently been developed to support the verification of asynchronous designs. It is believed that these positives of asynchronous design are beginning to spur the development of CAD systems, and more support tools should be available in the near future.

In summary, the advantages of asynchronous design are well-matched with the advancement of VLSI technology. The asynchronous design style is thus expected to become more practical and useful when the circuit size keeps on increasing.

Techniques for the design of asynchronous circuits have been investigated and developed over the years. Special types of such circuits have been also proposed.

Among them, are *speed-independent* circuits and *delay-insensitive* circuits. A speed-independent circuit is informally defined as a circuit of which the correctness is insensitive to element delay; a delay-insensitive circuit is a circuit of which the correctness is insensitive to both element delay and wire delay.

1.2 Verifying Asynchronous Systems with CCS

1.2.1 Formal Verification of Asynchronous Circuits

Formal verification of asynchronous circuits proves that an implementation meets a specification of its intended behavior (for all acceptable inputs) by using some formal reasoning frameworks. It not only checks whether an implementation conforms to its corresponding specification, but also verifies consequences of specifications such as *deadlock, livelock, safety* and *liveness* which should be possessed by a design. Informally,

- deadlock means a system may evolve into a state from which no further action is possible.
- livelock means that a system may get into an internal loop and make no further progress (accept no further input signals and emit no further output signals).
- a safety property means that nothing bad will happen when a system operates, e.g. there may never be more than one bus master.
- a liveness property means that something good will eventually happen when a system operates, e.g. a processor completes each instruction in finite time.

If a specification doesn't have these desired properties, it is pointless to implement it. Modifications should be made on the specification before embarking on implementation until satisfactory results have been achieved.

Formal verification for asynchronous design is being developed as an alternative to simulation. According to the reasoning framework used, there are two general approaches to the formal verification of asynchronous design. One is to take an existing general-type reasoning framework which is powerful enough to model the behavior of asynchronous circuits, and use it to construct proofs for correctness. For example, being originally a framework to express and reason mathematics, High Order Logic and its associated proof system HOL [Gor88] are well-used in the verification of hardware systems [Joy88, BGS+90]. An alternative approach is to develop a special-type reasoning framework for the area of interest. Examples of this are process algebras such as CSP and CCS developed by Hoare [Hoa85] and Milner [Mil89] respectively for describing and reasoning about concurrent systems.

This thesis will focus on the application of CCS to the formal verification of asynchronous designs. CCS maps well onto delay insensitive asynchronous design and has some mechanized support as well.

1.2.2 Appropriateness of CCS

CCS (Calculus of Communicating Systems) is a process algebra for describing and reasoning about concurrent systems developed by Milner [Mil89]. In CCS, a concurrent system is described as a collection of interacting processes which sometimes proceed on their own and sometimes need to synchronize with others before they can carry on. CCS provides well-defined syntax and semantics for specifying processes,

together with a set of laws for reasoning about these processes and how they communicate with each other. The Concurrency Workbench (CWB) [Mol91] is a tool supporting CCS which provides a very powerful model checker to verify whether a system behaves as expected by using Hennessy-Milner Logic (HML) [HM80, HM85] and the modal μ-calculus [Koz83]. Thus, once a concurrent system is specified in CCS, it becomes possible to reason about its processes and verify the system to be correct.

The purpose of using CCS and CWB in the verification of asynchronous circuit design is to put the building of asynchronous circuits on a firm formal basis. The key advantage of CCS/CWB is that we can investigate the consequences of a design specification before embarking upon an implementation. Provided with suitable propositions based upon modal μ-calculus, the CWB model checker can be used to check the important characteristics of a concurrent system, such as deadlock free, livelock free, safety and liveness. Compared with the normal practice of circuit simulation, verification results proved by the CCS/CWB hold over all input sequences, while circuit simulation results are only valid for limited testing sequences.

The thesis concentrates upon this aspect of formal verification. We show how to:

1. give parallel specifications to complicated asynchronous systems

2. develop a number of useful macros for testing the consequences of specifications

3. apply them to a range of asynchronous designs

Based upon the agreed specification and the chosen implementation, CCS/CWB may then be applied to see whether this implementation faithfully conforms to the

specification. The implementation should be equivalent to its corresponding specification. If so it will hold all the properties held by the specification, and we can replace the cumbersome implementation by a compact specification when reasoning further up the hierarchy. The equivalence checking is not included in this thesis because when building designs by composition, we have to show that each element is fitted into an environment which respects its delay insensitivity. This checking is not directly supported by the CWB and proving equivalence by hand is very painstaking, time consuming and tedious. Its automation seems to be a suitable PhD topic.

1.3 Structure of the Thesis

This thesis is structured as follows:

Chapter 2 describes the CCS notation and the Concurrency workbench (CWB). The syntax and semantics of CCS are described with examples. The main types of agent equivalence in CCS are covered. A mechanized style of specifying and testing CCS agents in the CWB is developed with examples. The limitations of CCS are investigated.

Chapter 3 describes the tools available for testing the consequences of a CCS specification in CWB, which are Hennessy-Milner Logic (HML) and the modal μ-calculus. Useful, general-purpose macros for hardware verification are proposed based upon the modal μ-calculus.

Chapter 4 gives the CCS specification of a cell set for self-timed design. The CWB testing results are also given.

Chapter 5 develops a methodology for applying CCS/CWB to the design and test

of specifications of asynchronous circuits. The examples have been chosen to cover a reasonable design spectrum: Sutherland's micropipeline and Ebergen's stack are flow-through architectures, Martin's distributed arbiter is a token ring, Brunvand's adder module is the basis of a self-timed ALU, and Sutherland's Move Machine is a tiny yet useful processor.

Finally, Chapter 6 summarizes the thesis work and gives suggestions for further work.

1.4 Contributions of the Thesis

Although there are several examples of CCS applied to asynchronous systems [Bre90, BA91, LM86, Par85b, Par85a], this thesis relates one of the first attempts to apply the CCS process algebra to asynchronous hardware description. CCS is used to write parallel specifications of asynchronous hardware and the CWB is used to test the consequences of these specifications. CCS/CWB has proved to be a nice tool because of its succinctness, scalability and equational reasoning capability.

The work in this thesis is perhaps the first serious application of process logics to test hardware specifications. Useful macros based upon modal μ-calculus and especially tailored for hardware verification are proposed to reason about the consequences of asynchronous designs expressed in CCS. The automated model checking tool embedded in the CWB has proved to be very powerful and practical since specifications can be tested thoroughly before embarking on implementations.

Chapter 2

CCS Notation and Support Tools

In this chapter, the CCS notation and CWB support tool are described with examples. The CCS syntax and semantics are explained first, followed by various notions of process equivalence in CCS. Examples of using the CWB are also given. Finally, the limitations of CCS are discussed.

2.1 Syntax and Semantics of CCS

In CCS, systems are described in terms of *agents*. An agent (or process) is a system whose behavior consists of interleaved, discrete actions. An agent may perform zero, one, or any number of sequential actions. More complex agents may be described as compositions of smaller agents executing in parallel.

Associated with each CCS agent is a set of visible actions called its *sort* through which it interacts with its environment (informally, its i/o ports). Compound agents will usually have purely internal communication lines which are hidden (not visible). Agents may evolve in two ways:

1. by a single visible interaction with the environment,

2. by an invisible, internal handshake (a simultaneous communication between two agents with one agent performing an *output* action on a hidden line and the other performing a complementary *input* action on the same hidden line simultaneously).

The semantics supported by CCS is interleaved and not fully parallel.

2.1.1 Syntax of CCS

$$
\begin{array}{lll}
\mathrm{E} & ::= & \mathrm{Nil} \\
 & | & \mathrm{A} \qquad\qquad\qquad\qquad\qquad\qquad \textit{constant} \\
 & | & \alpha.\mathrm{E} \qquad\qquad\qquad\qquad\qquad\quad \textit{prefix} \\
 & | & \mathrm{E}_1 + \mathrm{E}_2 + \ldots + \mathrm{E}_n \qquad \textit{summation} \\
 & | & \mathrm{E}_1 \mid \mathrm{E}_2 \mid \ldots \mid \mathrm{E}_n \qquad \textit{composition} \\
 & | & \mathrm{E} \setminus \mathrm{L} \qquad\qquad\qquad\qquad \textit{restriction} \\
 & | & \mathrm{E}\,[\,\mathrm{f}\,] \qquad\qquad\qquad\qquad\quad \textit{relabeling}
\end{array}
$$

where

A $\in$ Const, some fixed infinite set of agent constants,

$\alpha \in$ Act, the set of actions,

L is a subset of Names, and

f is a relabeling function.

2.1.2 Semantics of CCS

We give a semantics for CCS by induction over the structure of agent expressions.

- **Nil.** *Nil* represents a process which can do nothing. There is no rule for *Nil* since it cannot evolve.

- **Constant definition.** The behaviour of the defined agent A ($A \stackrel{def}{=} E$) is that of its definition E as expressed by the rule **Con:**

$$\textbf{Con} \quad \frac{\mathrm{E} \xrightarrow{\alpha} \mathrm{E}'}{\mathrm{A} \xrightarrow{\alpha} \mathrm{E}'}$$

- **Prefix.** If α is an action and E an agent then $\alpha.E$ is an agent which is capable of performing action α and then behaving as the agent E. This is expressed by the rule **Act**:

$$\textbf{Act} \quad \frac{}{\alpha.\mathrm{E} \xrightarrow{\alpha} \mathrm{E}}$$

- **Summation.** If E_1 and E_2 are agents, then $E_1 + E_2$ is an agent which non-deterministically behaves either like E_1 or like E_2. This is expressed by the rules $\textbf{Sum}_1$ and $\textbf{Sum}_2$:

$$\textbf{Sum}_1 \quad \frac{\mathrm{E}_1 \xrightarrow{\alpha} \mathrm{E}_1'}{\mathrm{E}_1 + \mathrm{E}_2 \xrightarrow{\alpha} \mathrm{E}_1'} \qquad \textbf{Sum}_2 \quad \frac{\mathrm{E}_2 \xrightarrow{\alpha} \mathrm{E}_2'}{\mathrm{E}_1 + \mathrm{E}_2 \xrightarrow{\alpha} \mathrm{E}_2'}$$

- **Composition.** If E_1 and E_2 are agents, then $E_1 \mid E_2$ is an agent whose behaviour is such that each of E_1 and E_2 may act independently of the other. This is expressed by the rules $\textbf{Com}_1$, $\textbf{Com}_2$:

$$\textbf{Com}_1 \quad \frac{\mathrm{E}_1 \xrightarrow{\alpha} \mathrm{E}_1'}{\mathrm{E}_1 \mid \mathrm{E}_2 \xrightarrow{\alpha} \mathrm{E}_1' \mid \mathrm{E}_2} \qquad \textbf{Com}_2 \quad \frac{\mathrm{E}_2 \xrightarrow{\alpha} \mathrm{E}_2'}{\mathrm{E}_1 \mid \mathrm{E}_2 \xrightarrow{\alpha} \mathrm{E}_1 \mid \mathrm{E}_2'}$$

E_1 and E_2 may also together engage in a communication whenever they are able to perform complementary output and input actions. This is expressed by the rule $\textbf{Com}_3$:

$$\textbf{Com}_3 \quad \frac{\mathrm{E}_1 \xrightarrow{\alpha} \mathrm{E}_1' \quad \mathrm{E}_2 \xrightarrow{\overline{\alpha}} \mathrm{E}_2'}{\mathrm{E}_1 \mid \mathrm{E}_2 \xrightarrow{\tau} \mathrm{E}_1' \mid \mathrm{E}_2'}$$

The τ-action introduced in $\textbf{Com}_3$ represents the occurrence of a communication event between two agents internally with no externally-visible effect. This internal communication is synchronized by one agent producing an output action while the other agent produces a complementary input action.

- **Restriction.** If E is an agent and L is a set of labels, then $E \setminus L$ is an agent which behaves like E except that it cannot perform any of the actions (as well as the corresponding complementary actions) lying in L externally, although each pair of these complementary actions can be performed for communication internally. This is expressed by the rule **Res**:

$$\textbf{Res} \quad \frac{\mathrm{E} \xrightarrow{\alpha} \mathrm{E}'}{\mathrm{E} \setminus \mathrm{L} \xrightarrow{\alpha} \mathrm{E}' \setminus \mathrm{L}} \ (\alpha, \overline{\alpha} \notin L)$$

- **Relabeling.** If E is an agent and f is a relabeling function, then $E[f]$ is an agent which behaves like E except that the labels are relabeled as specified by the function f. This is expressed by the rule **Rel**:

$$\textbf{Rel} \quad \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

The above CCS operators have decreasing binding power in the following order:

Restriction and Relabeling > Prefix > Composition > Summation

With these operators, processes can be described succinctly:

Example 1:

$$Match \stackrel{def}{=} strike.Nil$$

This example describes the behaviour of a match which is initially capable of performing the action *strike*. Thereafter, no further activity can be engaged in since it evolves into the agent *Nil*.

Example 2:

$$Clock \stackrel{def}{=} tick.Clock$$

Explicit recursive definition is used to describe the behaviour of a clock which ticks forever by repeatedly substituting the defining expression *tick.Clock* for *Clock* in the left hand side.

Example 3:

$$C \stackrel{def}{=} a.b.'z.C + b.a.'z.C$$

This example uses the summation operator to show the non-deterministic choice between two possible action sequences of a C-element. The C-element is a widely used cell in asynchronous circuit design which serves as the AND function for events. Upon receiving a transition on a, the C-element evolves into the agent $b.'z.C$; upon receiving a transition on b, the C-element evolves into the agent $a.'z.C$. But if the C-element receives a transition on both a and on b, it evolves into one of the above agents non-deterministically.

Example 4:

$$
\begin{array}{lcl}
Sem & \stackrel{def}{=} & 'g.p.Sem \\
U1 & \stackrel{def}{=} & \mathsf{ncs}_1.g.\mathsf{cs}_1.'p.U1 \\
U2 & \stackrel{def}{=} & \mathsf{nsc}_2.g.\mathsf{cs}_2.'p.U2 \\
System & \stackrel{def}{=} & (Sem \mid U1 \mid U2) \setminus \{g,p\}
\end{array}
$$

This example uses the composition operator and the restriction operator to describe the competition between two users for one resource. The resource is protected by semaphore *Sem* under the competition amongst several users (two users here). Each user has a cyclic behaviour of a non-critical section ncs (not using the resource), and a critical section cs (using the resource). The winning user has sole access to the resource during its corresponding critical section named as cs_1 or cs_2 respectively. This is guaranteed by the users following the protocol:

1. receiving a g request to compete for the semaphore before entering the *critical section*; and

2. producing a $'p$ acknowledge to release the semaphore when the *critical section* has been completed.

Here, by restricting *Sem* | *U1* | *U2* with the set $\{g,p\}$, all the labels in this set and also their complementary labels are externally inaccessible.

We note here that CCS does not broadcast; only one of the two users can gain the semaphore during one recursive *Sem* cycle:

$$\begin{array}{lcccccccc} & & (& 'g.p.Sem & | & g.cs_1.'p.U_1 & | & g.cs_2.'p.U_2 &) \backslash\{g,\ p\} \\ & \xrightarrow{\tau} & (& p.Sem & | & cs_1.'p.U_1 & | & g.cs_2.'p.U_2 &) \backslash\{g,\ p\} \\ \text{OR} & \xrightarrow{\tau} & (& p.Sem & | & g.cs_1.'p.U_1 & | & cs_2.'p.U_2 &) \backslash\{g,\ p\} \end{array}$$

Thus, this system evolves by interleaving, one of its many possible action sequences is listed as follows:

$$\begin{array}{lcccccccc} & (& Sem & | & U_1 & | & U_2 &) \backslash\{g,\ p\} \\ \equiv & (& 'g.p.Sem & | & ncs_1.g.cs_1.'p.U_1 & | & ncs_2.g.cs_2.'p.U_2 &) \backslash\{g,\ p\} \\ \xrightarrow{ncs_1} & (& 'g.p.Sem & | & g.cs_1.'p.U_1 & | & ncs_2.g.cs_2.'p.U_2 &) \backslash\{g,\ p\} \\ \xrightarrow{\tau} & (& p.Sem & | & cs_1.'p.U_1 & | & ncs_2.g.cs_2.'p.U_2 &) \backslash\{g,\ p\} \\ \xrightarrow{ncs_2} & (& p.Sem & | & cs_1.'p.U_1 & | & g.cs_2.'p.U_2 &) \backslash\{g,\ p\} \\ \xrightarrow{cs_1} & (& p.Sem & | & 'p.U_1 & | & g.cs_2.'p.U_2 &) \backslash\{g,\ p\} \\ \xrightarrow{\tau} & (& Sem & | & U_1 & | & g.cs_2.'p.U_2 &) \backslash\{g,\ p\} \\ \equiv & (& 'g.p.Sem & | & ncs_1.g.cs_1.'p.U_1 & | & g.cs_2.'p.U_2 &) \backslash\{g,\ p\} \\ \xrightarrow{\tau} & (& p.Sem & | & ncs_1.g.cs_1.'p.U_1 & | & cs_2.'p.U_2 &) \backslash\{g,\ p\} \end{array}$$

2.2 Equivalence of Processes

Having described the syntax and semantics of CCS, we next consider the equivalence of processes in CCS. This is important since it is the basis for system verification

which is basically concerned with checking whether an implementation conforms to its specification.

In CCS, we have four main types of agent equivalence, namely trace equivalence, strong equivalence, observation equivalence, and observation congruence. In the following sections, we are going to explain all these equivalences one by one, and finally reach to the right one for system verification: observation congruence.

2.2.1 Trace Equivalence $\sim_t$

In Trace Equivalence ($\sim_t$), two agents are regarded as being equivalent precisely when according to the operational semantics they perform the same sequences of actions:

> Given agents P and Q and a sequence $s = <a_1, a_2, ..., a_n>$ of actions, we write
>
> $$P \xrightarrow{s} Q$$
>
> whenever for some agents $P_1, P_2, ..., P_{n-1}$ we have
>
> $$P \xrightarrow{a_1} P_1, P_1 \xrightarrow{a_2} P_2, ..., P_{n-1} \xrightarrow{a_n} Q$$
>
> We regard agents P and Q as equivalent precisely when for all sequences s of actions, for some Q, $P \xrightarrow{s} Q$ holds if and only if for some Q', $P' \xrightarrow{s} Q'$ holds.

However, $\sim_t$ is insufficiently discriminating as it can be seen from the following example.

Example 5:

$$P \stackrel{def}{=} a.(b.Nil + c.Nil)$$
$$P' \stackrel{def}{=} a.b.Nil + a.c.Nil$$

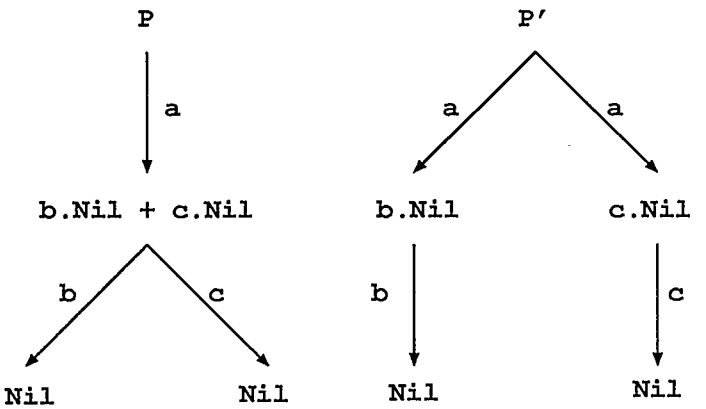


Figure 2.1: Trace Equivalence between Agent P and Agent P'

$$Traces(P) = \{\epsilon,\ a,\ ab,\ ac\ \} = Traces(P'),\ so\ P \sim_t P'$$

Though $P \sim_t P'$, they have different observable behaviours:

1. after an a, P reaches a state from which it can do either a b or a c;
2. after an a, P' reaches one of two states where:
   - it can do a b but not a c; or
   - it can do a c but not a b.

Hence we do not wish to regard these agents as equivalent. $\sim_t$ is thus rejected as a reasonable equivalence between CCS agents.

2.2.2 Strong Equivalence $\sim$

In order to define Strong Equivalence ($\sim$), it is necessary to introduce the notion of a Strong Bisimulation first.

A Strong Bisimulation $\mathcal{S}$ is a binary relation between two agents satisfying:

$\forall$ P:agent Q:agent and $\alpha \in Act$,

if $P \mathcal{S} Q$ then:

1. whenever $P \xrightarrow{\alpha} P'$, then for some Q'

   $Q \xrightarrow{\alpha} Q'$ and $P' \mathcal{S} Q'$

2. whenever $Q \xrightarrow{\alpha} Q'$, then for some P'

   $P \xrightarrow{\alpha} P'$ and $P' \mathcal{S} Q'$.

Then, Strong Equivalence is defined by:

$$P \sim Q \text{ iff } P \mathcal{S} Q \text{ for some strong bisimulation } \mathcal{S}$$

According to the above $\sim$ definition, if two agents are strongly equivalent, every action (including every τ-action of one agent) would have to be "matched" by an action (or a τ-action) of the other agent. Thus, the agents $a.Nil$ and $a.\tau.Nil$ would not be equivalent under this definition. But we would normally wish to regard them as equivalent.

We thus conclude that $\sim$ is not a reasonable equivalence between CCS agents either. Though it is useful, it makes too many distinctions.

