A Formal Approach to Hardware Analysis

Niklas Traub

Doctor of Philosophy
University of Edinburgh
1986

Abstract

Electronic circuits are entering many aspects of our daily lives. Considerations of safety and cost have prompted a good deal of interest in verifying that they function as expected. This thesis explores several ways of conducting such verifications using a formal, mathematical calculus. The calculus is based on sequences of events and has a simple semantics.

One of the most important influences on hardware designs is time. A variety of temporal concepts are therefore represented in the calculus, including multilevel clocks, clock skew and others. Moving between different granularities of time is easily accomplished, thus providing support for temporally as well as spatially structured designs.

Consequences of design decisions may not be immediately obvious, so it is important to have a way of investigating their ramifications. Accordingly, techniques are developed for experimenting on designs, as are several mechanisms for analyzing the results. Some exploit basic properties of the calculus to increase the efficiency of a simulation at no extra cost.

More formal methods of verification are also provided that permit a large difference in complexity between a specification and its implementation. To ensure that this difference does not unwittingly introduce design errors, a notion of constraints on the use of an implementation is introduced. This finds application in building libraries of pre-verified parts, as well as during the refinement of hierarchical designs.

Finally, the techniques are illustrated through several examples, including a hardware implementation of a deck of playing cards and a simple CRT controller.

Acknowledgements

I am indebted to my supervisor Dr. George Milne for guiding me through the pitfalls of producing a thesis. He first introduced me to the field of concurrency and provided an endless source of ideas. Without his support and encouragement this work would never have happened.

Many others contributed suggestions and discussions too. First and foremost, I would like to thank Kevin Mitchell for saving me from straying down some infeasible paths, for providing valuable hints and for quarrying. I would also like to thank K.V.S. Prasad for help with CCS, Sassan Mohseni for reading and commenting on a draft of this thesis and all those others whose discussions influenced my ideas. Moral support was provided in abundance by T. Horton and J. Wilson.

This research was made possible through grants from International Computers Ltd., Lattice Logic Ltd. and the Science and Engineering Research Council.

Declaration

I declare that this thesis has been composed by myself and the work is my own, except where indicated in the text.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Electronic circuits are finding their way into many of the objects that surround us in our daily lives. Some of these objects are responsible for our health and safety, so it is important that they function properly and without unexpected side-effects. The dramatic reduction in cost of Integrated Circuit building blocks made possible by large production volumes has played an important part in this proliferation. Not only are errors in the design of these blocks potentially dangerous, but they are also costly when duplicated in quantity. For these reasons, a great deal of interest has been shown in methods for validating designs of electronic circuits.

At one time, it was possible for the engineer to determine if a design would work correctly by building a prototype. The prototype would have been constructed from much the same parts as the end product, giving confidence that experimenting on one would yield results applicable to the other. The experiments were simplified considerably by the fact that basic building blocks were quite small and the connections between them could be examined using oscilloscopes, logic analyzers and other test equipment.

With the appearance of sophisticated microprocessors and other monolithic building blocks, this approach to design began to break down. Larger and larger portions of products began to migrate to silicon in the form of dedicated microchips. A silicon chip is a single entity whose functions can only be accessed through its interface pins; internal nodes are extremely difficult to examine because they are small and covered with oxides. In addition, the behavior of prototypes built from, say, discrete TTL parts might alter dramatically when reimplemented in silicon. Delays and hence timing information may change due to the different physical properties of the new medium.

Cost is another important factor. Prototypes built from small blocks are relatively easy to redesign should alterations become necessary, whereas the expense of fabricating a limited quantity of experimental chips can be considerable. The problem is exacerbated by the tendency of engineers to iterate their designs, changing features as problems or shortcomings are revealed.

These deficiencies of the prototyping/breadboard approach have prompted a large increase in popularity of software simulations of designs. Software models are easily mutable and, as computers grow more powerful, are being made larger and more complete. Models have been formulated that span all levels of abstraction from the interactions of large functional blocks to the detailed behavior of primitive devices in the fabrication medium. Experimenting on designs represented by such models is much simpler than on prototypes because all internal nodes are accessible and sophisticated analysis tools can be applied to the results.