2.2.3 Observation Equivalence $\approx$

Similar to the definition of Strong Equivalence, it is necessary to introduce the notion of a Weak Bisimulation first before we can give the definition of Observation Equivalence ($\approx$).

A Weak Bisimulation $\mathcal{S}$ is a binary relation between two agents satisfying:

$\forall$ P:agent Q:agent, and $\alpha \neq \tau$,

if $P \; \mathcal{S} \; Q$ then:

1. whenever $P \xrightarrow{\alpha} P'$, then for some Q'

   $Q \; (\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* \; Q'$ and $P' \; \mathcal{S} \; Q'$

2. whenever $P \xrightarrow{\tau} P'$, then for some Q'

   $Q \; (\xrightarrow{\tau})^* \; Q'$ and $P' \; \mathcal{S} \; Q'$

3. whenever $Q \xrightarrow{\alpha} Q'$, then for some P'

   $P \; (\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* \; P'$ and $P' \; \mathcal{S} \; Q'$.

4. whenever $Q \xrightarrow{\tau} Q'$, then for some P'

   $P \; (\xrightarrow{\tau})^* \; P'$ and $P' \; \mathcal{S} \; Q'$

Note: $(\xrightarrow{\tau})^*$ stands for zero or more τ transitions.

Then, Observation Equivalence is defined by:

$$P \approx Q \text{ iff } P \; \mathcal{S} \; Q \text{ for some weak bisimulation } \mathcal{S}$$

Thus, P and Q are observation equivalent iff, 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.

$\approx$ is *almost* a congruence relation of agents, i.e. in most cases, it is possible to substitute an agent in a complex system with an observationally equivalent agent and thereby obtain an observationally equivalent system. However, this is not always the case:

Example 6:

$$\begin{aligned} A &\stackrel{def}{=} \tau.a.Nil \\ B &\stackrel{def}{=} a.Nil \\ C &\stackrel{def}{=} l.Nil \end{aligned}$$

Though we have $A \approx B$, $A + C \approx B + C$ does not hold. This is because that agent $A + C$ may perform a τ-action to become $a.Nil$, but agent $B + C$ may not perform any sequence of τ-actions to become an agent observationally equivalent to $a.Nil$.

The above example illustrates that due to the pre-emptive power of τ-actions, $\approx$ is not a congruence for summation. This leads us to refine $\approx$ slightly to obtain our final notion of equivalence of CCS agents: Observation Congruence.

2.2.4 Observation Congruence =

The Observation Congruence (=) relation $P = Q$ holds if for all $\alpha \in \text{Act}$,

1. whenever $P \stackrel{\alpha}{\rightarrow} P'$, then for some Q',
   $Q\ (\stackrel{\tau}{\rightarrow})^* \stackrel{\alpha}{\rightarrow} (\stackrel{\tau}{\rightarrow})^* Q'$ and $P' \approx Q'$;
2. whenever $Q \stackrel{\alpha}{\rightarrow} Q'$, then for some P',
   $P\ (\stackrel{\tau}{\rightarrow})^* \stackrel{\alpha}{\rightarrow} (\stackrel{\tau}{\rightarrow})^* P'$ and $P' \approx Q'$

The only difference between $=$ and $\approx$ is that for $P = Q$, every action *even a τ-action* of P, must be matched by at least one corresponding action of Q, and vice versa. In this sense, $\tau.P$ is generally not $=$ to P.

According to the above definition, $=$ is a congruence relation of agents, it is a fully substitutive equivalence relation. This is the right equivalence we want to have between CCS agents, and further, for system verification.

2.3 The Concurrency Workbench

The Edinburgh Concurrency Workbench (CWB) [Mol91, CPB90] is an automated tool for analyzing concurrent systems expressed in CCS. With the CWB, concurrent systems can be specified as a hierarchy of subsystems composed of a series of CCS agents. After the specifications are input, the CWB can be used to check the specifications for such properties as *sort, sequence, states, deadlock* and various *equivalences.*

It is worth mentioning that the command we use to check equivalence between two agents is the Observation Equivalence (eq) instead of Observation Congruence (cong). If neither of the agents P and Q can initially perform a τ-action and $P \approx Q$, then we have $P = Q$.

An agent which cannot initially perform a τ-action is called a *stable* agent and whether or not an agent is *stable* can also be checked in the CWB. If τ-actions are not deliberately added, most CCS agents are stable agents. Hence we can use eq instead of cong in checking equivalence between agents.

Example 7:

```
Command: bi A

Agent: t.nil


Command: bi B

Agent: nil

Command: stable

Agent: A

**false


Command: stable

Agent: B

**true


Command: eq

Agent: A

Agent: B

**true


Command: cong

Agent: A

Agent: B

**false
```

With the additional τ-action introduced, agent A is not a stable agent like agent B. Thus, though $A \approx B$, we don't have $A = B$.

Example 8: (due to Parrow [Par85a])

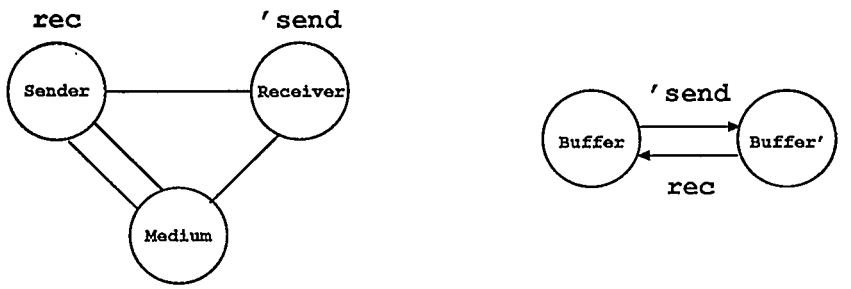


Figure 2.2: Behaviour of Two Simple Communication Protocols

$$
\begin{array}{lcl}
Sender & \stackrel{def}{=} & rec.'sm.Sender' \\
Sender' & \stackrel{def}{=} & ms.'sm.Sender' + rs.Sender \\
Medium & \stackrel{def}{=} & sm.Medium' \\
Medium' & \stackrel{def}{=} & ('mr.Medium + \tau.'ms.Medium) \\
Receiver & \stackrel{def}{=} & mr.'send.'rs.Receiver
\end{array}
$$

$$
\begin{array}{lcl}
Protocol & \stackrel{def}{=} & (Sender \mid Medium \mid Receiver)\backslash\{sm,ms,mr,rs\} \\
Buffer & \stackrel{def}{=} & rec.'send.Buffer
\end{array}
$$

- The agent *Protocol* provides a model of the behaviour of a simple communication protocol, which takes into account the possibility that a message might be lost during transmission between *Sender* and *Receiver*. Upon receiving a message with an input event *rec*, *Sender* transmits it to *Medium*. *Medium* may either transmit the

message to *Receiver*, or lose the message (modeled by a τ-action) in which case the message needs to be retransmitted. After receiving the message, *Receiver* produces an output event *'send*, and then sends an acknowledge signal directly to *Sender*. Only after it is acknowledged, may *Sender* receive another message.

- The agent *Buffer* provides a very high level description of the behaviour of a communication protocol.

With the bisimulation technology, we can construct a Weak Bisimulation relation between *Protocol* and *Buffer*:

$$
\begin{array}{lllrcccrll}
\mathcal{S} & = & \{ & ((\; S & | & M & | & R &)\backslash L, & B\,), \\
 & & & ((\; S' & | & M & | & 'rs.R &)\backslash L, & B\,), \\
 & & & ((\; 'sm.S' & | & M & | & R &)\backslash L, & 'send.B\,), \\
 & & & ((\; S' & | & M' & | & R &)\backslash L, & 'send.B\,), \\
 & & & ((\; S' & | & M & | & 'send.'rs.R &)\backslash L, & 'send.B\,), \\
 & & & ((\; S' & | & M & | & 'rs.R &)\backslash L, & 'send.B\,), \\
 & & \} & & & & & & &
\end{array}
$$

where, S stands for *Sender*, S' stands for *Sender'*, M stands for *Medium*, M' stands for *Medium'*, R stands for *Receiver*, B stands for *Buffer*, and L is defined as $\{sm, ms, mr, rs\}$.

Here, we have established that *Protocol* and *Buffer* are *Observation Equivalent* $\approx$. Further, we know that both *Protocol* and *Buffer* are *stable* agents, thus we have reached that *Protocol* and *Buffer* are *Observationally Congruent* $=$. All these facts can be proved using the CWB:

```
************************************************************
*                          On the CWB                      *
************************************************************
Command: bi Sender
Agent: rec.'sm.Sender'

Command: bi Sender'
Agent: ms.'sm.Sender' + rs.Sender

Command: bi Medium
Agent: sm.Medium'

Command: bi Medium'
Agent: 'mr.Medium + t.'ms.Medium

Command: bi Receiver
Agent: mr.'send.'rs.Receiver

Command: bi Protocol
Agent: ( Sender | Medium | Receiver )\{sm,ms,mr,rs}

Command: bi Buffer
Agent: rec.'send.Buffer
```

```
Command: sort Protocol
**{rec,'send}

Command: min Protocol
Save result in identifier: Protocol'
**Protocol' has 2 states.

Command: vs 4 Protocol
**=== rec 'send rec 'send ===>

Command: eq
Agent: Protocol
Agent: Buffer
**true

Command: stable
Agent: Protocol
**true

Command: stable
Agent: Buffer
**true

Command: cong
```

```
Agent: Protocol
Agent: Buffer
**true
```

The CWB also has a powerful model checker which can be used to check important characteristics of a concurrent system, such as deadlock free, livelock free, safety and liveness based upon the modal μ-calculus. This will be discussed in the following chapter.

2.4 Limitations of CCS

Although the CCS process algebra can be used to specify the structure of concurrent systems accurately and succinctly, it is not entirely satisfactory for several reasons:

1. Functionality is omitted. Note that Milner's text uses value passing CCS but this is not supported by the CWB nor by the modal μ-calculus.

2. Individual actions (here we exclude complementary handshake actions!) are not allowed to happen simultaneously.

3. It is incapable of describing timing properties, the probabilities and priorities of actions performed by the components of the system being modeled:

   (a) There is no explicit timing in CCS, all actions that can fire will fire in due course, but we do not know how long this may take.

   (b) The probability that an action is to be performed is undefined.

(c) No priority can be added to control the internal actions performed inside a concurrent system.

Note however that many varieties of CCS exist which partially remedy this list of defects. Amongst them are (i) the value-passing CCS [Bru91a], (ii) the *Synchronous Calculus of Communicating Systems* (SCCS) [Mil83a] (which makes it possible to have individual actions happen exactly at the same moment), (iii) the *Temporal Calculus of Communicating Systems* (TCCS) [MT89] (which can be used to model real time processes), (iv) the *Weighted Synchronous Calculus of Communicating Systems* (WSCCS) [Tof90a, Tof90b] (which allows probabilistic branching and is being used to formalise discrete event simulations), and (v) Milner's π-calculus [MPW89a, MPW89b, Mil91] (which merges full functionality with the sequencing of CCS).

2.5 Summary

In this chapter we have detailed the syntax and semantics of CCS and explained various notions of process equivalence. Although CCS has its limitations, it is a very compact notation with very clear semantics, and an amazing ratio of "range of application : size of language". Since this is my first foray into the complex world of asynchronous hardware, it was deemed best to tackle a limited range of problems with a small and clean tool so as best to understand the real problems and how to overcome them.

Chapter 3

Process Logics

In chapter 2, we described some notions of equivalence between CCS agents. With the aid of the CWB, we can check whether an implementation conforms to its specification. But equivalence checking may take a long time if the agents have many states. Thus it pays to see whether we can spot any differences between an implementation and a specification first before we attempt equivalence checking. Process logics provide the framework for such an enterprise. Even more importantly, process logics can be used to examine specifications for their consequences, e.g. deadlock, livelock, safety and liveness before we embark on implementation. In this chapter, we cover the process logics for CCS supported by the CWB, namely Hennessy-Milner Logic (HML) and the modal μ-calculus, together with examples.

3.1 Hennessy-Milner Logic

Hennessy-Milner Logic (HML) is a special type of modal logic, which uses labeled transition systems as a model. With HML, we can show whether an agent can carry out a certain trace by one move at each state. In this section, we describe HML syntax, the satisfaction relation and give some examples on the CWB.

3.1.1 Syntax of HML

Labeled transition systems have the form

$$(\mathcal{P}, \mathcal{A}, \{ \mathcal{T} \mid \alpha \in \mathcal{A} \})$$

where

$\mathcal{P}$ is a non-empty set of agents,

$\mathcal{A}$ is an action set,

$\mathcal{T}$ is the set of transition relations, $\xrightarrow{\alpha} \subseteq \mathcal{P} \times \mathcal{P}$ for each $\alpha \in \mathcal{A}$.

Example 1: simple vending machine

$$V \stackrel{def}{=} 1p.little.'collect.V + 2p.big.'collect.V$$

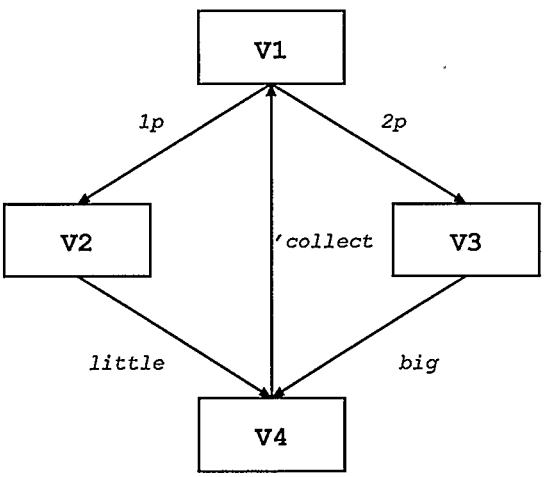


Figure 3.1: A Simple Vending Machine

we have,

$$\begin{array}{lll} \mathcal{P} & \stackrel{def}{=} & \{\ V1,\ V2,\ V3,\ V4\ \} \\ \mathcal{A} & \stackrel{def}{=} & \{\ 1p,\ 2p,\ little,\ big,\ 'collect\ \} \\ \mathcal{T} & \stackrel{def}{=} & \{\ V1 \stackrel{1p}{\rightarrow} V2,\ V1 \stackrel{2p}{\rightarrow} V3,\ V2 \stackrel{little}{\rightarrow} V4,\ V3 \stackrel{big}{\rightarrow} V4,\ V4 \stackrel{'collect}{\rightarrow} V1\ \} \end{array}$$

Let K range over subsets of an action set $\mathcal{A}$, the syntax of HML is defined,

$$\mathrm{A} ::= T \mid \neg A \mid A \wedge B \mid [\ K\ ]\ A$$

where

A is a formulae of HML,

T is the constant true, the only predefined atomic formula in HML,

$\neg$ is the negation of a formula,

$\wedge$ is the conjunction of two formulae,

$[\ K\ ]\ A$ means: A holds after every action in K.

Other common operators are derived:

$$F \quad \stackrel{def}{=} \quad \neg \mathrm{T} \qquad\qquad F \text{ is the constant formula false}$$

$$A \vee B \quad \stackrel{def}{=} \quad \neg(\neg A \wedge \neg B) \qquad \vee \text{ is the disjunction of two formulae}$$

$$< K > A \quad \stackrel{def}{=} \quad \neg[\, K \,] \neg A \qquad\qquad < K > \text{ is the dual of } [\, K \,]$$

3.1.2 Satisfaction of HML

For every formula A of HML, we interpret $E \models A$ as meaning "process $E \in \mathcal{P}$ satisfies the property A", and $E \not\models A$ as meaning "E fails to have the property A". The satisfaction relation $\models$ is defined inductively over the structure of HML formulae:

1. $E \models T \qquad \forall\, E$
2. $E \models \neg\, A \qquad \text{iff } E \not\models A$
3. $E \models A \wedge B \quad \text{iff } E \models A \wedge E \models B$
4. $E \models [\, K \,] A \quad \text{iff } \forall\, E' \in \mathcal{P}, \forall\, \alpha \in K.\ \text{if } E \stackrel{\alpha}{\rightarrow} E' \text{ then } E' \models A$

For the derived operators, we have,

5. $E \models F \qquad \text{iff } E \not\models T$
6. $E \models A \vee B \quad \text{iff } E \models A \vee E \models B$
7. $E \models < K > A \quad \text{iff } \exists\, E' \in \mathcal{P}, \exists\, \alpha \in K.\ E \stackrel{\alpha}{\rightarrow} E' \text{ and } E' \models A$

Their interpretations are:

1. Every process in $\mathcal{P}$ has the property T.
2. A process has property $\neg A$ when it fails to have property A.
3. A process has property $A \wedge B$ when it has both property A and property B.

4. A process satisfies $[\ K\ ]\ A$ if after every performance of any action in K, all the resulting processes have property A
5. Every process fails to have property T.
6. A process has property $A \vee B$ when it has either property A or property B.
7. A process satisfies $< K > A$ if it is possible to perform an action in K such that the resulting process has property A.

Given the HML satisfaction relations, we can prove whether an agent has some desired properties by the deduction rules.

Example 2:

$$A \stackrel{def}{=} a.(b.Nil + c.Nil)$$
$$B \stackrel{def}{=} a.b.Nil + a.c.Nil$$

$$E \stackrel{def}{=} < a >(< b >\mathrm{T} \wedge < c >\mathrm{T})$$

As mentioned in chapter 2, although agents A and B have the same trace, they are not considered to be equivalent CCS agents. We now show the difference by proving $A \models E$ but $B \not\models E$.

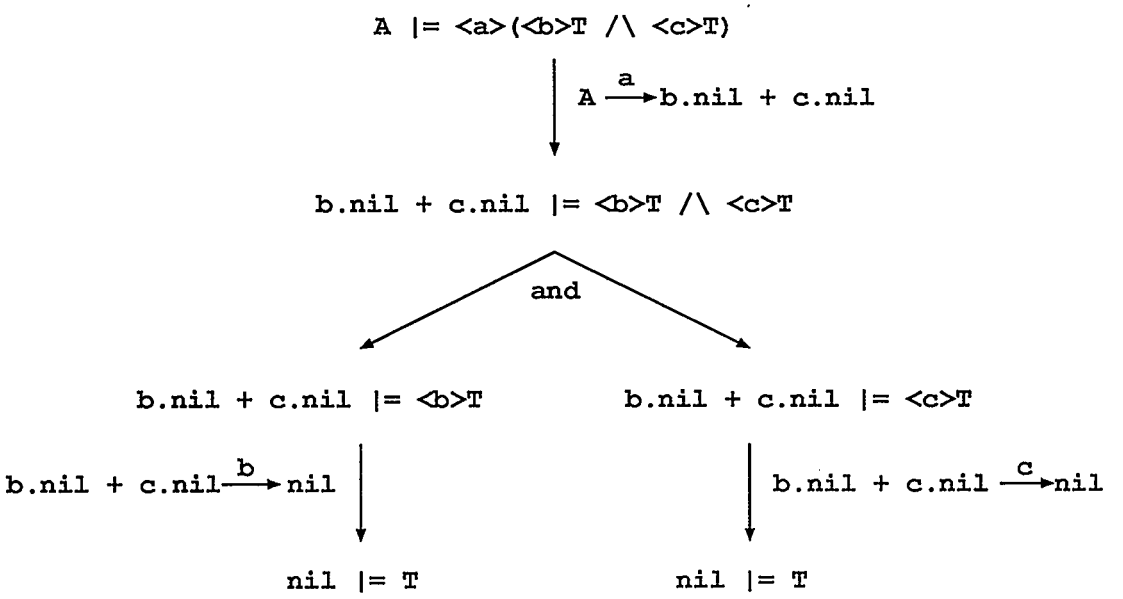


Figure 3.2: Proof of $A \models < a >(< b >\mathrm{T} \wedge < c >\mathrm{T})$

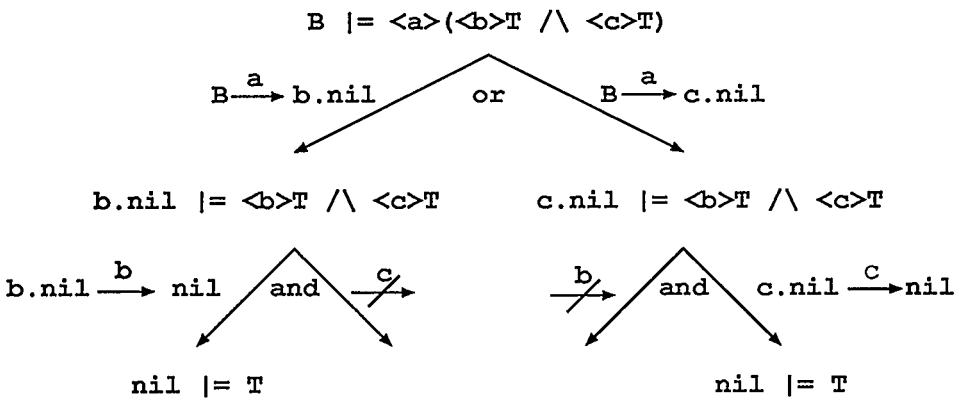


Figure 3.3: Proof of $B \not\models < a >(< b >\mathrm{T} \wedge < c >\mathrm{T})$

3.1.3 Expressing Properties in HML

Using the satisfaction relation, we can show whether an agent can carry out a certain trace one move at a time. This is realised by expressing agent properties at a certain state in HML and checking for correctness using the HML satisfaction. Several useful formulae for expressing agent properties and their interpretations are listed below:

$E \models [\, a \,] F$		E cannot do an a action
$E \models < a > T$		it is possible for E to do an a action
$E \models [\, - \,] F$		E cannot do any action (it is deadlocked)
$E \models < - > T$		E can do some action (it is live)
$E \models < - > T \wedge [\, - \, a \,] F$		E can do an a action and nothing else
$E \models \not< - a > < b > T$		E can do a non-a action then a b action
$E \models [\, a \,] < b > T$		after all a actions from E, one can do a b action
$E \models [\, a \,] T$		always true
$E \not\models < a > F$		always false

As an example of using HML, we show how the properties of the simple vending machine (see *Example 1*) are tested automatically in the workbench.

Example 3: $V \stackrel{def}{=} 1p.little.'collect.V + 2p.big.'collect.V$

Notice that on the CWB, $\vee$ is written |, and $\wedge$ is written &.

```
************************************************************
*                          On the CWB                      *
************************************************************
-- It is possible for V to do a 2p move.
Command: cp V
Propositions: <2p>T
```

```
**true

-- The only possible move after all 2p action is a big.
Command: cp V
Propositions: [2p](<->T & [-big]F)
**true

-- Initially, neither big nor little is possible for V.
Command: cp V
Propositions: [big,little]F
**true

-- After all 1p or 2p actions, a 1p move is impossible.
Command: cp V
Propositions: [1p,2p]<1p>T
**false

-- Starting from its initial state, the third move for V
-- can only be a 'collect.
Command: cp V
Propositions: <->T & [-](<->T & [-](<->T & [-'collect]F))
**true
```

Actually, there are two modal logics for CCS associated with the two transitions defined in conjunction with the strong and weak bisimulation. If τ-actions are to be accounted for (strong equivalence), the associated modalities are as mentioned above; if τ-actions are to be ignored (weak equivalence), the associated modalities are expressed as [[ ]] and << >>. We show this using one of the examples discussed in chapter 2.

Example 4:

$$
\begin{aligned}
Sem &\stackrel{def}{=} {}'g.p.Sem \\
U1 &\stackrel{def}{=} \mathsf{ncs}_1.\mathsf{g}.\mathsf{cs}_1.'p.U1 \\
U2 &\stackrel{def}{=} \mathsf{nsc}_2.\mathsf{g}.\mathsf{cs}_2.'p.U2 \\
\\
System &\stackrel{def}{=} (Sem \mid U1 \mid U2) \setminus \{g,p\}
\end{aligned}
$$

```
************************************************************
*                          On the CWB                      *
************************************************************
-- When tau-actions are taken into account, it is impossible
-- for System to do a cs1 immediately after all nsc1 moves.
Command: cp System
Proposition: [ncs1]<cs1>T
**false

-- When tau-actions are ignored, it is possible to do a cs1
-- immediately after all nsc1 moves.
```

```
Command: cp System

Proposition: [ncs1]<<cs1>>T

**true
```

3.2 Modal μ-calculus

HML is used to check whether an agent can carry out a certain trace by "asking questions one move at a time". It is suitable for checking the properties of simple agents with straight line or tree-like behaviors as shown in Figure 3.4.

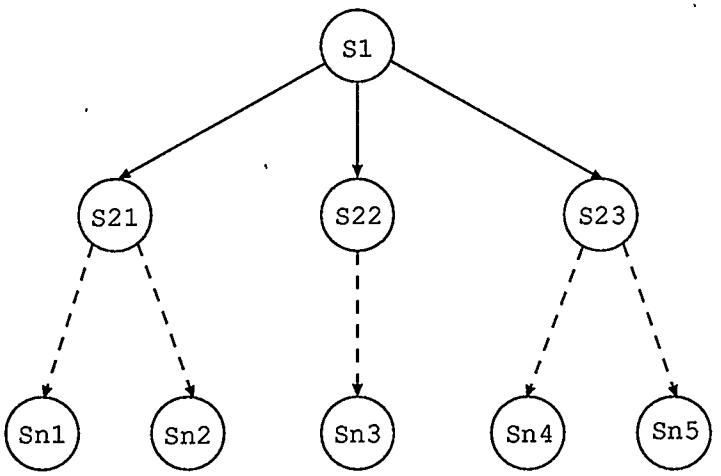


Figure 3.4: Simple Agent

Unfortunately, interesting agents usually have loops or iterations (as shown in Figure 3.5).

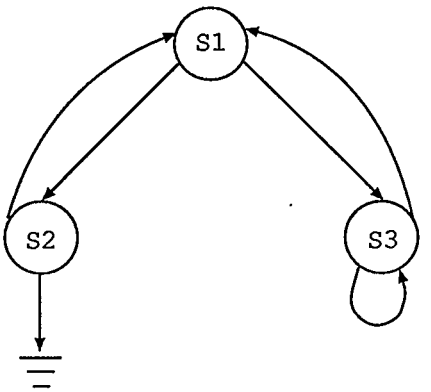


Figure 3.5: Agent with Loops between States

Here are some typical "threads" or paths of states of this agent starting from agent S_1.

- Self iteration, such as $(S_1 \rightarrow)\ S_3 \rightarrow S_3$ ....
- Iteration on a thread, such as $S_1 \rightarrow S_2 \rightarrow S_1 \rightarrow S_2$ ... or $S_1 \rightarrow S_3 \rightarrow S_1 \rightarrow S_3$ ...
- Deadlock at the end of a thread, such as $(S_1 \rightarrow)\ S_2 \rightarrow \mathit{nil}$.

Interesting propositions associated with recursive agents are inevitably themselves recursive and cannot be handled by HML. Fortunately, by adding just one construct (fix point definition) to HML, we get the modal μ-calculus [Koz83, SW91] which does permit recursive propositions:

$$\text{A} ::= T \mid \neg\ A \mid A_1 \wedge A_2 \mid [\ K\ ]\ A \mid \mathit{fix}(X.A)$$

3.2.1 Raw Modal μ-calculus

The behaviour of the vending machine V (figure 3.1) is cyclic (every third action is a *'collect*) and goes on forever. Clearly

$$V \models <->< - >< - >< ' \mathit{collect} > T$$

$$V \models <->< - >< ' \mathit{collect} >< - >< - >< ' \mathit{collect} > T$$

$$V \models <->< - >< ' \mathit{collect} >< - >< - >< ' \mathit{collect} >< - >< - >< ' \mathit{collect} > T$$

which suggests that V satisfies the equation

$$V = <->< - >< ' \mathit{collect} > V, \text{ or}$$

$$V = \mathbf{fix}\ (V. <->< - >< ' \mathit{collect} > V)$$

This is an example of a *fix point* equation. In general, fix point equations may have no solutions ($X = \neg X$) or several solutions. There is a simple syntactic check for the existence of at least one solution:

> *There will always be at least one solution provided that each fix point variable is within the scope of an even number of negations.*

Example 5: fix points

Consider the simple system specified below in Figure 3.6.

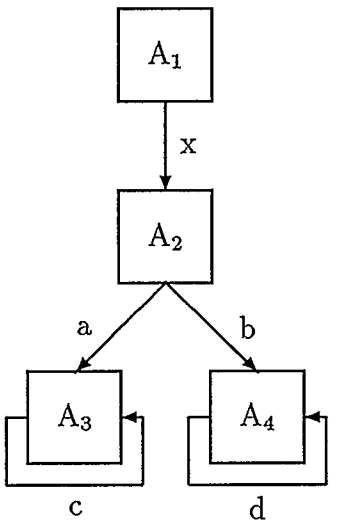


Figure 3.6: Fix Point Example

The system has 4 states (A_1, A_2, A_3, A_4), 5 labels (x, a, b, c, d) and the following transition relations:

Relations : (A_1, x, A_2)
: (A_2, a, A_3)
: (A_2, b, A_4)
: (A_3, c, A_3)
: (A_4, d, A_4)

1. [–]A. We interpret $[-]A$ over a labelled transition system by looking in the relations for each X such that $(X, \_, A)$ and EVERY action from X leads into A. In this case,

$$
\begin{aligned}
[-]A_1 &= \emptyset \\
[-]A_2 &= \mathrm{A}_1 \\
[-]A_3 &= \mathrm{A}_3 \\
[-]A_4 &= \mathrm{A}_4
\end{aligned}
$$

2. <->A. We interpret $< - > A$ by looking in the relations for each X such that $(X, \_, A)$ and SOME action from X leads into A.

$$
\begin{aligned}
< - > A_1 &= \emptyset \\
< - > A_2 &= A_1 \\
< - > A_3 &= A_2, A_3 \\
< - > A_4 &= A_2, A_4
\end{aligned}
$$

As an example, the fix points of $Y = [-]Y \vee < x > T$ are

$$
\begin{aligned}
\mathrm{Y} &= \{ \mathrm{A}_1 \} && \text{min fix point} \\
\mathrm{Y} &= \{ \mathrm{A}_1, \mathrm{A}_3 \} \\
\mathrm{Y} &= \{ \mathrm{A}_1, \mathrm{A}_4 \} \\
\mathrm{Y} &= \{ \mathrm{A}_1, \mathrm{A}_2, \mathrm{A}_3, \mathrm{A}_4 \} && \text{max fix point}
\end{aligned}
$$

Since an equation may have several fix points, natural questions to ask are: *Which are of most interest?* to which the answer is:

- the *maximumfixpoint* which includes everything except that which is necessarily false. It is used to express *safety.*
- the *minimumfixpoint* which includes only that which is necessarily true. It is used to express *liveness.*

and *How do we find them?* The algorithms for finding minimum and maximum fix-points are easy to explain:

Minimum fix point of $Y = FY$

1. start with $Y_0 = \emptyset$
2. compute $Y_1 = F\ Y_0$
3. compute $Y_{k+1} = F\ Y_k$
4. until $Y_{k+1} = Y_k$ (= Y the min fp of F)

E.g. min(Y. <d>T ∨ <->Y)

$$< d > T \qquad < - > Y_k$$

$$\begin{array}{lcl} Y_0 & = & \emptyset \\ Y_1 & = & \{ A_4 \} \quad \cup \quad \emptyset \quad = \quad \{ A_4 \} \\ Y_2 & = & \{ A_4 \} \quad \cup \quad \{ A_2, A_4 \} \quad = \quad \{ A_2, A_4 \} \\ Y_3 & = & \{ A_4 \} \quad \cup \quad \{ A_1, A_2, A_4 \} \quad = \quad \{ A_1, A_2, A_4 \} \\ Y_4 & = & \{ A_4 \} \quad \cup \quad \{ A_1, A_2, A_4 \} \quad = \quad \{ A_1, A_2, A_4 \} \end{array}$$

STOP

Maximum fix point of $Z = GZ$

1. start with $Z_0 = \mathcal{P}$, the set of all states
2. compute $Z_1 = G\ Z_0$
3. compute $Z_{k+1} = G\ Z_k$
4. until $Z_{k+1} = Z_k$ (= Z the max fp of G)

E.g. max(Z. [d]F ∧ [-]Z) is given as:

$$
\begin{array}{lcl@{\qquad}l@{\qquad}l@{\quad}l@{\quad}l}
 & & [d]F & & [-]Z_k & & \\
Z_0 & = & \{ A_1, A_2, A_3, A_4 \} & & & & \\
Z_1 & = & \{ A_1, A_2, A_3 \} & \cap & \{ A_1, A_3, A_4 \} & = & \{ A_1, A_3 \} \\
Z_2 & = & \{ A_1, A_2, A_3 \} & \cap & \{ A_3 \} & = & \{ A_3 \} \\
Z_3 & = & \{ A_1, A_2, A_3 \} & \cap & \{ A_3 \} & = & \{ A_3 \}
\end{array}
$$

STOP

Interestingly enough, these extremal fix points are related: if Y is the min fix point of Fy, and Z is the max fix point of $\neg F(\neg y)$ (the dual of Fy), then $\| Y \| = \mathcal{P} - \| Z \|$.

Above, we have shown that

$$
\begin{array}{lcll}
\text{min F Y} & = & < d > T \vee < - > Y & \text{is } \{ A_1, A_2, A_4 \} \\
\text{max G Z} & = & [d]F \wedge [-]Z & \text{is } \{ A_3 \}
\end{array}
$$

hence

$$
\begin{array}{rcl}
\neg F(\neg Z) & = & \neg(< d > T \vee < - > (\neg Z)) \\
 & = & (\neg < d > T) \wedge (\neg < - > \neg Z) \\
 & = & [d]F \wedge [-]Z \\
 & = & GZ
\end{array}
$$

$$\| Y \| = \{A_1, A_2, A_4\} = \mathcal{P} - \{A_3\} = \mathcal{P} - \| Z \|$$

3.2.2 A Collection of Macros

Here are some properties of the vending machine expressed in raw modal-μ:

```
** it is possible never to do a big

Command: cp V

Proposition: max(Z . <-big>Z)

**true
```

```
** it is always possible to do a big now and in the future
Command: cp V
Proposition: min(Z . <big>T | <->Z)
**true

** when a coin is inserted a 'collect must eventually happen
Command: cp V
Proposition:
max(Z.[1p,2p](min(Y. ONLY 'collect | [-'collect]Y)) & [-]Z)
**true
```

As can be seen, propositions expressed in the raw modal μ-calculus can be very hard to read. Further we often need to describe "properties within properties" which require nested fix point equations. But the modal μ-calculus is a very expressive logic and it has been shown that all the the provenly-useful temporal logic operators can be expressed within it [Dam90]. These operators are considerably more intuitively understandable than their raw modal μ equivalents. We therefore choose to present our arguments in terms of selected temporal operators.

Following Manna and Pnueli [MP92], two basic properties that we want are:

- P holds on every state reachable from state S. This is expressed by $S \models \Box\ P$ (read as "box P" or "always P").

  For example, referring to Figure 3.4, $S_1 \not\models \Box$ *live* states that not all the states reachable from S_1 are live.

- P holds on at least one thread from state S. This is expressed by $S \models \Diamond\ P$ (read as "diamond P" or "possible P").

  For example, $S_1 \models \Diamond$ *deadlock* states that it is possible to get deadlock on at least one thread from S_1.

We can combine these operators in two useful ways:[1]

- It is always possible to do P. Informally, if we don't do it this time, we might do it next time (when we loop back). This is expressed by $S \models \Box \Diamond P$.

  For example, $S_1 \models \Box \Diamond$ *deadlock* states that starting from S_1 wherever we move to, it is possible to deadlock (reach the state *nil*).

- It is possible to reach a set of states where P always holds. Informally, we can get possible stability after a warm up. This is expressed by $S \models \Diamond \Box P$.

  For example, $S_3 \models \Diamond \Box$ *deadlock* states that starting from S_1 it is possible to reach a state (S_2) from which we can always get deadlock.

Though $\Box \Diamond$ and $\Diamond \Box$ can be simply constructed with the combination of $\Box$ and $\Diamond$ defined above, there are macros which cannot be constructed directly from $\Box$ and $\Diamond$. One such useful operator is *eventually*: EV. Eventual properties are concerned with expressing that some desired properties eventually hold *whichever* thread is chosen. $S \models$ EV P states that either P holds at state S or state S has at least one derivative and EV P holds on every derivative. For example, in the simple vending-machine we have previously mentioned, no matter which thread we take, a *'collect* happens eventually.

[1] $\Box\ \Box = \Box$ and $\Diamond\ \Diamond = \Diamond$ and so are of no further interest.

The differences between $\Box$, $\Diamond$ and EV are summarized as:

- $S \models \Box\ P$ has P holding at S and at every state reachable from S;
- $S \models \mathsf{EV}\ P$ has P holding at S or on every thread reachable from S;
- $S \models \Diamond\ P$ has P holding at S or at one state reachable from S

In the following sections, we present examples of these basic macros. We close this section with their definitions in the modal μ-calculus and give some other macros found to be generally useful.

Safety Property

Safety properties are concerned with expressing that some undesirable property cannot happen. The basic operator is $\Box$. $S \models \Box\ P$ states that P holds on every state accessible from S. Usually we use $[...]F$ to state an undesirable property P.

Example 6: (due to Bradfield and Stirling [BS90])

This is a simple finite state system representing a road crossing a railway, in which *train* and *car* represent the approach of a train and a car respectively. *green* is the receipt of a green signal by the train, *tcross* is the train crossing and *'red* sets the lights red. *up* is the gates opening for the car, and *ccross* is the car crossing and *'down* closes the gates.

$$
\begin{array}{lll}
Rail & \stackrel{def}{=} & train.green.tcross.'red.Rail \\
Road & \stackrel{def}{=} & car.up.ccross.'down.Road \\
Signal & \stackrel{def}{=} & 'green.red.Signal + 'up.down.Signal \\
Crossing & \stackrel{def}{=} & (\ Road \mid Rail \mid Signal\ )\backslash\{green,red,up,down\}
\end{array}
$$

A crucial *safety* property of this system is that it is never possible for a *ccross* immediately after a *tcross*, and vice versa.

```
Command: cp Crossing
Proposition: BOX ([tcross][ccross]F)
**true

Command: cp Crossing
Proposition: BOX ([ccross][tcross]F)
**true
```

Instead, the car or the train should wait for a change of the crossing signal controlled by the invisible τ-actions within this system, hence the above claim results in *false* when weak modality is used in the model checking.

```
Command: cp Crossing
Proposition: BOX ([[tcross]][[ccross]]F)
**false

Command: cp Crossing
Proposition: BOX ([[ccross]][[tcross]]F)
**false
```

We can express *Conditional Safety* (if $\Box\ P$ holds only under a certain condition Q) by

$$\Box\ (Q \Rightarrow \Box\ P)$$

Example 7: (also due to Bradfield and Stirling [BS90])

The following system represents a process *Ticker* whose observable behavior is to perform a finite number of *'tick*s and then stops. But the process also has the feature that it may diverge by performing τ-actions indefinitely.

$$\begin{array}{lcll} Up_i & \stackrel{def}{=} & \tau.Up_{i+1} + Down_i & (i \geq 0) \\ Down_i & \stackrel{def}{=} & 'tick.Down_{i-1} & (i \geq 1) \\ Down_0 & \stackrel{def}{=} & nil & \\ Ticker & \stackrel{def}{=} & Up_0 & \end{array}$$

Since we cannot represent an infinite state machine on the CWB, we approximate it. Here is a *Ticker* of a "size 2",

```
Command: bi Up0
Agent: t.Up1 + Down0

Command: bi Up1
Agent: t.Up2 + Down1

Command: bi Up2
Agent: t.Up2 + Down2

Command: bi Down0
Agent: nil
```

```
Command: bi Down1
Agent: 'tick.Down0

Command: bi Down2
Agent: 'tick.Down1

Command: bi Ticker
Agent: Up0
```

Though the *Ticker* may perform τ-actions indefinitely, once we reach a state where a τ-action is impossible (after the first *'tick*), we can never do another τ.

```
Command: cp Ticker
Proposition: BOX([t]F => BOX [t]F)
**true
```

Liveness Property

Liveness properties are concerned with expressing that a system can possibly "escape". The basic operator is $\Diamond$. $S \models \Diamond P$ states that P holds on at least one thread accessible from S. For example, $\Diamond$ <->T states the *possible* liveness property of a system, and $\Diamond$ [-]F states the possibility of deadlock in a system.

```
Command: cp Ticker
Proposition: POSS <t>T
**true
```

```
Command: cp Ticker
Proposition: POSS <'tick>T
**true

Command: cp Ticker
Proposition: POSS <->T
**true

Command: cp Ticker
Proposition: POSS [-]F
**true
```

This is not the standard definition for *liveness*, but is close enough for our purposes in this introduction. Obviously, we can combine $\Box$, $\Diamond$ and other operators (e.g. $\neg$, $\wedge$, $\vee$) to express more elaborate notions of *liveness* quite easily.

With these definitions, the classes of liveness and safety properties are *dual.* The complement of a liveness property is a safety property, and vice versa:

$$\Box\ P = \neg\ \Diamond\ \neg\ P$$

$$\Diamond\ P = \neg\ \Box\ \neg\ P$$

Response Property

Response properties are concerned with expressing that some event happens infinitely many times. A basic response macro is formed by the operator $\Box\ \Diamond$. $S \models \Box\ \Diamond\ P$

states that one can always move to a state from which it is possible for P to hold on some following accessible threads.

For example, in the *Crossing* system specified above, it is always possible for a train or a car to cross.

```
Command: cp Crossing
Proposition: BOX POSS <tcross>T
**true

Command: cp Crossing
Proposition: BOX POSS <ccross>T
**true
```

An alternative form for response property is expressed as:

$$\Box\ (Q \Rightarrow \Diamond\ P)$$

where P is a guaranteed response to Q. For example, in the *Ticker* system specified above, the claim that if a move is possible (not deadlock) then a *'tick* is possible in the future is always true.

```
Command: cp Ticker
Proposition: BOX (<->T => POSS <'tick>T)
**true
```

Persistence Property

Persistence properties are concerned with expressing the possible stabilization of some state of a system. It allows an arbitrary delay until the stabilization occurs,

but require that once it occurs, it is continuously maintained. A basic persistence macro is based upon the operator $\Diamond\,\Box$. $S \models \Diamond\,\Box\,P$ states that it is possible to reach at least one state from which P holds on every following accessible thread.