Experiments, whether on a bread-boarded circuit or on a software simulation, have their own attendant problems. The aim of the experiments is usually to show that the design behaves according to some predetermined idea of how it should behave. Verifying this for a large design can require a very large number of test stimuli. In fact, the only way to be completely confident in the absence of errors is to apply all possible combinations of input values. Fueled by this rapid growth in complexity, attention has been turned to formal, mathematical, methods for conducting the verification in the hopes that they might alleviate the problem. This approach relies on mathematical laws to prove properties for large classes of input values. Analogous methods have been used for many years in other disciplines, such as civil engineering and aerospace, where mathematical models of physical processes are manipulated to produce safer and higher performance products.

Producing validated designs using a software representation requires support in three areas. First and foremost, a mechanism for capturing the physical behavior of a design is needed:

(1) A Hardware Description Language

Hundreds of hardware description languages have been created to serve widely varying needs. Some only describe the structure of a design, while others also incorporate ideas of how elements behave. The former are used as input to automatic design systems and will not concern us here. The latter type can be used for simulation and/or verification, which is what we are interested in.

Since the goal of using a language to describe hardware is to show that a design will work properly when it is built, preferably without exhaustively simulating it, we need support for:

(2) A Method for Verification

Existing programming languages, which have influenced many hardware description languages, are notorious for their convoluted semantics. Verification involves showing that a specification of the intended behavior of a design is satisfied by an implementation. The task of demonstrating this is simplified considerably if the representation medium has a simple semantics. Even with simple semantics, expressing the ideas behind a design in a particular language can be difficult. Often the only way to convince oneself that an attempt to do this is reasonable is to experiment on it and see if the ideas are adequately represented. Thus the last area that should be supported is:

(3) A Method for Experimentation

Some researchers [Rem 81] believe that the experimental (or simulation) approach to design should be abandoned altogether in favor of formal verification alone. We shall not be so limiting since it seems that experimentation has a legitimate place in the design cycle. After all, testing various possibilities is just a simple form of the proof technique known as case analysis.

This thesis seeks to address each of the three areas. A mathematical calculus is chosen as the Description Language, because it is both powerful and has a simple semantics. As will be shown, it can be used to represent designs at many levels of abstraction and possesses facilities for moving between the levels. The formal basis allows proofs of equivalence to be conducted and several ways of doing this are presented. Also demonstrated are ways of conducting experiments on descriptions without leaving the framework. Such a cohesive approach means that several avenues of attack can be applied to a particular problem, thus increasing the chance of a successful resolution. Indeed, the size of problem that can be worked for a given amount of effort increases as analytic techniques are developed through the thesis.

1.1 Related Work

Simulation as a way of experimenting on a design has been around a long time. Every type of simulator, from the detailed device level—typified by SPICE [Valdimirescu 81]—through the high level functional simulator, has put on an appearance at one time or another. Functional simulators, in particular, take many forms since they are often simply a model of the design coded in a high level programming language (*e.g.,* PASCAL) especially for the project.

The majority of simulators operate at a single level of abstraction because they can then be optimized for execution speed. Many levels of abstraction are passed through during the refinement of a design, however, and maintaining consistency between the inputs to different simulators is notoriously difficult. Prompted by this, steps have been taken toward integrating several simulation levels, particularly at the primitive device level [Chen 84, Antognetti *et al.* 81, De Man *et al.* 81]. The simulator DIANA [De Man *et al.* 81] typifies this approach by allowing parts to be specified in varying degrees of detail between the gate level (in terms of discreet valued signals) and the circuit level (in terms of continuous voltages and currents). Because of the optimizations needed to make these simulators practical, it is usually impossible to simulate a design at any but the provided levels of detail.

More general and more extensible simulators have put on an appearance in recent years. Called *Behavioral* [Coelho 84] or *Functional* simulators, they provide most of the flexibility of a high level programming language (as well as the complicated semantics in most cases [Nichols 83]). A typical example is the

ELLA simulation system [Morison 85]. ELLA has a functional input language that lets the user define new data types as well as operations on them. Libraries of predefined functions provide software analogues of the parts that will be used to fabricate a design. ELLA differs from most related simulators, however, in that some encouraging work has been done on combining it with a formal verification system in order to check a commercial microprocessor [Cullyer 85]. High level blocks were described with the LCF–LSM notation (discussed in more detail below) with manual proofs of equivalence conducted between their descriptions and those of their refinements. At a certain point, the design was translated into ELLA and simulated. This was done because the designers felt that LCF–LSM was not descriptive enough for low level circuits.