We show this using the modified *Ticker* system which keeps on generating the warning signal *'flash* upon receiving an *error* message.

```
Command: bi Ticker_modify
Agent: Ticker + error.ERR

Command: bi ERR
Agent: 'flash.ERR

Command: cp Ticker
Proposition: POSS BOX [-'tick]F
**true
```

In many cases, the stabilization is triggered by a preceding event. This conditional persistence property is expressed as:

$$\Box\,(Q \Rightarrow \Diamond\,\Box\,P)$$

which specifies the eventual stabilization of P is caused by Q.

```
Command: cp Ticker_modify
Proposition: BOX (['flash]F => POSS BOX [-'tick]F)
**true
```

The classes of persistence and response properties are *dual.* The complement of a persistence property is a response property, and vice versa:

$$\neg \Box \Diamond P = \Diamond \Box \neg P$$

$$\neg \Diamond \Box P = \Box \Diamond \neg P$$

3.2.3 Defining Macros on the CWB

In the workbench, the basic property macros are defined as follows based upon the modal μ-calculus:

```
Command: bmi BOX P
Body: max(Z. P & [-]Z)

Command: bmi POSS P
Body: min(Z. P | <->Z)

Command: bmi EV P
Body: min(Z. P | ([-]Z & <->T))
```

3.2.4 Some Other Useful Macros

In this section, we list some additional useful property macros from which other interesting propositions can be constructed. We will use these macros without further ado in the following chapters.

- Only

  $S \models$ Only a states that it is possible to do an a action and no other action.

```
Command: bmi ONLY a
Body: (<a>T & [-a]F)
```

Hence, the only possible move at state S is performing an a.

- Only-Then

  $S \models$ Only-Then a P states that the only possible move at state S is performing an a, and we move to a state satisfying P after it is performed.

```
Command: bmi ONLY_THEN a P
Body: (ONLY a & [a]P)
```

- Must-Do

  $S \models$ Must-Do a states that starting from S we will eventually reach a state where a is the only possible move.

```
Command: bmi MUST_DO a
Body: EV (ONLY a)
```

Hence, eventually a must happen.

- Nec-For

  $S \models$ Nec-For a z states that without a, a z is impossible, hence a is necessary for a z. It does not guarantee that after an a is performed, we will definitely have a z move.

```
Command: bmi NEC_FOR a z
Body: max(Z. [z]F & [-a]Z)
```

We can further extend this macro to Nec-For′ defined as:

```
Command: bmi NEC_FOR' P z
Body: max(Z. [z]F & [-P]Z)
```

where P is an action list which may consist of any number of actions. Nec-For′ $P\ z$ states that at least one of the actions in the action list P is necessary for producing a z.

Notice that all the macros listed in this section are based upon the labeled modal μ-calculus instead of the unlabeled modal μ-calculus. Though using sets of labels adds nothing to the expressive power of the language, we do achieve flexibility and conciseness in expressing interesting properties.

Example 8: Vending machine revisited

- The inputs *1p* and *2p* of the vending-machine are mutually exclusive. Once *1p* happens, *2p* cannot happen until a *'collect* signaling the end of buying a chocolate has happened; and vice versa.

```
Command: bpi SV
Proposition:
max(SV. [1p](ONLY_THEN little (ONLY_THEN 'collect SV)) & \
        [2p](ONLY_THEN big    (ONLY_THEN 'collect SV)))

Command: cp V
```

```
Proposition: SV
**true
```

- A *'collect* must happen: eventually we reach a state where *'collect* is the only possible move.

```
Command: cp V
Proposition: MUST_DO 'collect
**true
```

- A *little* is necessary for producing a *'collect* after inserting *1p*; and a *big* is necessary for producing a *'collect* after inserting *2p*.

```
Command: cp V
Proposition: [1p] NEC_FOR little 'collect
**true

Command: cp V
Proposition: [2p] NEC_FOR big 'collect
**true
```

With the unlabeled and labeled property macros defined above, it becomes an easy task to construct various property macros rich enough to express the properties concerned with testing asynchronous hardware.

3.3 Summary

In this chapter we introduced the HML logic and the modal μ-calculus supported by the CWB. HML is used to show whether an agent can carry out a certain trace from a named state one move at a time. The modal μ-calculus is used to show the properties of recursive CCS agents over all states. After a short introduction to minimum and maximum fix points, we presented a number of macros, each written in modal-μ, which will be used in the rest of this thesis. The macros form a reasonably powerful basis for reasoning about hardware specifications, and are much more readable and intuitive than raw modal-μ expressions. We consider this raising of the level of abstraction to be a useful thesis contribution.

Chapter 4

Cell Library Specification

In this chapter we specify a number of small cells which have been suggested as basic library components by various researchers [Bru91c, Sut89, Ebe88]. The cells fall into three categories:

1. Trivial control path modules: Merge, C-element, Toggle, Wire and IWire, and Fork.

2. Non-trivial control path modules: Call, Arbiter and Mutual Exclusion, Select and Q-Select, and Join and Sequencer.

3. Data path modules: Enable, Register, Latch and Boolean Register.

In the following sections, we specify each of these modules in turn and present a variety of tests on their specifications.

4.1 The Trivial Control Path Modules

4.1.1 Merge

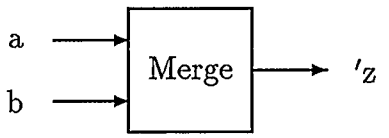


Figure 4.1: A Merge Module

Function

A Merge module serves as the "OR" function for transition signals: a transition on either input (a or b) causes a transition on the output ($'z$), but after a transition on a or b, a subsequent input event cannot occur until an output event $'z$ has been generated.

CCS specification

$$Merge = a.'z.Merge + b.'z.Merge$$

CWB testing

```
Command: sort Merge
**{a,b,'z}


Command: size Merge
**Merge has 2 states.


Command: vs 4 Merge
```

```
**=== a 'z a 'z ===>
**=== a 'z b 'z ===>
**=== b 'z a 'z ===>
**=== b 'z b 'z ===>
```

Behaviour verification

1. In its initial state, the Merge module is ready to accept an a or a b, but a $'z$ is impossible. Upon receiving either an a or a b, the module reaches a state where it is ready for producing a $'z$ (both a and b are impossible), and then evolves back to its initial state after $'z$ is produced.

```
Command: bpi SMab
Proposition: max(SMab. [a]SMz & [b]SMz & ['z]F)

Command: bpi SMz
Proposition: max(SMz. [a]F & [b]F & ['z]SMab)

Command: cp MERGE
Proposition: SMab
**true
```

2. After an input transition, the only possible move is to produce a $'z$.

```
Command: cp Merge
Proposition: (BOX [a](ONLY 'z)) & (BOX [b](ONLY 'z))
**true
```

4.1.2 C-Element

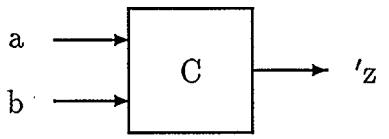


Figure 4.2: A C-Element Module

Function

A C-Element[1] serves as the "AND" function for transition signals: only after a transition has arrived on both of its inputs (a and b), will a transition be generated on the output ($'z$).

CCS specification

$$C = a.b.'z.C + b.a.'z.C$$

This style of specification does not extend well to C-elements with more than two inputs. For example, a 3−input C-element would be specified as:

$$C_3 = a.(b.c.'z.C_3 + c.b.'z.C_3) + b.(a.c.'z.C_3 + c.a.'z.C_3) + c.(a.b.'z.C_3 + b.a.'z.C_3)$$

[1]It is named the Join module in Brunvand's library.

The parallel style of specification examplified by

$$\begin{aligned} A &\stackrel{def}{=} a.'g.p.A \\ B &\stackrel{def}{=} b.'g.p.B \\ C &\stackrel{def}{=} c.'g.p.C \\ Z &\stackrel{def}{=} g.g.g.'z.'p.'p.'p.Z \\ \\ C_3 &\stackrel{def}{=} (A \mid B \mid C \mid Z) \setminus \{g,p\} \end{aligned}$$

is to be preferred as it is linear in the number of inputs.

CWB testing

```
Command: sort C

**{a,b,'z}


Command: size C

**C has 4 states.


Command: vs 6 C

**=== a b 'z a b 'z ===>

**=== a b 'z b a 'z ===>

**=== b a 'z a b 'z ===>

**=== b a 'z b a 'z ===>
```

Behaviour verification

1. In its initial state, the C-Element is ready to accept either an a or a b, but a $'z$ is impossible. Upon receiving an a, it waits for a b before producing a $'z$; upon receiving a b, it waits for an a before producing a $'z$.

```
Command: bpi SCab
Proposition: max(SCab . [a]SCb & [b]SCa & ['z]F)

Command: bpi SCa
Proposition: max(SCa  . [a]SCz & [b]F   & ['z]F)

Command: bpi SCb
Proposition: max(SCb  . [a]F   & [b]SCz & ['z]F)

Command: bpi SCz
Proposition: max(SCz  . [a]F   & [b]F   & ['z]SCab)

Command: cp C
Proposition: SCab
**true
```

2. After both inputs receive a transition, the only possible move is a $'z$.

```
Command: cp C
Proposition: (BOX [a][b](ONLY 'z)) & (BOX [b][a](ONLY 'z))
**true
```

4.1.3 Toggle

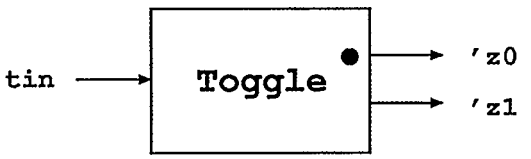


Figure 4.3: A Toggle Module

Function

A Toggle module routes an input transition (*tin*) alternatively to its two outputs ($'z_0$, $'z_1$). After initialisation, the first input transition will be routed to $'z_0$ and the subsequent input transition will be routed to $'z_1$. The output that receives the first transition starting from the initial state is marked with a dot as shown in Figure 4.3.

CCS specification

$$Toggle = tin.'z_0.tin.'z_1.Toggle$$

CWB testing

```
Command: sort Toggle
**{tin,'z0,'z1}

Command: size Toggle
**Toggle has 4 states.

Command: vs 8 Toggle
**=== tin 'z0 tin 'z1 tin 'z0 tin 'z1 ===>
```

Behaviour verification

The behaviour of Toggle module is straightforward: starting from its initial state, the Toggle module is ready for accepting a *tin* and producing a $'z_0$, then before it goes back to its initial state, the Toggle module should wait for another input *tin* to produce a $'z_1$.

```
Command: bpi ST
Proposition:
max(ST. [tin](ONLY_THEN 'z0 (ONLY_THEN tin (ONLY_THEN 'z1 ST))))

Command: cp Toggle
Proposition: ST
**true
```

4.1.4 Wire and IWire

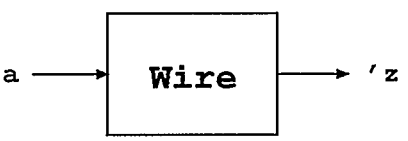


Figure 4.4: Wire (and IWire) Module

Function

A Wire module produces an output transition ($'z$) upon receiving a transition on its input. An IWire module fires $'z$ first before receiving any input transitions and then behaves as a Wire.

CCS specification

$$Wire \stackrel{def}{=} a.'z.Wire$$

$$IWire \stackrel{def}{=} 'z.a.IWire$$

The relationship between Wire and IWire is :

$$IWire \stackrel{def}{=} 'z.Wire$$

4.1.5 Fork

Figure 4.5: A Fork Module

Function

A Fork module steers an input transition (a) to both of its outputs ($'b$ and $'c$).

CCS specification

$$Fork \stackrel{def}{=} a.('b.'c.Fork + 'c.'b.Fork)$$

4.2 The Non-trivial Control Path Modules

4.2.1 Call

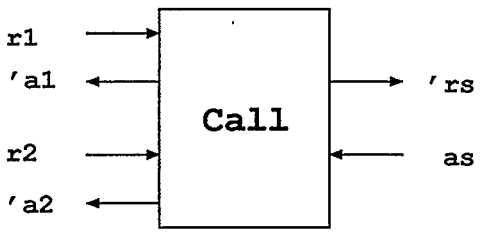


Figure 4.6: A Call module

Function

The transition Call Module implements the hardware equivalent of a subroutine call. After a transition on either of the two request lines *r1* or *r2*, the Call module starts a *subroutine* process with the *'rs* signal. When the subprocess completes and acknowledges with *as*, the Call module acknowledges the appropriate client on *'a1* or *'a2*. A full request-acknowledge transaction must be completed before either side may request again and the input request signals must be mutually exclusive.

CCS specification

$$Call = r1.'rs.as.'a1.Call + r2.'rs.as.'a2.Call$$

CWB testing

```
Command: sort Call

**{as,r1,r2,'a1,'a2,'rs}
```

```
Command: size Call
**CALL has 7 states.

Command: vs 8 CALL
**=== r1 'rs as 'a1 r1 'rs as 'a1 ===>
**=== r1 'rs as 'a1 r2 'rs as 'a2 ===>
**=== r2 'rs as 'a2 r1 'rs as 'a1 ===>
**=== r2 'rs as 'a2 r2 'rs as 'a2 ===>
```

Behaviour verification

1. The input requests *r1* and *r2* of the Call module are mutually exclusive. Once *r1* happens, *r2* cannot happen until the corresponding done event *'a1* has been generated; once *r2* happens, *r1* cannot happen until the corresponding done event *'a2* has been generated.

```
Command: bpi SCall
Proposition:
max(SCall.                                                    \
[r1](ONLY_THEN 'rs (ONLY_THEN as (ONLY_THEN 'a1 SCall)))  & \
[r2](ONLY_THEN 'rs (ONLY_THEN as (ONLY_THEN 'a2 SCALL))))

Command: cp Call
Proposition: SCall
**true
```

2. The input requests *r1* and *r2* of the Call module are mutually exclusive.

```
Command: cp Call
Proposition: (BOX [r1][r2]F) & (BOX [r2][r1]F)
**true
```

4.2.2 Arbiter and Mutual Exclusion

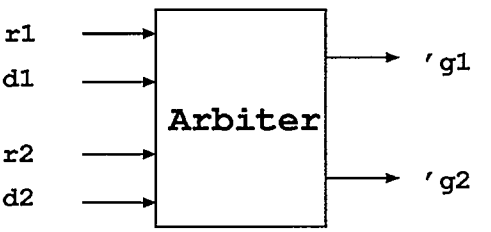


Figure 4.7: An Arbiter Module

Function

A two-way transition Arbiter (RGD arbiter) guarantees the mutually exclusive access to a resource of two independent users. If only one of the users' requests ($'r_1$ *or* $'r_2$), access is granted promptly ($'g_1$ or $'g_2$). If both users request, the Arbiter will grant access to only one of the two users. Whichever user receives the grant enters its critical section, and tells the Arbiter after desired actions have been performed by the *done*-transition (d_1 or d_2) which allows the Arbiter to grant access to the next requester.

CCS specification

The specification of arbiter module is a typical parallel specification. Two users contend for one resource, but the access is only granted to one of the two users. This

is guaranteed by an agent *Sem*:

$$
\begin{array}{lcccccc}
U_1 & \stackrel{def}{=} & r_1. & \clubsuit. & 'g_1.d_1. & \spadesuit. & U_1 \\
 & & & \Downarrow & & \Uparrow & \\
Sem & \stackrel{def}{=} & & \clubsuit. & & \spadesuit. & Sem \\
 & & & \Uparrow & & \Downarrow & \\
U_2 & \stackrel{def}{=} & r_2. & \clubsuit. & 'g_2.d_2. & \spadesuit. & U_2
\end{array}
$$

Once ♣ is captured by one of this two users, ♠ can only be returned by the same user in one *Sem* cycle.

Formally in CCS we have,

$$
\begin{array}{lcl}
U_1 & \stackrel{def}{=} & r_1.g.'g_1.d_1.'p.U_1 \\
U_2 & \stackrel{def}{=} & r_2.g.'g_2.d_2.'p.U_2 \\
Sem & \stackrel{def}{=} & 'g.p.Sem \\
 & & \\
Arbiter & \stackrel{def}{=} & (U_1 \mid U_2 \mid Sem) \setminus \{g,p\}
\end{array}
$$

CWB testing

```
Command: sort Arbiter

**{d1,d2,r1,r2,'g1,'g2}


Command: min Arbiter

Save result in identifier: A'

**A' has 12 states.
```

```
Command: pi A'

A' = A'0

where A'0  =  r1.A'12 + r1.A'2  + r2.A'6  + r2.A'13
  and A'10 = 'g2.A'11
  and A'11 =  d2.A'2  + d2.A'12
  and A'12 =   t.A'2  + r2.A'7  + r2.A'8  + r2.A'10 + 'g1.A'3
  and A'13 =  r1.A'10 +'g2.A'14
  and A'14 =  d2.A'0  + r1.A'11
  and A'2  =  r2.A'8  +'g1.A'3
  and A'3  =  d1.A'0  + r2.A'9
  and A'6  =   t.A'13 + r1.A'7  + r1.A'8  + r1.A'10 + 'g2.A'14
  and A'7  =   t.A'8  +  t.A'10 +'g1.A'9  +'g2.A'11
  and A'8  = 'g1.A'9
  and A'9  =  d1.A'6  + d1.A'13
end
```

From the above CWB testing on the Arbiter module, the benefits of parallel specification are clear compared with developing a specification state by state (A'). The latter is tedious and error prone (with much bookkeeping due to the large number of possible states) and becomes exponentially worse as the number of users of the Arbiter increases.

```
Command: vs 4 A'

=== r1 r2 'g1 d1 ===>
```

```
=== r1 'g1 d1 r1 ===>
=== r1 'g1 d1 r2 ===>
=== r1 'g1 r2 d1 ===>
=== r1 r2 'g2 d2 ===>
=== r2 r1 'g2 d2 ===>
=== r2 'g2 d2 r1 ===>
=== r2 'g2 d2 r2 ===>
=== r2 'g2 r1 d2 ===>
=== r2 r1 'g1 d1 ===>
```

Behaviour Verification

1. Requests from users may overlap.

```
Command: cp Arbiter'
Proposition: (BOX [r1][r2]F) | (BOX [r2][r1]F)
**false
```

2. But once a request is granted, it is necessary for the corresponding *done*-transition to be produced before another grant is allowed. For example, if the request from r_1 is granted ($'g_1$), neither $'g_1$ nor $'g_2$ can happen before d_1 is produced.

```
Command:  cp Arbiter'
Proposition: ['g1]((NEC_FOR d1 'g1) & (NEC_FOR d1 'g2))
**true
```

Mutual Exclusion

A Mutual Exclusion module grants permission ($'g_1$ or $'g_2$) to one of the two users competing for a resource (r_1 and r_2). It can be viewed as a simpler version of the arbiter module, with the difference that it resets the request line to its initial state after permission is granted (Return-to-Zero Signalling).

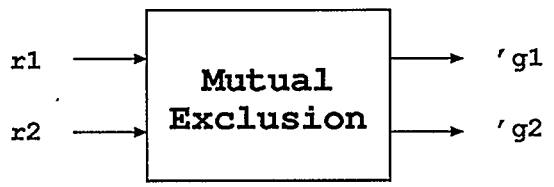


Figure 4.8: A Mutual Exclusion Module

It is specified as:

$$
\begin{aligned}
U_1 &\stackrel{def}{=} r_1.g.'g_1.r_1.'g_1.'p.U_1 \\
U_2 &\stackrel{def}{=} r_2.g.'g_2.r_2.'g_2.'p.U_2 \\
Sem &\stackrel{def}{=} 'g.p.Sem \\
\\
ME &\stackrel{def}{=} (U_1 \mid U_2 \mid Sem) \setminus \{g,p\}
\end{aligned}
$$

4.2.3 Select and Q-Select

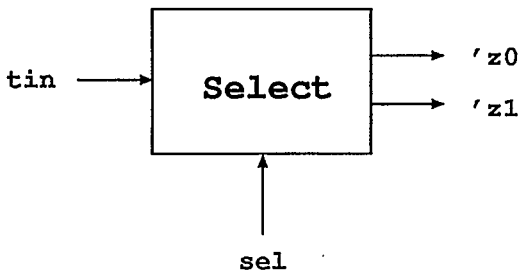


Figure 4.9: A Select Module

Function

A two-way transition Select module steers an input transition *tin* to either of its two outputs ($'z_0$,$'z_1$) depending on the value of a boolean data signal (*sel*). This Boolean data signal must be valid from before a transition occurs on the input until after a transition is generated at one of the outputs.

A Q-Select module (as shown in Figure 4.10) has the same function as that of the Select module except that it is delay-insensitive: there is no bundling constraint on *sel*. Thus transition on *sel* might happen after the input request *tin* has occurred but before an event on one of the outputs has been produced. A special circuit must be used to sample the changing of *sel* so that output is produced according to the sampled value on *sel*.

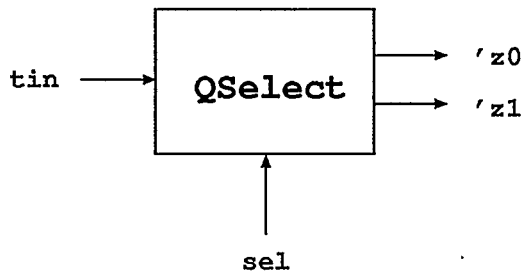


Figure 4.10: A Q-Select Module

CCS specification of Select

$$\begin{aligned} Select_0 &\stackrel{def}{=} sel.Select_1 + tin.'z_0.Select_0 \\ Select_1 &\stackrel{def}{=} sel.Select_0 + tin.'z_1.Select_1 \end{aligned}$$

$$Select \stackrel{def}{=} Select_0$$

CCS specification of Q-Select

$$\begin{aligned} QSelect_0 &\stackrel{def}{=} sel.QSelect_1 + 'zero.outzero.QSelect_0 \\ QSelect_1 &\stackrel{def}{=} sel.QSelect_0 + 'one.outone.QSelect_1 \\ TIN &\stackrel{def}{=} tin.(zero.'z_0.'outzero.TIN + one.'z_1.'outone.TIN) \end{aligned}$$

$$QSelect \stackrel{def}{=} (\ QSelect_0 \mid TIN\ ) \setminus \{\ zero, one, outzero, outone\}$$

CWB testing

```
Command: sort Select

**{sel,tin,'z0,'z1}

Command: sort QSelect
```

```
**{sel,tin,'z0,'z1}

Command: size Select
**Select has 4 states.

Command: min QSelect
Save result in identifier: QSelect'
**QSelect' has 6 states.

Command: vs 4 Select
=== sel sel sel sel ===>
=== sel sel sel tin ===>
=== sel sel tin 'z0 ===>
=== sel tin 'z1 sel ===>
=== sel tin 'z1 tin ===>
=== tin 'z0 sel sel ===>
=== tin 'z0 sel tin ===>
=== tin 'z0 tin 'z0 ===>

Command: vs 4 QSelect'
=== sel sel sel sel ===>
=== sel sel sel tin ===>
=== sel sel tin 'z0 ===>
=== sel sel tin sel ===>
```

```
=== sel tin 'z1 sel ===>
=== sel tin 'z1 tin ===>
=== sel tin sel 'z0 ===>
=== sel tin sel sel ===>
=== tin 'z0 sel sel ===>
=== tin 'z0 sel tin ===>
=== tin 'z0 tin 'z0 ===>
=== tin 'z0 tin sel ===>
=== tin sel 'z1 sel ===>
=== tin sel 'z1 tin ===>
=== tin sel sel 'z0 ===>
=== tin sel sel sel ===>
```

Behaviour verification

1. Initially, the Boolean data signal *sel* is *0*. The Select module is ready to either accept a transition on *sel* and evolve into a new state where the Boolean data signal *sel* is *1*, or accept a *tin* to produce a $'z_0$ before a *sel* can happen. When the Boolean data signal *sel* is *1*, the Select module may accept a transition on *sel* and evolve back into the state where the Boolean data signal *sel* is *0*, or accept a *tin* to produce a $'z_1$ before a *sel* can happen.

```
Command: bpi SS0
Proposition: max(SS0. [sel]SS1 & [tin](ONLY_THEN 'z0 SS0))

Command: bpi SS1
```

```
Proposition: max(SS1. [sel]SS0 & [tin](ONLY_THEN 'z1 SS1))

Command:  cp Select
Proposition: SS0
**true
```

2. No matter whether the Select module is in state $Select_0$ or in state $Select_1$, once an input transition *tin* happens, a corresponding output must occur before the boolean data signal *sel* can change its logic state.

```
Command: cp Select0
Proposition: [tin](NEC_FOR 'z0 sel)
**true

Command: cp Select1
Proposition: [tin](NEC_FOR 'z1 sel)
**true
```

3. We here show how a Q-Select module differs from a Select module: a transition on *sel* immediately after a *tin* is possible in the Q-Select, but impossible in the Select.

```
Command:  cp QSelect
Proposition:  BOX ([tin]<sel>T)
**true
```

```
Command: cp Select

Proposition: BOX ([tin]<sel>T)

**false
```

4.2.4 *2-by-1* Join and Sequencer

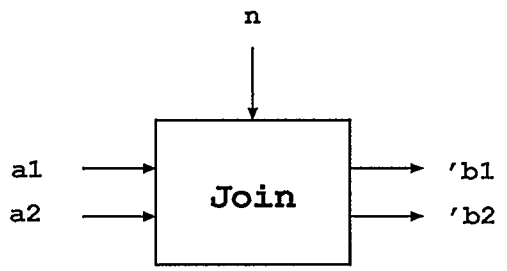


Figure 4.11: A *2-by-1* Join Module

Function

The *2-by-1* Join Module filters one (mutually exclusive) stream through (either $a_1.'b_1$ or $a_2.'b_2$) once permitted by a "go" signal on n.

CCS specification of Join

We might specify the Join module by one of

$$Join_1 \stackrel{def}{=} a_1.n.'b_1.Join_1 + a_2.n.'b_2.Join_1 + n.a_1.'b_1.Join_1 + n.a_2.'b_2.Join_1 \quad (1)$$

$$Join_2 \stackrel{def}{=} a_1.n.'b_1.Join_2 + a_2.n.'b_2.Join_2 + n.(a_1.'b_1.Join_2 + a_2.'b_2.Join_2) \quad (2)$$

but (1) is not satisfactory in that the action sequences are decided immediately after the first input event is received. (2) has the desired behaviour but we can make it

look neater by using a parallel specification. Again, think of its expansion to several users.

$$\begin{aligned} J &\stackrel{def}{=} a_1.g.'b_1.'p.J + a_2.g.'b_2.'p.J \\ N &\stackrel{def}{=} n.'g.p.N \end{aligned}$$

$$Join \stackrel{def}{=} (J \mid N) \setminus \{g,p\}$$

CWB testing

```
Command: sort Join

{a1,a2,n,'b1,'b2}


Command: min Join

Save result in identifier: Join'

Join' has 6 states.


Command: vs 3 Join'

=== a1 n 'b1 ===>

=== a2 n 'b2 ===>

=== n a1 'b1 ===>

=== n a2 'b2 ===>
```

Behaviour Verification

1. The two independent inputs a_1 and a_2 are mutually exclusive.

```
Command: cp Join'
```

```
Proposition: (BOX [a1][a2]F) & (BOX [a2][a1]F)
**true
```

2. After an a_1 and an n the next action must be a $'b_1$, and symmetrically for level 2.

```
Command: cp Join'
Proposition: (BOX [a1][n](ONLY 'b1)) & (BOX [n][a1](ONLY 'b1))
**true

Command: cp Join'
Proposition: (BOX [a2][n](ONLY 'b2)) & (BOX [n][a2](ONLY 'b2))
**true
```

Sequencer

A Sequencer module differs from the Join module in that its inputs a_1 and a_2 are not necessarily mutually exclusive. Because of this, the pair (a_1, n) and the pair (a_2, n) may contend for producing their corresponding output (b_1 or b_2).

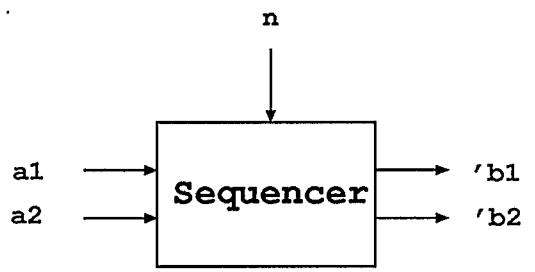


Figure 4.12: A Sequencer Module

The parallel specification of the Sequencer is:

$$
\begin{array}{lcl}
S_1 & \stackrel{def}{=} & a_1.g.'b_1.'p.S_1 \\
S_2 & \stackrel{def}{=} & a_2.g.'b_2.'p.S_2 \\
SN & \stackrel{def}{=} & n.'g.p.SN \\
 & & \\
Sequencer & \stackrel{def}{=} & (S_1 \mid S_2 \mid SN) \setminus \{g,p\}
\end{array}
$$

The behaviour of the Sequencer can also be verified on the CWB:

1. a_1 and a_2 are not necessarily mutually exclusive.

```
Command: cp Sequencer'
Proposition: (BOX [a1][a2]F) | (BOX [a2][a1]F)
**false
```

2. After a pair of desired inputs happens, it is not necessary for the corresponding output to be produced before the third input occurs (because the third input action can enter to contend for producing its corresponding output).

```
Command: cp Sequencer'

Proposition:

([a1][n] ~(NEC_FOR 'b1 a2)) & [a2][n] ~(NEC_FOR 'b2 a1))

**true
```

4.3 Data Path Modules

4.3.1 Enable

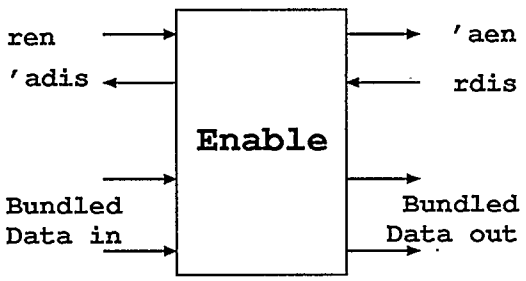


Figure 4.13: An Enable Module

Function

An Enable module is used to gate bundled data input onto a shared output bus. An enable request (*ren*) signifies that data is valid on the bundled data input and an enable acknowledge (*aen*) signifies the output data bundle is valid at all the receivers connected to the bus. When a disable request (*rdis*) is received, the data outputs are placed in a high impedance state and a disable acknowledge (*adis*) is generated.

CCS specification

$$Enable \stackrel{def}{=} ren.'aen.rdis.'adis.Enable$$

4.3.2 Register

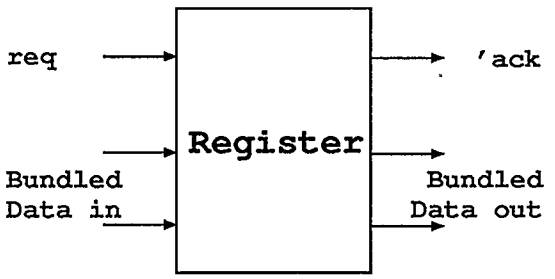


Figure 4.14: A Register Module

Function

An register module is used to store bundled data information with transition signaling as its control signals. When an input request (*req*) arrives at the register, the input data on the bundled data path are latched. An acknowledge transition (*'ack*) is generated after the data is latched, and the output data are then valid at all recipients of the data.

CCS specification

$$Register \stackrel{def}{=} req.'ack.Register$$

4.3.3 Latch

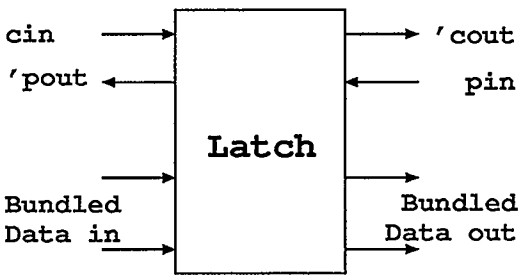


Figure 4.15: A Latch Module

Function

A Latch module has the similar function to that of a Register module, except with a slightly different set of control signals which has an explicit control over the transparent state and opaque state: a *cin* signal tells the Latch to capture the data and become opaque, and a *'cout* signal signals to the environment that the data has been latched; a *pin* signal tells the Latch to pass the data and become transparent, and a *'cout* signal signals that the data has been passed from the Latch to output.

CCS specification

$$Latch \stackrel{def}{=} cin.'cout.pin.'pout.Latch$$

4.3.4 Boolean Register

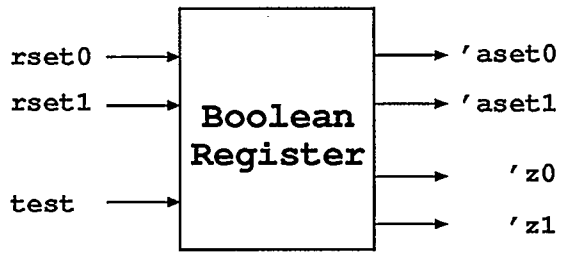


Figure 4.16: A Boolean Register Module

Function

A Boolean register is a single-bit register which uses transition signaling interfaces for changing the register's contents. A transition on the $rset_0$ wire sets the value of the register to zero and acknowledges on the $'aset_0$ line; a transition on the $rset_1$ wire sets the value of the register to one and acknowledges on the $'aset_1$ line. Uses of these interfaces must be mutually exclusive. The value of the register can also be tested: a transition on the test input ($test$) produces a transition on either outputs ($'z_0$ or $'z_1$) depending on the current value of the register.

CCS specification

$$BReg \stackrel{def}{=} BReg_0$$

$$BReg_0 \stackrel{def}{=} rset_0.'aset_0.BReg_0 + rset_1.'aset_1.BReg_1 + test.'z_0.BReg_0$$

$$BReg_1 \stackrel{def}{=} rset_0.'aset_0.BReg_0 + rset_1.'aset_1.BReg_1 + test.'z_1.BReg_1$$

CWB testing

```
Command: sort BReg
**{rset0,rset1,test,'aset0,'aset1,'z0,'z1}

Command: size BReg
**BReg has 6 states.

Command: vs 4 BReg
=== rset0 'aset0 rset0 'aset0 ===>
=== rset0 'aset0 rset1 'aset1 ===>
=== rset0 'aset0 test 'z0 ===>
=== rset1 'aset1 rset0 'aset0 ===>
=== rset1 'aset1 rset1 'aset1 ===>
=== rset1 'aset1 test 'z1 ===>
=== test 'z0 rset0 'aset0 ===>
=== test 'z0 rset1 'aset1 ===>
=== test 'z0 test 'z0 ===>
```

Behaviour verification

1. The interfaces for setting the Boolean register value to zero or one are mutually exclusive. It is necessary for a set to be acknowledged before the module can be set again.

```
Command: cp BReg
Proposition: (BOX [rset0][rset1]F) & (BOX [rset1][rset0]F)
**true
```

```
Command:  cp BReg
Proposition: [rset0] NEC_FOR 'aset0 rset1
**true

Command:  cp BReg
Proposition: [rset1] NEC_FOR 'aset1 rset0
**true
```

2. When a *test* signal is received, reporting the current value of the Boolean Register is the only possible move followed.

```
Command: cp BReg0
Proposition: [test] (ONLY 'z0)
**true

Command: cp BReg1
Proposition: [test] (ONLY 'z1)
**true
```

4.4 Summary

In this chapter, we have specified and tested a library of control path modules and data path modules for self-timed design. Where possible, parallel composition was used for generating neat and compact specifications. The key advantage of this specification style is that one can avoid developing specifications state by state (which is tricky, tedious and error prone). It is especially efficient for specifying multi-client modules (such as the multi-input C-element and the multi-user Arbiter) with several users, since a parallel specification stays linear in the number of clients. The next chapter examines various asynchronous designs which may be built from these components and finite state machines. As in the synchronous case, smart tools exist for the synthesis of asynchronous finite state machines (see [CDS93]).

Chapter 5

Design and Testing Circuit Specifications

In this chapter, we show applications of CCS and the CWB to designing and testing the specifications of asynchronous subsystems. The subsystems described cover a variety of designs, including flow through architectures (Sutherland's micropipeline [Sut89] and Ebergen's stack [EG91]), a token ring structure (Martin's distributed arbiter [Mar85]), an arithmetic unit (Brunvand's Carry-Completion Sensing Adder [Bru91c]), and a small processor, Sutherland's Move Machine.
The methodology adopted is summarized in the following steps:

Step 1: Design of Specification

Design a specification using parallel composition where possible.

Step 2: Test of Specification

Test the specification using process logics in the CWB. The specification should be proved to be deadlock free, livelock free, safe and also live. Other special properties may be shown to hold in order to gain confidence in the specification.

Step 3: Implementation

An implementation is given in terms of existing library modules.

The main contributions of this chapter lie in demonstrating a style of specification and in systematically applying our library of macros to examining the consequences of the specifications.

5.1 Sutherland's Micropipeline

Ivan Sutherland described a new VLSI design style called micropipelines [Sut89] in his 1989 ACM Turing Award lecture. The design of micropipelines is based upon the transition signaling conceptual framework, and is significantly different from the conventional VLSI design style of clock driven synchrony. As Sutherland concluded in his paper, complex systems can be built easily by hierarchical composition with this new micropipeline design style. Further, the composability offered by micropipelines and transition signaling provides a simple way to upgrade systems when new technologies become available.

5.1.1 Control Circuit for a Micropipeline

We first specify the control circuit for a micropipeline. The control part organizes the transfer of data from one micropipeline stage to another according to a simple request and acknowledge protocol.

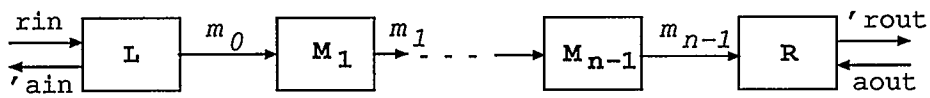


Figure 5.1: Specification of Control Circuit for a Micropipeline

The specification of a $n-$stage control circuit for micropipeline is constructed by having one cell L at its left side to deal with input requests and one cell R at its right side to deal with output requests. $n-1$ M cells between L and R provide the buffering spaces:

$$CCnspec = (L \mid M_1 \mid ... \mid M_{n-1} \mid R) \setminus \{m_0, m_1, ..., m_{n-1}\}$$

Cell L and Cell R provide the actual interface. Because they are operating in parallel, the actions belonging to L (*rin, 'ain*) and R (*'rout, aout*) can interleave.

The flexibility of this parallel specification style makes it trivial to specify an n—stage circuit: we merely adjust the number of M cells. As an example, here is the control circuit for a 4—stage micropipeline.

Step 1: Design of Specification

```
CC4spec = ( L                          \
          | M [ m0/left, m1/right ]    \
          | M [ m1/left, m2/right ]    \
          | M [ m2/left, m3/right ]    \
          | R [ m3/m0 ]                \
          ) \{m0,m1,m2,m3}

where
  L     = rin.'m0.'ain.L
  R     = m0.'rout.aout.R
  M     = left.'right.M

Command: sort CC4spec
**{aout,rin,'ain,'rout}

Command: min CC4spec
Save result in identifier: CC4spec'
```

```
**CC4spec' has 20 states.

Command: vs 4 CC4spec'
**=== rin 'ain rin 'ain ===>
**=== rin 'ain rin 'rout ===>
**=== rin 'ain 'rout aout ===>
**=== rin 'ain 'rout rin ===>
**=== rin 'rout aout 'ain ===>
**=== rin 'rout 'ain aout ===>
**=== rin 'rout 'ain rin ===>
```

Step 2: Test of Specification

1. Deadlock Free

   Deadlock means a state from which no further actions are possible, e.g., no part of a concurrent system is able to proceed. We test for each reachable node using □ operator.

```
Command: bpi Deadlock
Proposition: [-]F

Command: cp CC4spec'
Proposition: BOX (~Deadlock)
**true
```

2. Livelock Free

   Livelock means that the system may be stuck looping by τ actions in a cycle of one or more states moving from one to another without any visible actions on input and output ports. Thus no visible progress is made. It differs from deadlock in that it is always active, but nothing shows.

```
Command: bmi CYCLE_ON
Parameters: a
Body: POSS BOX <a>T

Command: bpi Livelock
Proposition: CYCLE_ON t

Command: cp CC4spec
Proposition: ~Livelock
**true
```

3. Safety

   Safety properties state that something bad never happens, that is, the system never enters an unacceptable state, such as deadlock. Different systems will have different classes of safety properties. Here, we consider one typical safety criterion which arises naturally from the above specification.

   - Absence of Unsolicited Response

Every acknowledgement is in response to a request. Once an acknowledgement is produced, another request is necessary for producing a further acknowledge.

```
Command: bmi Absence_of_Unsolicited_Response
Parameters: req ack
Body: (NEC_FOR req ack) & (BOX [ack](NEC_FOR req ack))
```

In the above specification, an input request *rin* is necessary for producing a corresponding acknowledge *'ain*. Once an *'ain* is produced, another *rin* is necessary for producing a further *'ain*.

```
Command: cp CC4spec'
Proposition: Absence_of_Unsolicited_Response rin 'ain
**true
```

And also, an output request *'rout* is necessary for accepting a corresponding acknowledge *aout*. Once an *aout* is accepted, another *'rout* is necessary for having a further *aout*.

```
Command: cp CC4spec'
Proposition: Absence_of_Unsolicited_Response 'rout aout
**true
```

4. Liveness

Liveness properties state that something good eventually does happen, i.e. it is always possible for the system to enter a desirable state. Here, we consider a few typical liveness properties for the above specification.

- Guaranteed Events

  An event which will always eventually be performed is called a guaranteed event.

```
Command: bmi Guaranteed_Event
Parameters: a
Body: BOX (EV <a>T)
```

  We here show that all the input and output actions in this specification are guaranteed events.

```
Command: cp CC4spec'
Proposition:
(Guaranteed_Event rin)   & (Guaranteed_Event 'ain) &  \
(Guaranteed_Event 'rout) & (Guaranteed_Event aout)
**true
```

Step 3: Implementation

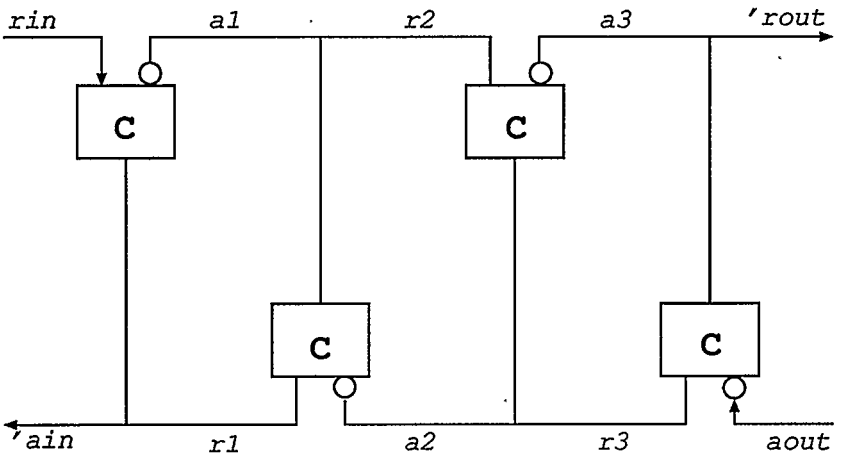


Figure 5.2: Implementation of the Control Circuit for a 4-stage Micropipeline

As shown in Figure 5.2, the C-element is the only element needed in implementing Sutherland's control circuit. Each stage of the control circuit uses a C-element with one of the inputs inverted to implement the following state rule:

If the predecessor and successor differ in state
Then copy predecessor's state
Else hold present state

We have already specified the C-element and tested its behaviour in Chapter 4. The behavior of a C-element with an inverted input is the same as that of the C-element except for its initial state. With one of the inputs inverted, it requires only a single event on the other input to trigger an output event. Once this has occurred, its behavior is the same as that of the normal C-element. Hence we can

adapt the specification for C-element in specifying Sutherland's implementation of control circuit but starting from state *Ca*.

```
CC4imp = ( Ca   [ rin/a,  a1/b,  z1/z ]  \
         | Ca   [  r1/a,  a2/b,  z2/z ]  \
         | Ca   [  r2/a,  a3/b,  z3/z ]  \
         | Ca   [  r3/a,aout/b,  z4/z ]  \
         | Fork [  z1/a, ain/b,  r1/c ]  \
         | Fork [  z2/a,  a1/b,  r2/c ]  \
         | Fork [  z3/a,  a2/b,  r3/c ]  \
         | Fork [  z4/a,  a3/b,rout/c ]  \
         ) \ { z1,z2,z3,z4,a1,a2,a3,r1,r2,r3}
where
  Ca   = a.'z.C
  C    = a.b.'z.C + b.a.'z.C
```

The above implementation only represents the interaction of a 4—stage micropipeline control circuit with no regard to its interface with the environment in which it operates. The *safety* property possessed by the specification implies that the implementation, in fact, operates in an environment which can be expressed in the following input and output constraints:

1. Input Constraint: After an input request *rin*, an acknowledge *ain* must occur before another *rin*.

2. Output Constraint: An output request *'rout* must occur before an acknowledge *aout* can be received.

5.1.2 FIFO Micropipeline

A FIFO micropipeline is simply a flow-through first-in-first-out buffer. The control for a single stage of FIFO has a request and acknowledge interface at both its input and ouput. The input request is used to signal that new data is available on the input data path, and the output request is used to indicate the next stage that new data is available at the output data path. The operation of each FIFO stage is to accept new input data, and then accept subsequent data only after its current data item has been taken by the next stage.

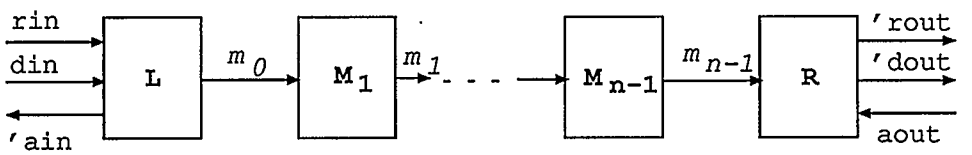


Figure 5.3: Specification of a FIFO Micropipeline

A $n-$stage FIFO is specified as having one cell L at its left side, one cell R at its right side and $n-1$ M cells between L and R:

$$FFnspec = (L \mid M_1 \mid ... \mid M_{n-1} \mid R) \setminus \{m_0, m_1, ..., m_{n-1}\}$$

In the following specification, we will see that M_1 is slightly different from M_i (i = 2, $n-1$) in that an additional semaphore s_1 is used to prevent further data from coming in when the first stage of FIFO is already occupied.

We here show a 4−stage FIFO micropipeline on the CWB.

Step 1: Design of Specification

```
FF4spec = ( L                              \
          | M1 [ m0/left, m1/right ]       \
          | M  [ m1/left, m2/right ]       \
          | M  [ m2/left, m3/right ]       \
          | R  [ m3/m0 ]                   \
          ) \ { s1,m0,m1,m2,m3 }

where

  L     = rin.s1.din.'m0.'ain.L

  M1    = left.'right.M1 + 's1.M1

  M     = left.'right.M

  R     = m0.'rout.'dout.aout.R


Command: sort FF4spec

**{aout,din,rin,'ain,'dout,'rout}


Command: min FF4spec

Save result in identifier: FF4spec'

**FF4spec' has 39 states.


Command: vs 6 FF4spec'

=== rin din 'ain rin din 'ain ===>

=== rin din 'ain rin din 'rout ===>

=== rin din 'ain rin 'rout din ===>
```

```
=== rin din 'ain rin 'rout 'dout ===>
=== rin din 'ain 'rout rin din ===>
=== rin din 'ain 'rout rin 'dout ===>
=== rin din 'ain 'rout 'dout aout ===>
=== rin din 'ain 'rout 'dout rin ===>
=== rin din 'rout 'ain rin din ===>
=== rin din 'rout 'ain rin 'dout ===>
=== rin din 'rout 'ain 'dout aout ===>
=== rin din 'rout 'ain 'dout rin ===>
=== rin din 'rout 'dout aout 'ain ===>
=== rin din 'rout 'dout 'ain aout ===>
=== rin din 'rout 'dout 'ain rin ===>
```

Step 2: Test of Specification

1. Deadlock Free

```
Command: cp FF4spec'
Proposition: BOX (~Deadlock)
**true
```

2. Livelock Free

```
Command: cp FF4spec
Proposition: ~Livelock
**true
```

3. Safety

- Absence of Unsolicited Response

  At the input port of a FIFO micropipeline, an input request *rin* is necessary for producing a corresponding acknowledge *'ain*. Once an *'ain* is produced, another *rin* is necessary for producing a further *'ain*. Similarly, *rin* is also necessary for *din*.

```
Command: cp FF4spec'
Proposition: Absence_of_Unsolicited_Response rin 'ain
**true

Command: cp FF4spec'
Proposition: Absence_of_Unsolicited_Response rin din
**true
```

  At the output port of a FIFO micropipeline, an output request *'rout* is necessary for accepting a corresponding acknowledge *aout*. Once an *aout* is accepted, another *'rout* is necessary for having a further *aout*. Similarly, *'rout* is also necessary for *dout*.

```
Command: cp FF4spec'
Proposition: Absence_of_Unsolicited_Response 'rout aout
**true

Command: cp FF4spec'
```

```
Proposition:  Absence_of_Unsolicited_Response 'rout 'dout
**true
```

4. Liveness

   - Guaranteed Events

     All the input and output actions in this specification are guaranteed events.

```
Command: cp FF4spec'
Proposition:
(Guaranteed_Event rin)   & (Guaranteed_Event 'ain) &  \
(Guaranteed_Event 'rout) & (Guaranteed_Event aout) &  \
(Guaranteed_Event din)   & (Guaranteed_Event  'dout)
**true
```

Step 3: Implementation

FIFO micropipeline can be constructed by using registers to hold data in the FIFO and the C-elements (with an inverted input) to control the transfer of data from one register to another.

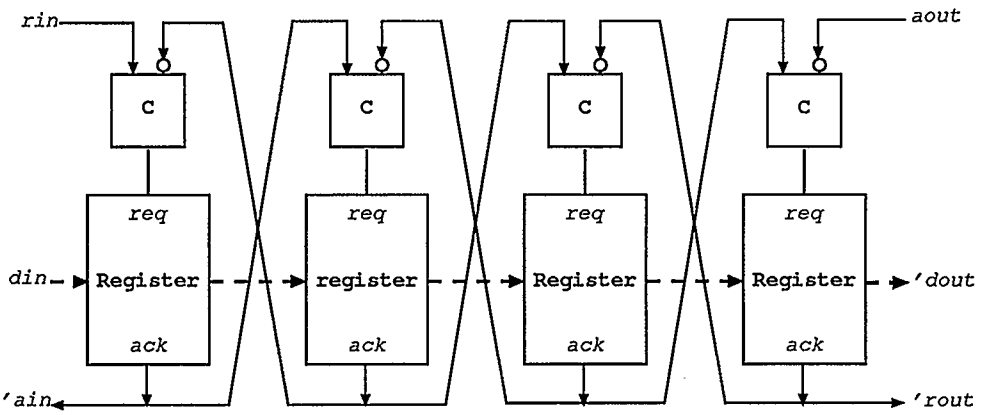


Figure 5.4: Implementation of a 4-stage FIFO Micropipeline

The register module used here in the above implementation is Brunvand's asynchronous register module with transition signaling as its control signals which we have specified in Chapter 4.

```
FF4imp =
( FF1imp [ q1/rout,p1/aout,d1/dout                          ] \
| FF1imp [ q1/rin,p1/ain,q2/rout,p2/aout,d1/din,d2/dout ] \
| FF1imp [ q2/rin,p2/ain,q3/rout,p3/aout,d2/din,d3/dout ] \
| FF1imp [ q3/rin,p3/ain,d3/din                          ] \
) \ { p1,p2,p3,q1,q2,q3,d1,d2,d3 }
where
  FF1imp = ( Ca       [ rin/a, aout/b, req/z ]  \
           | Register                           \
           | Fork     [ ack/a, ain/b, rout/c ]  \
           ) \ { req,ack }
```

5.2 Ebergen's Stack

Inspired by Sutherland's Micropipelines and Martin's Lazy Stack in [Mar90a], Ebergen presents a simple, fast design for an asynchronous stack [EG91]. The design of this asynchronous stack is split into two parts: a control part and a data part, where the control part dictates the transferring of data from one register stage to another.

A feature of this design is that its control part is delay insensitive, but the data part is not. Hence some extra delay constraints are to be satisfied in the realization of the data part, while the correctness of the control part is insensitive to any variations in the response time of the basic components and delays in the connection wires.

Figure 5.5 shows the interface between one stack cell and its left and right neighbors. Each stack has one private register and shares two registers with its neighbours. Any data transfer between two private registers has to go through their shared register. Pushing an item onto the stack results in shifting all data items one private register to the right, and popping an item from the stack shifts all items one private register to the left.

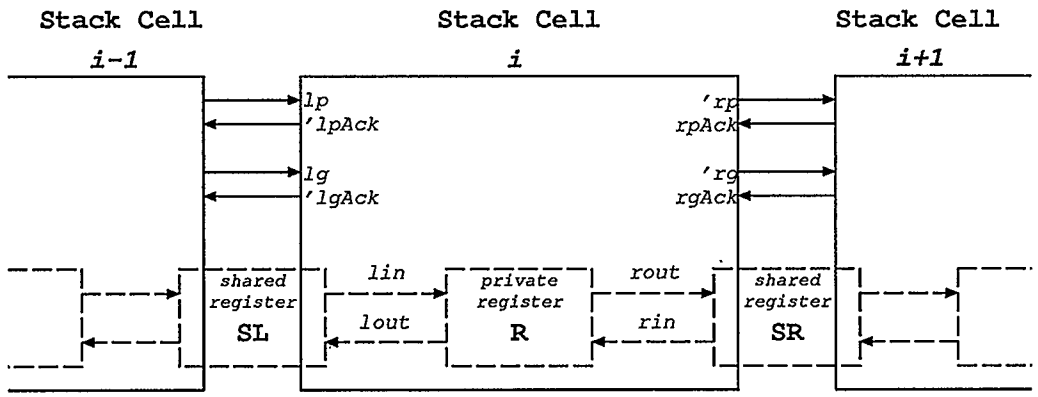


Figure 5.5: Control Part (solid lines) and Data Part (dashed lines) of a Stack Cell

The function of the control part is to control the pushing of data into a register and also the popping of data out of a register. This design is not concerned with whether the stack is full or empty; it is always possible to push values onto and pop values from the stack.

We specify the control part for a 2–stage stack. Information regarding private registers is hidden at this level of abstraction. Control parts with more stages can be accordingly expanded.

Step 1: Design of Specification

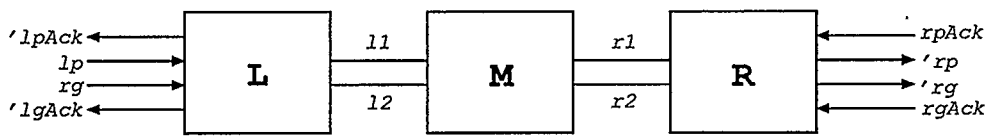


Figure 5.6: Specification of the Control Circuit for 2-stage Stack

There are two mutually exclusive processes in this design: request of *push* and *pop* from the previous stage at the left port; request of *push* and *pop* to the following stage at the right port. These two processes are related in that a request of *push* at the left port is the trigger event for a request of *push* at the right port; a request of *pop* at the left port is the trigger event for a request of *pop* at the right port.

Parallel specification methodology is adopted:

$$
\begin{array}{lcl}
L & \stackrel{def}{=} & lp.\ \clubsuit.\ 'lpAck.L \ + \ lg.\ \spadesuit.\ 'lgAck.L \\
 & & \Downarrow \qquad\qquad\qquad\qquad \Downarrow \\
R & \stackrel{def}{=} & \clubsuit.\ 'rp.rpAck.R \ + \ \spadesuit.\ 'rg.rgAck.R
\end{array}
$$

Formally in CCS we have,

```
S2spec = ( L | R ) \ { p,g }
where
  L    =  lp.p.'lpAck.L +  lg.g.'lgAck.L
  R    = 'p.'rp.rpAck.R + 'g.'rg.rgAck.R

Command: sort S2spec
**{lg,lp,rgAck,rpAck,'lgAck,'lpAck,'rg,'rp}

Command: min S2spec
Save result in identifier: S2spec'
S2spec' has 19 states.
```

Step 2: Test of Specification

1. Deadlock Free

```
Command: cp S2spec'
Proposition: BOX (~Deadlock)
**true
```

2. Livelock Free

```
Command: cp S2spec
Proposition: ~Livelock
**true
```

3. Safety

   - Mutual Exclusion

     Mutually exclusive processes in a program will never execute their critical sections at the same time.

     ```
     Command: bmi Mutual_Exclusion
     Parameters: a b
     Body: (BOX ([a][b]F)) & (BOX ([b][a]F))
     ```

     There are two mutually exclusive processes in the above specification. One is left port push *lp* and left port pop *lg*; the other is right port push *'rp* and right port pop *'rg*.

```
Command:  cp S2spec'
Proposition: Mutual_Exclusion lp lg
**true

Command:  cp S2spec'
Proposition: Mutual_Exclusion 'rp 'rg
**true
```

We can further show that once a process enters its critical section, it's only possible for its mutually exclusive counterpart to start after the critical section has been finished. For example, once *lp* happens, the corresponding acknowledgement *'lpAck* is necessary for starting *lg*. And similarly, after *'rp* happens, the corresponding acknowledgement *rpAck* is necessary for starting *'rg*.

```
Command: cp S2spec'
Proposition: BOX ([lp] NEC_FOR 'lpAck lg)
**true

Command: cp S2spec'
Proposition: BOX (['rp] NEC_FOR rpAck 'rg)
**true
```

- Absence of Unsolicited Response

  Again, we show that all the acknowledgements are in response to the corresponding input requests.

```
Command: cp S2spec'
Proposition: Absence_of_Unsolicited_Response lp 'lpAck
**true

Command: cp S2spec'
Proposition: Absence_of_Unsolicited_Response lg 'lgAck
**true

Command:  cp S2spec'
Proposition: Absence_of_Unsolicited_Response 'rp rpAck
**true

Command: cp S2spec'
Proposition: Absence_of_Unsolicited_Response 'rg rgAck
**true
```

4. Liveness

- Guaranteed Events

  Input actions *lp* and *lg* are Guaranteed events.

```
Command: cp S2spec'
Proposition: (Guaranteed_Event lp) & (Guaranteed_Event lg)
**true
```

  But this is not true for the rest of the input and output actions.

```
Command: cp S2spec'
Proposition:
(Guaranteed_Event 'lpAck) | (Guaranteed_Event 'lgAck) |  \
(Guaranteed_Event    'rp) | (Guaranteed_Event  rpAck) |  \
(Guaranteed_Event    'rg) | (Guaranteed_Event  rgAck)    \
**false
```

Actually, these input and output actions will eventually happen only after the pre-request (trigger event) has been produced. We call this an ensured response.

- Ensured Response

  An ensured response is a response which will eventually be produced after the corresponding request has occurred. It is defined as:

```
Command: bmi Ensured_Response
Parameters: req ack
Body:  BOX ([req] EV <ack>T)
```

Here, we show that once the pre-request *lp* happens, *'lpAck, 'rp* and *rpAck* must happen.

```
Command: cp S2spec'
Proposition: (Ensured_Response lp 'lpAck) &  \
             (Ensured_Response lp 'rp   ) &  \
             (Ensured_Response lp rpAck ) &
**true
```

And once the pre-request *lg* happens, *'lgAck, 'rg* and *rgAck* must happen.

```
Command: cp S2spec'
Proposition: (Ensured_Response lg 'lgAck) &  \
             (Ensured_Response lg 'rg   ) &  \
             (Ensured_Response lg rgAck ) &
**true
```

Step 3: Implementation

As shown in Figure 5.7, basic components *2-by-1* Join, Merge and IWire are the only elements necessary in implementing the control part of Ebergen's stack.

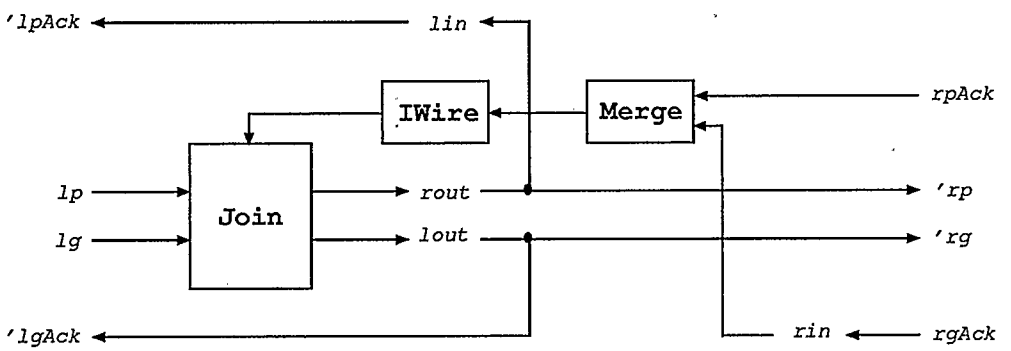


Figure 5.7: Implementation of the Control Part for a 1-stage Stack

With these modules, the control part for a 1−stage stack is implemented as:

```
S1imp = ( Join  [    lp/a1,   lg/a2,z2/n,rout/b1,lout/b2 ]  \
        | Fork  [  rout/a, lpAck/b, rp/c                  ]  \
        | Fork  [  lout/a, lgAck/b, rg/c                  ]  \
```

```
    | Merge [ rpAck/a, rgAck/b, z1/z                    ]  \
    | IWire [    z1/a,    z2/z                         ]  \
    ) \ { rout,lout,z1,z2 }
```

For the control part of a 2—stage stack, we have,

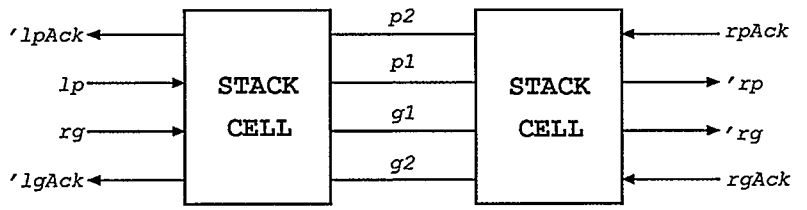


Figure 5.8: Implementation of the Control Part for a 2-stage Stack

```
S2imp = ( S1imp [ p1/rput,p2/rputAck,g1/rget,g2/rgetAck ]  \
        | S1imp [ p1/lput,p2/lputAck,g1/lget,g2/lgetAck ]  \
        ) \ { p1, p2, g1, g2 }
```

5.3 Martin's Distributed Arbiter

The basic idea in distributed arbiter is that users contend through separate nodes joined together by a token ring. When the token reaches a node at which there is a request, the token stays until the request is satisfied and then moves on; when the token reaches a node at which there is no request, the token moves on straight forward.

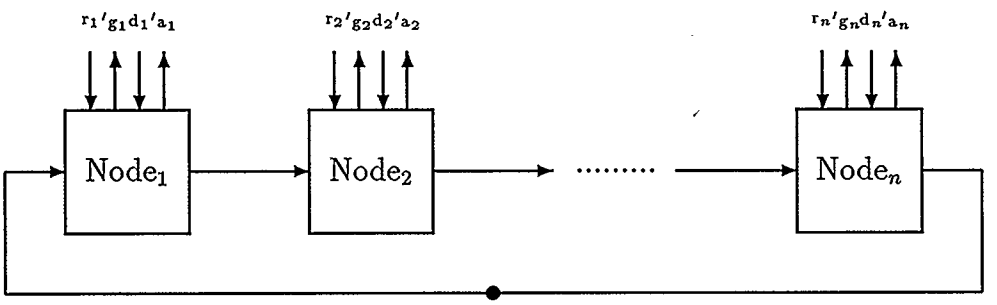


Figure 5.9: A Distributed Arbiter

The deficiency with this design is that the token is in motion even if there are no requests from the users. Because of this, livelock may happen.

Alain Martin suggested another distributed arbiter where the token does not cycle round the ring when there are no requests, but remains at the node where it last did some work until fetched.

Here we show how the node in Martin's distributed arbiter is specified.

Step 1: Design of Specification

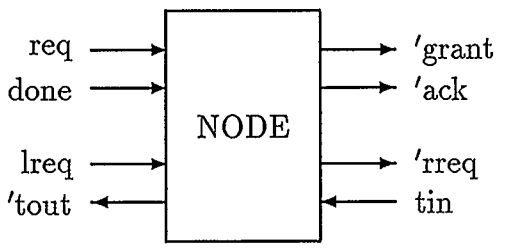


Figure 5.10: Node of Martin's Distributed Arbiter

We note that a node can either be initialised to have token or not have token. Hence, the possible behaviours per node are:

1. The node has the token:

   - a request is accepted:

     The firing sequence is *req.'grant.done.'ack*

   - the token is requested on the left and passed on:

     The firing sequence is *lreq.'tout*

2. The node does not have the token:

   - a request is accepted, the token is fetched on the right, then the request is granted:

     The firing sequence is *req.'rreq.tin.'grant.done.'ack*

   - the token is requested on the left, is fetched on the right, and is then passed on:

     The firing sequence is *lreq.'rreq.tin.'tout*

We design the specification of node in Martin's distributed arbiter by interfacing the user of the node *User* and the token of the node *Token* with a finite state machine which sorts out the current state of the NODE (the *User* has token, the *User* does not have token), the current request (from the *User* or from the *Token*) and then carries out the appropriate actions.

$$
\begin{array}{lcccccccccc}
User & \stackrel{def}{=} & & req. & \clubsuit. & & \heartsuit. & 'grant. & done. & \blacksquare. & 'ack. & User \\
 & & & & \Downarrow & & \Uparrow & & & \Updownarrow & & \\
FSM & \stackrel{def}{=} & & & \clubsuit. & fetch?. & \heartsuit. & & & \blacksquare. & & FSM \\
 & & + & & \spadesuit. & fetch?. & \diamondsuit & & & & & FSM \\
 & & & & \Uparrow & & \Updownarrow & & & & & \\
Token & \stackrel{def}{=} & & lreq. & \spadesuit. & & \diamondsuit. & 'tout. & & & & Token
\end{array}
$$

where FSM splits into

$$
\begin{array}{lcccccc}
State_0 & \stackrel{def}{=} & \clubsuit.'rreq.tin.\heartsuit.\blacksquare.State_1 & + & \spadesuit.'rreq.tin.\diamondsuit.State_0 \\
State_1 & \stackrel{def}{=} & \clubsuit.\heartsuit.\blacksquare.State_1 & + & \spadesuit.\diamondsuit.State_0
\end{array}
$$

in which there are five rendezvous points:

1. ♣ — a user request arrives. The *FSM* is woken up and will be either in state $State_0$ (have to fetch the token) or in state $State_1$ (already has the token).

2. ♡ — the node has the token and the user request may proceed. *User* is woken up and the *FSM* lies dormant until the transaction is completed.

3. ■ — after the *done* signal is accepted, the *FSM* is set to state $State_1$ and the *User* may send out the *'ack.*

4. ♠ — a token request arrives. The *FSM* is woken up and will be either in state $State_0$ (have to fetch the token) or in state $State_1$ (already has the token).

5. $\Diamond$ — the node has the token and the token may be passed out. The *Token* is woken up and the *FSM* is set to state $State_0$.

Here is this specification in formal CCS:

```
User   = req.'gt0.cs.'grant.done.pt0.'ack.User
Token  = lreq.'gt1.pt1.'tout.Token
State0 = gt0.'rreq.tin.'cs.'pt0.State1 + gt1.'rreq.tin.'pt1.State0
State1 =           gt0.'cs.'pt0.State1 + gt1.'pt1.State0
```

when node is initialised to not have token ($NODE_0 spec$),

```
NODE0spec  = ( User | Token | State0 ) \ { gt0,gt1,cs,pt0,pt1 }
```

when node is initialised to have token ($NODE_1 spec$),

```
NODE1spec  = ( User | Token | State1 ) \ { gt0,gt1,cs,pt0,pt1 }
```

```
Command: sort NODE0spec
**{done,lreq,req,tin,'ack,'grant,'rreq,'tout}

Command: sort NODE1spec
**{done,lreq,req,tin,'ack,'grant,'rreq,'tout}

Command: min NODE0spec
Save result in identifier: NODE0spec'
**NODE0spec' has 36 states.
```

```
Command: min NODE1spec
Save result in identifier: NODE1spec'
**NODE1spec' has 36 states.

Command: vs 4 NODE0spec'
=== lreq req 'rreq tin ===>
=== lreq 'rreq req tin ===>
=== lreq 'rreq tin req ===>
=== lreq 'rreq tin 'tout ===>
=== req lreq 'rreq tin ===>
=== req 'rreq lreq tin ===>
=== req 'rreq tin lreq ===>
=== req 'rreq tin 'grant ===>
```

Step 2: Test of Specification

1. Deadlock Free

   ```
   Command: cp NODE0spec'
   Proposition: BOX (~Deadlock)
   **true
   ```

2. Livelock Free

   ```
   Command: cp NODE0spec
   ```

```
Proposition: ~Livelock
**true
```

3. Safety

   - Mutual Exclusion

     The mutually exclusive process in the above specification is that the token is requested and captured by the *User* and the token is moved on anti-clockwise upon receiving a request from its left adjacent neighbour.

     ```
     Command: cp NODE0spec'
     Proposition: [tin](Mutual_Exclusion 'grant 'tout)
     **true
     ```

     We further show that once the *User* enters its critical section after capturing the token, it is only possible for the mutually exclusive counterpart to move the token left after the critical section has been finished.

     ```
     Command: cp NODE0spec'
     Proposition: [tin] BOX (['grant] NEC_FOR 'ack 'tout)
     **true
     ```

     If the left neighbour wins the token, the *User* has to capture the token again before being allowed to enter its critical section.

     ```
     Command: cp NODE0spec'
     Proposition: [tin] BOX (['tout] NEC_FOR tin 'grant)
     **true
     ```

- Absence of Unsolicited Response

  All acknowledgements are in response to corresponding input requests. For example, the acknowledgement from the *User* (*'ack*) signalling the end of its critical section is in response to the input request *req* for entering the critical section.

```
Command: cp NODE0spec'
Proposition: Absence_of_Unsolicited_Response req 'ack
**true
```

  The token's left moving *'tout* is in response to the request from the current node's left adjacent neighbour *lreq*.

```
Command: cp NODE0spec'
Proposition: Absence_of_Unsolicited_Response lreq 'tout
**end
```

4. Liveness

- Guaranteed Events

  Because of the mutually exclusive divergence, none of the input and output actions are guaranteed events.

```
Command: cp NODE0spec'
Proposition:
(Guaranteed_Event   req) | (Guaranteed_Event   lreq) |  \
(Guaranteed_Event   tin) | (Guaranteed_Event  'rreq) |  \
```

```
(Guaranteed_Event 'tout) | (Guaranteed_Event   'ack) |  \
(Guaranteed_Event  done) | (Guaranteed_Event 'grant)
**false
```

- Ensured Response

  We cannot show that *'grant* and *'tout* are ensured responses to some trigger events in this specification because these two actions are the trigger events signalling which critical section the node is in. But *'grant* and *'tout* are ensured responses once their corresponding critical sections are chosen. This can be shown when watching-signals are inserted at the beginning of each critical section.

  By using *'grant* as the pre-request, we show that once *'grant* happens, *done* and *'ack* are ensured to happen.

```
Command:  cp NODE0spec'
Proposition: (Ensured_Response 'grant done) &  \
             (Ensured_Response 'grant 'ack)
**true
```

  Similarly, we can show that *tin* is the ensured response of *'rreq*. This means that once a token is requested by the node from its left adjacent neighbour, the token will eventually be passed to the node.

```
Command: cp NODE0spec'
Proposition: Ensured_Response 'rreq tin
**true
```

Step 3: Implementation

The implementation of Martin's arbiter is given by Ebergen [EBG92], and shown in Figure 5.11. It is achieved by combining a two-way RGD Arbiter and a finite state machine.

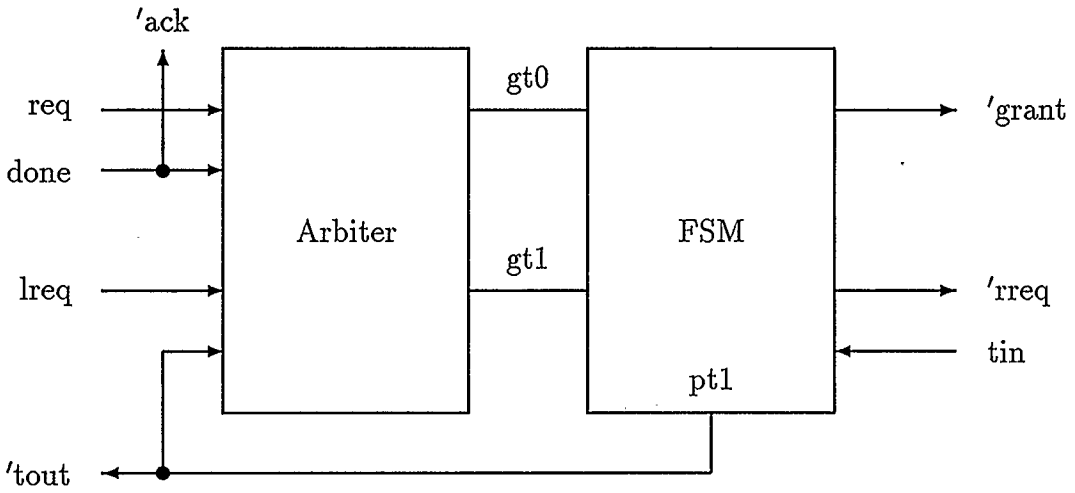


Figure 5.11: Implementation of Martin's Arbiter

The RGD arbiter is specified in Chapter 4 as one of Brunvand's Control Path Modules.

The finite state machine used here deals with requests from the RGD arbiter.

```
FSM0 = gt0.'rreq.tin.'grant.FSM1 + gt1.'rreq.tin.'pt1.FSM0
FSM1 = gt0.'grant.FSM1           + gt1.'pt1.FSM0
```

1. FSM_0 stands for the state when token is elsewhere:

   - A signal on gt_0 corresponds to a user request. The token is fetched and then the request is granted. FSM_0 moves to FSM_1 signifying that the

token is here.

- A signal on gt_1 corresponds to a request for the token from the left. The token is fetched and then passed on. The output signal on pt_1 is forked to *'tout* and also back to the RGD arbiter so that the arbiter is cleared. FSM_0 remains at state FSM_1.

2. FSM_1 stands for the state when the token is local:

   - A signal on gt_0 corresponds to a user request. The request is granted at once. FSM_1 remains at state FSM_1.
   - A signal on gt_1 corresponds to a token request from the left. The token is passed on at once. Similarly, the output signal on pt_1 is forked to *'tout* and also back to the RGD arbiter. FSM_1 then moves back to FSM_0.

Hence we have,

```
NODE0imp =
( Fork    [ done/a,    r/b,  ack/c                          ]  \
| Arbiter [  req/r1, gt0/g1,   r/d1, lreq/r2,gt1/g2, d/d2 ]  \
| FSM                                                        \
| Fork    [ pt1/a,  tout/b,    d/c                          ]  \
) \ {gt0, gt1, pt1, d2, r, d}
where
  FSM  = FSM0
```

The node with the token has $FSM = FSM_1$.

5.4 Brunvand's CSA Adder

In his Ph.D thesis [Bru91c], Brunvand described a *Carry-completion Sensing Addition* (CSA[1]) module suitable for building self-timed adders. The trick in this design is that the carry of the addition results is propagated using two separate signals: *'cout* signifying a carry, and *'dout* signifying the absence of a carry. Only one of the two signal lines may be active during one computing period.

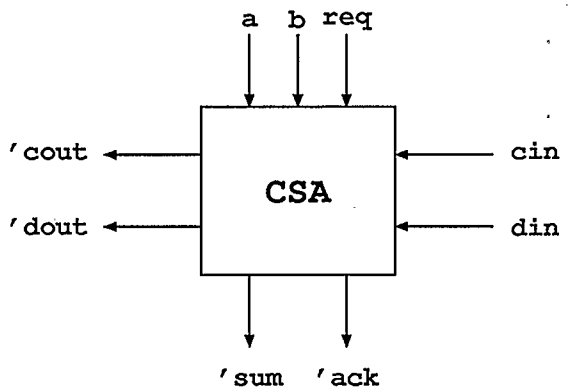


Figure 5.12: Carry-completion Sensing Addition (CSA) Module

Self-timed adder circuits can be implemented in term of the CSA modules. It is interesting to note that the specification of a $n-$bit adder circuit is exactly the same as that of the basic CSA module, though the complexity of implementation increases a lot. We here develop the specification of the CSA module, which is also suitable for the $n-$bit self-timed adder constructed with n CSA modules.

[1]It is CCS in Brunvand's thesis, we name it CSA here to distinguish it from the CCS process algebra.

Step 1: Design of Specification

The specification of the CSA module is quite tricky compared with specifications we have had before. In this specification, we have three agents, namely north N, east E, and west W, acting in parallel under suitable timing constraints.

The main idea here is that the CSA module operates in two phases: (i) compute the appropriate sum and carry, and then (ii) reset all the internal carry lines. And it uses four-phase signaling: a rising *req* transition initiates the addition and a rising *'ack* indicates the completion of computation; a falling *req* transition is used to initialize the module for the next addition by resetting the carry signal (or the no carry signal) back to low again and a falling *'ack* is used to indicate that the module is ready to accept new data for computation.

Informally, we give the specification as follows:

1. the CSA module is enabled via *req* when both *a* and *b* inputs are set
2. either a *cin* or a *din* is accepted for computing
3. the sum value is computed and carry out available *'cout* or carry out unavailable *'dout* rippled out
4. when the computing has been finished by a *'ack*, the computing results *'sum* is ready for read
5. the module is then enabled again by another *req*
6. carry in signal (*cin* or *din*) and carry out signal (*'cout* or *'dout*) are reset to their initial states (logic 0) in any order

7. the final completion is signalled by another *'ack*

And the timing precedences are tabulated below:

a, b	**phase 1**							**'sum**	**phase 2**				
		1		2		3				4		5	
a			cin		'cout						cin + din		
	req	≺	+	≺	+	≺	'ack	'sum	req	≺		≺	'ack
b			din		'dout						'cout + 'dout		

Formally in CCS, we have,

```
CSA = ( N | E | W ) \ { s1,s2,s3,s4,s5 }
where
  N = data.req.'s1.s3.'ack.'sum.req.'s4.'s4.s5.s5.'ack.N
  E = s1.(cin.'s2.s4.cin.'s5.E     + din.'s2.s4.din.'s5.E)
  W = s2.('cout.'s3.s4.'cout.'s5.W + 'dout.'s3.s4.'dout.'s5.W)
```

In the above specification, we use signal *data* to express both bundled data a and b for simplicity (instead of $a.b + b.a$). The issue here is that a *req* will only be produced after all the data is valid on the data path, and we don't care about in which sequence a and b are set.

```
Command: sort CSA
**{cin,data,din,req,'ack,'cout,'dout,'sum}
```

```
Command: min CSA
Save result in identifier: CSA'
**CSA' has 26 states.

Command: vs 10 CSA'
=== data req cin 'cout 'ack 'sum req cin 'cout 'ack ===>
=== data req cin 'cout 'ack 'sum req 'cout cin 'ack ===>
=== data req cin 'dout 'ack 'sum req cin 'dout 'ack ===>
=== data req cin 'dout 'ack 'sum req 'dout cin 'ack ===>
=== data req din 'cout 'ack 'sum req din 'cout 'ack ===>
=== data req din 'cout 'ack 'sum req 'cout din 'ack ===>
=== data req din 'dout 'ack 'sum req din 'dout 'ack ===>
=== data req din 'dout 'ack 'sum req 'dout din 'ack ===>
```

Step 2: Test of Specification

1. Deadlock Free

```
Command: cp CSA
Proposition: BOX (~Deadlock)
**true
```

2. Livelock Free

```
Command: cp CSA
Proposition: ~Livelock
**true
```

3. Safety

- Absence of Unsolicited Response

  It is obvious that the addition result *'sum* is in response to input data *data*; and the acknowledgement for completing the addition *'ack* is in response to the input request of starting the computation *req*. (The pair of *req* and *'ack* can also be viewed as the completion of resetting internal carry lines in response to the request of resetting.)

```
Command: cp CSA'
Proposition: Absence_of_Unsolicited_Response data 'sum
**true
```

```
Command: cp CSA'
Proposition: Absence_of_Unsolicited_Response req 'ack
**true
```

- Mutual Exclusion

  According to the design strategy, the two carry in lines which stand for carry in available (*cin*) or carry in unavailable (*din*) respectively must be mutually exclusive. This must also be true for the two carry out lines (*'cout* and *'dout*).

```
Command: cp CSA'
Proposition: Mutual_Exclusion cin din
**true
```

```
Command: cp CSA'
Proposition: Mutual_Exclusion 'cout 'dout
**true
```

Further, we show that one of the two carry in signals is necessary for completing the addition (producing a 'sum) although neither *cin* nor *din* should necessarily be available.

```
Command: cp CSA'
Proposition: NEC_FOR cin 'sum
**false

Command: cp CSA'
Proposition: NEC_FOR din 'sum
**false

Command: bsi Carry_in
Enter action list: cin din

Command: cp CSA'
Proposition: NEC_FOR' Carry_in 'sum
**true
```

4. Liveness

   - Guaranteed Events

In this design, all the input and output actions are guaranteed events.

```
Command: cp CSA'
Proposition:
(Guaranteed_Event  data) & (Guaranteed_Event  req) &  \
(Guaranteed_Event  'ack) & (Guaranteed_Event 'sum) &  \
(Guaranteed_Event   cin) & (Guaranteed_Event  din) &  \
(Guaranteed_Event 'cout) & (Guaranteed_Event 'dout)
**true
```

Step 3: Implementation

A 4−bit self-timed adder can be implemented using four CSAs. The CSAs are chained together with the *'cout* and *'dout* of one stage connected to *cin* and *din* of the next. The *req* signal for starting computation is forked to the four CSAs, while the *'ack* signals from each CSA are joined together with C-elements. The *data* signal here stands for the bundled data a (a_0, a_1, a_2 and a_3 in each CSA), and the bundled data b (b_0, b_1, b_2 and b_3 in each CSA).

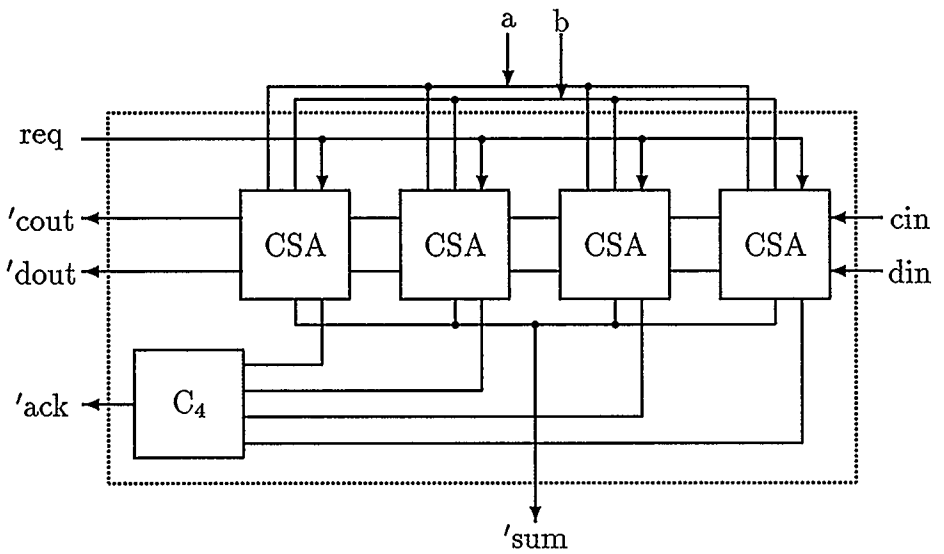


Figure 5.13: A 4-Bit Self-timed Adder Based upon the CSA Module

In CCS, the above implementation can be specified as:

```
ADD4 = ( Fork [ data/a, data0/b,   d00/c ]  \
       | Fork [  d00/a, data1/b,   d11/c ]  \
       | Fork [  d11/a, data2/b, data3/c ]  \
       | Fork [  req/a,    r0/b,   r00/c ]  \
       | Fork [  r00/a,    r1/b,   r11/c ]  \
       | Fork [  r11/a,    r2/b,    r3/c ]  \
       | CSA4                               \
       | C    [   a0/a,    a1/b,   a01/z ]  \
       | C    [   a2/a,    a3/b,   a23/z ]  \
       | C    [  a01/a,   a23/b,   ack/z ]  \
       | C    [ sum0/a,  sum1/b, sum01/z ]  \
       | C    [ sum2/a,  sum3/b, sum23/z ]  \
```

```
        | C     [sum01/a, sum23/b,    sum/z ]  \
        )\{data0,data1,data2,data3,d00,d11,r0,r1,r2,r3,r00,r11, \
           a0,a1,a2,a3,a01,a23,sum0,sum1,sum2,sum3,sum01,sum23}
where
  CSA4 = ( CSA [ data0/data,   r0/req,  cin/cin,  din/din,   \
                    c1/cout,   d1/dout,  a0/ack, sum0/sum ]  \
         | CSA [ data1/data,   r1/req,   c1/cin,   d1/din,   \
                    c2/cout,   d2/dout,  a1/ack, sum1/sum ]  \
         | CSA [ data2/data,   r2/req,   c2/cin,   d2/din,   \
                    c3/cout,   d3/dout,  a2/ack, sum2/sum ]  \
         | CSA [ data3/data,   r3/req,   c3/cin,   d3/din,   \
                  cout/cout, dout/dout,  a3/ack, sum3/sum ]  \
         ) \ { c1,c2,c3,d1,d2,d3 }
```

5.5 Sutherland's Move Machine

The Move Machine was first suggested by Sutherland, who observed that conventional processing units spend much of their time moving data back and forth between the memory and the CPU. The instruction set of the Move Machine merely controls the flow of instructions and data. It has no instructions for arithmetic or logic operations.

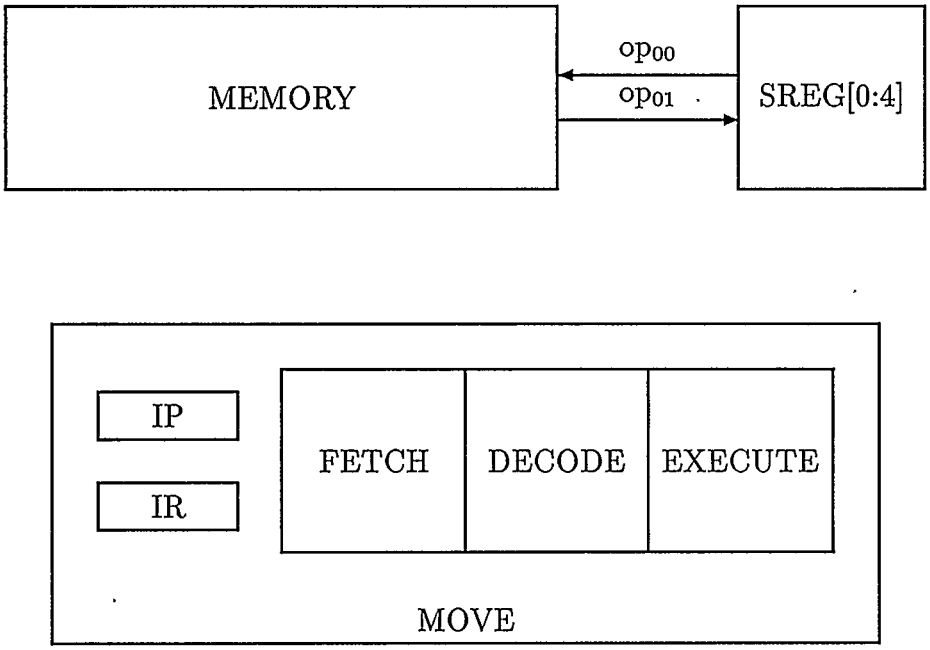


Figure 5.14: The Structure of Sutherland's Move Machine

Although most of the processors have at least 100 to 200 instructions in complexity and the Move Machine has only a few instructions, it follows the usual design principles as larger processors. Hence the Move Machine is a handy-sized example for learning to reason about processors in general.

A VHDL description of the Move Machine is given by Roy [RKDV92] with minor modifications from the ISPS (Instruction Set Processor Specifications) description proposed by Drongowski [Dro89]. The more abstract CCS specification follows their lead and decomposes the top level specification into three major processes:

1. FETCH

   This is the start of the Move Machine. It fetches an instruction (*sIR*) from the instruction register IR according to the current instruction pointer IP.

2. DECODE

   After the instruction is fetched, it is decoded according to the 2–bit address mode identification. The four possible modes are absolute decode mode (*case00*), immediate decode mode (*case01*), indirect decode mode (*case10*) and IP relative decode (*case11*). Here we do not model these cases in any detail.

3. EXECUTE

   Before starting execution, the instruction pointer is updated first. This is achieved by a *modifyIP* action which modifies the IP according to the mode of decode. The instruction is then executed according to the 2–bit operation code. The four possible operations are load register (*op00*), store register *op01*, jump (*op10*) and halt (*op11*). Again, we do not model these operations in detail.

Step 1: Design of Specification

Formally in CCS we have,

```
MOVE = ( FETCH | DECODE | EXECUTE ) \ {sDEC,sEXEC,sFETCH}
where
  FETCH   = sIR.'sDEC.'sFETCH.FETCH
  DECODE  = sDEC.(case00.D' + case01.D' + case10.D' + case11.D')
  EXECUTE = sEXEC.modifyIP.((op00.E'+op01.E'+op10.E') + op11.nil)
  D'      = 'sEXEC.DECODE
  E'      = sFETCH.EXECUTE
```

N.B., agent D′ and E′ are used for conciseness and clarity in specification.

```
Command: sort MOVE
**{case00,case01,case10,case11,modifyIP,op00,op01,op10,op11,sIR}

Command: min MOVE
Save result in identifier: MOVE'
**MOVE' has 5 states.

Command: pi MOVE'
**MOVE' = MOVE'0
where MOVE'0 = sIR.MOVE'2
  and MOVE'2 = case00.MOVE'4 + case01.MOVE'4 +
               case10.MOVE'4 + case11.MOVE'4
  and MOVE'4 = modifyIP.MOVE'5
  and MOVE'5 = op00.MOVE'0 + op01.MOVE'0 + op10.MOVE'0 + op11.MOVE'7
  and MOVE'7 = nil
end
```

Step 2: Test of Specification

1. Not deadlock free

   Due to the *halt* operation in the execution process (op11), it is possible for the Move Machine to get deadlock. But this is the normal termination of the Move Machine.

   ```
   Command: cp MOVE'
   Proposition: BOX (~Deadlock)
   ```

```
**false
```

We further show that the normal termination of the Move Machine is after doing an *op11*:

```
Command: cp MOVE'
Proposition: BOX ([op11] Deadlock)
**true
```

But we cannot deadlock after any none *op11* move.

```
Command: cp MOVE'
Proposition: BOX ([-op11] ~Deadlock)
**true
```

2. Livelock free

```
Command: cp MOVE
Proposition: ~Livelock
**true
```

3. Safety

   - Mutual Exclusion

     We have defined the mutually exclusive macro for two actions, we now extend this macro to make it suitable for more than two actions. This is achieved by using the action list P whose length can be changed according to the particular example.

```
Command: bmi Mutual_Exclusion' a P
Body: (BOX ([a][P]F)) & (BOX ([P][a]F))
```

With the extended macro, we can show that after one of the four branches in DECODE (or EXECUTE) is processed, none of the other branches in parallel can be processed, they are mutually exclusive.

```
Command: bsi Rest_case
Enter action list: case01 case10 case11

Command: cp MOVE'
Proposition: Mutual_Exclusion' case00 Rest_case
**true

Command: bsi Rest_op
Enter action list: op01 op10 op11

Command: cp MOVE'
Proposition: Mutual_Exclusion' op00 Rest_op
**true
```

4. Liveness

- Guaranteed Events

  Because the Move Machine will stop operating immediately after an *op11* action, none of the actions in this design is a guaranteed event.

```
Command: cp MOVE'
Proposition:
(Guaranteed_Event    sIR) | (Guaranteed_Event modifyIP) |  \
(Guaranteed_Event case00) | (Guaranteed_Event   case01) |  \
(Guaranteed_Event case10) | (Guaranteed_Event   case11) |  \
(Guaranteed_Event   op00) | (Guaranteed_Event     op01) |  \
(Guaranteed_Event   op10) | (Guaranteed_Event     op11)
**false
```

But this does not mean that we have lost the liveness property in this system. We can still show that all the actions will eventually happen provided an *op11* does not occur (we have already shown that Move Machine stops operating upon receiving an *op11*).

```
Command:  cp MOVE'
Proposition: ( BOX (EV ([-op11]<sIR>T)))      &  \
             ( BOX (EV ([-op11]<modifyIP>T))) &  \
             ( BOX (EV ([-op11]<case00>T)))   &  \
             ( BOX (EV ([-op11]<case01>T)))   &  \
             ( BOX (EV ([-op11]<case10>T)))   &  \
             ( BOX (EV ([-op11]<case11>T)))   &  \
             ( BOX (EV ([-op11]<op00>T)))     &  \
             ( BOX (EV ([-op11]<op01>T)))     &  \
             ( BOX (EV ([-op11]<op10>T)))
**true
```

5.6 Summary

In this chapter, we have specified a variety of asynchronous hardware architectures using parallel specifications. Using this style, designing specifications becomes quite methodical. After finding a suitable decomposition of the interface into parallel agents, we write down their interactions separately, and then weave them together by considering timing constraints. Since the set of possible behaviours of a specification is hard to fathom, we proposed various property macros for testing the consequences of hardware specifications in chapter 3. These together with others found useful in practice were applied systematically to the examples in this chapter. This made it possible for us to test our specifications thoroughly for the desired properties. This work on the methodology of asynchronous design is the major contribution of the research described in this thesis.

Implementations (by others) corresponding to our specifications were given in all but one case (the Move Machine). The problem of proving the equivalence between specification and implementation is discussed in the final chapter.

Chapter 6

Conclusions

6.1 Summary

The contributions of this thesis have been to:

- develop a parallel specification style which results in neat and compact specifications for complex asynchronous hardware, and which scales well when the number of inputs to a system increases;
- propose a set of property macros based upon the modal μ-calculus to test the consequences of specifications, such as deadlock, livelock, safety and liveness;
- and to apply the parallel specification style and the macro-based testing style to a modest range of asynchronous designs.

In Chapter 2, we detailed the syntax and semantics of CCS and explained various notions of process equivalence.

In Chapter 3, we covered the Hennessy-Milner Logic (HML) and the modal μ-calculus supported by the CWB. A set of basic property macros were proposed and motivated.

In Chapter 4, we specified a library of control path modules and data path modules for self-timed design using parallel specifications wherever possible. We tested the behaviour of these specifications using the basic property macros proposed in Chapter 3.

In Chapter 5, we specified a variety of asynchronous hardware architectures using parallel specifications. We also tested these specifications for desired properties such as deadlock-free, livelock-free, safety and liveness.

Because of its succinctness, scalability and equational reasoning capability, the CCS/CWB has proved to be a good tool for specifying and testing asynchronous designs. The parallel specification style makes it possible to avoid developing specifications state by state, which is tricky, tedious and error prone. The macro-based testing style makes it possible to investigate the consequences of a design specification thoroughly before embarking upon an implementation — after all, it is rather pointless implementing something that can deadlock, livelock, is unsafe, isn't live, etc. Such properties are hard to locate via simulation, and it is usual practice never even to look for such possible defects. In this regard, applying process logics to hardware descriptions is an important improvement in the design methodology.

The specification and testing experiences gained through this thesis work also show up certain deficiencies in CCS/CWB. (i) it takes a long time to minimize a specification (usually several hours for 1000 states). (ii) the CWB is not efficient in detecting the location of deadlock. It usually takes several iterations to achieve a satisfactory specification and each iteration has to be checked for deadlock. (iii) since CCS does not support the simultaneous synchronisation of several actions, we cannot formalise the isochronous fork assumption in CCS. This turns out to be essential in implementing basic level modules [Mar90c, BE90]. The CWB also supports SCCS [Mil83b] which does permit the simultaneous operation of several actions. We believe it is the right tool for modelling this level of implementation, but an implementation defect (now rectified) stopped us from carrying out the work this time.

6.2 Future Work

6.2.1 Equivalence between Specification and Implementation

In addition to the specifying and testing of an asynchronous design, another important aspect of formal verification is concerned with checking whether an implementation conforms to its specification. The importance of this equivalence checking is that once proved, we know that the implementation will hold all the properties possessed by the specification, and we can then replace the notationally cumbersome implementation by a compact specification when reasoning further up the hierarchicy.

Unfortunately, equivalence checking between specification and implementation is not an easy task in the CWB. The action sequences of input transitions in the specification are inherently well-handled, but there is no easy way to constrain the input action sequences to an implementation. Hence, if there are some constraints on the operating sequences of input actions, they can only be enforced by the environment in which the implementation operates.

For example, we pointed out the operating environment for the control circuit for a 4–stage micropipeline in Chapter 5:

1. Input Constraint: After an input request *rin*, an acknowledge *ain* must occur before another *rin*.

2. Output Constraint: An output request *'rout* must occur before an acknowledge *aout* can be received.

Expressed in CCS we have,

```
ENVin  = reqin.'rin.ain.'ackin.ENVin
ENVout = rout.'reqout.ackout.'aout.ENVout
```

With this operating environment we have

```
CC4specENV = ( CC4spec | ENVin | ENVout )\{rin,ain,aout,rout}
CC4impENV  = ( CC4imp  | ENVin | ENVout )\{rin,aout,ain,rout}
```

Although we can prove that *CC4impENV* is equivalent to *CC4specENV* on the CWB

```
Command: eq
Agent: CC4specENV
Agent: CC4impENV
**true
```

we cannot be completely confident in that each component of the implementation sits in an environment (provided by the rest of the design) which corresponds exactly to that holding when we proved it equivalent to its specification, and if it isn't, we cannot replace it by its specification. In the above example, it means that each of the four C-elements used to implement the control circuit should be operating in an environment which does not change its delay insensitivity; and this should also hold when the 4–stage control circuit is used as basic module to construct control circuit with more stages.

It is necessary and important to prove an implementation faithfully conforms to its specification, but the testing we require is not directly supported by the CWB now. The automation in the CWB of such delay insensitivity guarantees at different hierarchy levels would be an interesting and worthy component in a PhD. Larsen's PhD thesis [Lar86] provides a suitable starting point for this research.

6.2.2 Silicon Compilation

The CCS process algebra has a succinct and compact notation and very clear and clean semantics. Since it has only three basic operators (., | and +) all a prototype silicon compiler has to be able to do is translate these operators into hardware.

1. "." sequences actions and is implemented by just a wire
2. "+" is used to express the nondeterministic choices amongst independent agents and can be implemented by the sequencer;
3. "|" has to cope with various synchronisations between parallel agents. these include:
    - the operation of external transitions
    - the internal 1–1 handshake is a simple C-element with its output forking back to both clients as an acknowledgement
    - the many-to-1 and 1-to-many handshakes which can again be handled by the sequencer
    - the many-to-many handshake which still requires careful specification and a cheap implementation

Such a prototype silicon compiler would be inefficient, and there is much work to do in locating special cases (e.g. if we know that inputs are mutually exclusive, we can use the cheaper join instead of the expensive sequencer). Again this is a suitable topic for a PhD, and Brunvand [Bru91c] is a good source for possible optimisations.

Finally, we believe another suitable PhD topic would be mechanisation of the design methodology proposed in this thesis.

Putting all these ideas together leads towards tool support for the automatic synthesis of asynchronous circuits from specifications expressed in CCS.

Bibliography

[BA91] G. Bruns and S. Anderson. The Formalization and Analysis of a Communications Protocol. Technical Report LFCS, Edinburgh University, 1991.

[BE90] J. Brzozowski and J. Ebergen. On the Delay-Sensitivity of Gate Networks. Technical Report 90–5, University of Eindhoven, 1990.

[BGS+90] G. Birtwistle, B. Graham, T. Simpson, K. Slind, M. Williams, and S. Williams. Verifying an SECD Chip in HOL. In L. J. M. Claesen, editor, *Formal VLSI Corrrectness Verification. VLSI Design Methods II, Proceedings of the IFIP TC10/WG10.5 Workshojp held in Leuven, November 13–16, 1989*, pages 369–378, Amsterdam, 1990. North Holland.

[Bre90] G. Brebner. A CCS-based Investigation of Deadlock in a Multi-process Electronic Mail System. Technical Report, University of Edinburgh, 1990.

[Bru87] E. Brunvand. Parts-R-Us: A chip aparts... Technical Report CMU-CS-87-119, Carnegie Mellon University, 1987.

[Bru91a] G. Bruns. A Language for value-passing CCS. Technical Report LFCS ECS-LFCS-91-175, Edinburgh University, 1991.

[Bru91b] E. Brunvand. A Cell Set for Self-timed Design Using Actel FPGA's. Technical Report UUCS-91-013, University of Utah, 1991.

[Bru91c] E. Brunvand. Translating Concurrent Communicating Programs into Asynchronous Circuits. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.

[BS90] J. Bradfield and C. Stirling. Verifying Temporal Properties of Processes. In *Proceedings of CONCUR '90*, number 458 in LNCS, pages 115–125. Springer-Verlag, 1990.

[CDS93] Bill Coates, Al Davis, and Ken Stevens. Automatic Synthesis of Fast Compact Self-Timed Control Circuits. Submitted for publication to the IFIP Manchester Asynchronous Design Workshop, 1993.

[Com92] R. Comerford. How DEC developed Alpha. *IEEE Spectrum*, pages 26–31, July, 1992.

[CPB90] R. Cleaveland, J. Parrow, and B.Steffen. The Concurrency Workbench. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 24–37. Springer Verlag, 1990.

[Dam90] M. Dam. Translating CTL into the Modal μ-calculus. Technical Report ECS-LFCS-90-123, Department of Computer Science, University of Edinburgh, Edinburgh, 1990.

[Dil89] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, Cambridge, Massachusetts, 1989.

[Dro89] P. J. Drongowski. An Organization-Level Story Board for Agent – A VLSI Designer's Assistant. Internal Report, DSRG, CES Department, Case Western Reserve University, Cleveland, Ohio, 1989.

[Ebe88] J. Ebergen. A Formal Approach to Designing Delay-Insensitive Circuits. Computing science note 88/10, Eindhoven University of Technology, 1988.

[EBG92] J. C. Ebergen, P. F. Bertrand, and S. Gingras. Distributed Mutual Exclusion: Specification and Implementation of Delay-Insensitive Protocols. Tech report, Department of Computer Science, University of Waterloo, 1992.

[EG91] J. C. Ebergen and S. Gingras. An Asynchronous Stack with a Constant Response Time. Tech report, Department of Computer Science, University of Waterloo, 1991.

[Gor88] M. J. C. Gordon. HOL: A Proof Generating System for Higher Oorder Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128, Norwell, Massachusetts, 1988. Kluwer.

[HM80] M. Hennessy and R. Milner. On Observing Nondeterminism and Concurrency. In *Lect. Notes in Computer Science 85.* Springer, 1980.

[HM85] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *J. Assoc. Comput. Mach.*, 32:137–161, 1985.

[Hoa85] C. A. R. Hoare. *Communicationg Sequential Processes.* Prentice Hall International, London, 1985.

[Joy88] J. Joyce. Formal Verification and Implementation of a Microprocessor. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157, Norwell, Massachusetts, 1988. Kluwer.

[Koz83] D. Kozen. Results on the Propositional mu-calculus. *Theoretical Computer*, 27:333–354, 1983.

[Lar86] K. G. Larsen. Context-Dependent Bisimulation between Processes. Technical Report ECS-LFCS-86-4, University of Edinburgh, 1986.

[LM86] K. G. Larsen and R. Milner. A Complete Protocol Verification using Relativized Bisimulation. Technical report ecs-lfcs-86-13, University of Edinburgh, 1986.

[Mar85] A. Martin. Distributed Mutual Exclusion on a Ring of Processes. *Science of Computer Programming*, 5:265–276, 1985.

[Mar90a] A. J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, New York, 1990. Addison-Wesley.

[Mar90b] A. J. Martin. Synthesis of Asynchronous VLSI Circuits. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, North Holland, 1990.

[Mar90c] A. J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In *Proc. 6th MIT Conference on Advanced Research in VLSI.* MIT Press, 1990.

[MFR85] C. E. Molnar, T. P. Fang, and F. U. Rosenberger. Synthesis of Delay-insensitive Modules. In H. Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI.* Computer Science Press, 1985.

[Mil83a] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil83b] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.

[Mil91] R. Milner. The Polyadic π-Calculus: A Tutorial. Technical Report ECS-LFCS-91-180, Computer Science Department, University of Edinburgh, 1991.

[Mol91] F. G. Moller. The Edinburgh Concurrency Workbench, Version 6.0. Tech Report, Computer Science Department, University of Edinburgh, 1991.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: specification.* Springer-Verlag, New York, 1992.

[MPW89a] R. Milner, J. Parrow, and D. Walker. A calculus of Mobile Processes: Part I. Technical Report ECS-LFCS-89-85, Computer Science Department, University of Edinburgh, 1989.

[MPW89b] R. Milner, J. Parrow, and D. Walker. A calculus of Mobile Processes: Part II. Technical Report ECS-LFCS-89-86, Computer Science Department, University of Edinburgh, 1989.

[MT89] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. Technical Report ECS-LFCS-89-104, Department of Computer Science, University of Edinburgh, Edinburgh, 1989.

[Par85a] J. Parrow. Fairness Properties in Process Algebra. PhD thesis, Uppsala University, Uppsala, Sweden, 1985.

[Par85b] J. Parrow. Verifying a CSMA/CD-protocol with CCS. Technical report, University of Edinburgh, 1985.

[RKDV92] J. Roy, N. Kumar, R. Dutta, and R. Vemuri. DSS: A Distributed High-Level Synthesis System. *IEEE Design and Test of Computers*, 9(2):18–32, 1992.

[Sei80] C. L. Seitz. System Timing. In C. A. Mead and L. A. Conway, editors, *An Introduction to VLSI Systems*, pages 218–262, Reading, Massachusetts, 1980. Addison Wesley.

[Sut89] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[SW91] C. Stirling and D. Walker. Local model checking in the modal μ-calculus. *Theoretical Computer Science*, 89:161–177, 1991.

[Tof90a] C. Tofts. A Synchronous Calculus of Relative Frequency. In J. W. Klop J. C. M. Baeten, editor, *CONCUR '90*, number 458 in LNCS. Springer-Verlag, 1990.

[Tof90b] C. Tofts. The Autosynchronisation of Leptothorax Acervorum (Fabricius) Described in WSCCS. Technical Report ECS-LFCS-90-128, Department of Computer Science, University of Edinburgh, Edinburgh, 1990.