Complete verification of hardware without recourse to simulation has attracted a lot of interest in the last few years. Hardware systems are characterized by a high degree of parallelism in their operation—indeed, explicit steps must taken to introduce sequentiality. For this reason, many of the formalisms that were developed to reason about concurrent systems have been applied to hardware. The most popular of these are covered below.

The work of Milne and Milner [MM 79] on Concurrent Processes has inspired a number of calculi for reasoning about concurrency. Milne went on to develop the Dot Calculus [Milne 80] and then CIRCAL, which forms the basis for the work in this thesis. Milner produced the Calculus of Communicating Systems (CCS) [Milner 80] and then a more general synchronous version (SCCS). The calculi all describe behavior in terms of sequences of atomic events and algebraic laws are given that define how parallel sequences interact. The ideas behind these calculi have influenced several interesting approaches to hardware verification.

Milner gives an elegant proof of correctness for a latch constructed from NOR gates in SCCS [Milner 83]. Also using SCCS, Backhouse [Backhouse 83] verifies a regular language recognizer built from primitive recognizer boxes. He proves that the specification of what the system should do is equivalent to the aggregate behaviors of the primitive components by means of a technique called *bisimulation* [Milner 83]. Briefly, this involves showing that every state in either of the behaviors can be matched with one in the other. Hennessy [Hennessy 84] introduces a transformational method of proving equivalences in the same calculus that is a bit easier to work with. The expression denoting a specification is transformed into that of an implementation (or vice versa) by syntactic manipulations based on a collection of axioms. The axioms are powerful enough to permit a good deal of difference in complexity between the two expressions, thus supporting simpler specifications. By way of example, several systolic array designs are shown correct with the method. Finally, Subrahmanyam [Subrahmanyam 83] uses CCS as a basis for an input language to a form of "silicon compiler" that generates layout from high level descriptions.

One of the most comprehensive applications of formal techniques to all aspects of hardware analysis has been that of Cardelli [Cardelli 82]. Using a small set of combinators (similar to those of SCCS) he introduces an algebra for manipulating networks of hardware components. The algebra is applied to several levels of description with mappings given between them. Examples include clocked systems, transistor networks and topological connections of layout primitives. Also presented is an implementation of a general purpose programming language for manipulating layout information. The language supports algebraic operations on pictures corresponding to mask level data. The semantics of hardware are captured using an interesting calculus of Real Time Agents. This calculus is powerful enough to support reasoning about low level device behavior through (continuous) time. The emphasis throughout the thesis is on algebraic techniques for all levels of VLSI design activity, from the abstract architectural to the primitive fabrication data.

Perhaps the most successful of these types of formalisms has been the work Gordon on LSM (Logic for Sequential Machines) [Gordon 81]. He develops a simple model of hardware behavior in terms of functions from signals to signals, with a notion of state added to capture sequential behavior. The methodology is formulated as a calculus with combinators similar to those of CCS and given a denotational definition. Simulations of LSM descriptions could easily be run due to the denotational basis, but Gordon adopts a more ambitious approach by embedding the logic in an existing theorem proving environment (LCF) to produce a true verification system (LCF–LSM) [Gordon 83a]. Using this system, he has proven a microcode implementation of a small computer correct [Gordon 83b]. Other researchers have also used the same formalism with some success. Barrow [Barrow 83, Barrow 84] implements it as a PROLOG program with which he has automatically verified a module containing three multipliers and two adders down to the transistor level. The LSM formalism has limitations when applied to designs having multiple levels of clocks and those where explicit timing information is needed. Because of this, Gordon has since moved on to another formalism, which will be discussed below.

Another promising approach to hardware verification utilizes temporal logic to describe the behavior of a system through time. Moskowski defines a form of temporal logic called *Interval Temporal Logic* that captures many of the concepts needed to model hardware [Moszkowski 83]. Time is not measured explicitly, but is instead broken up into implicit intervals corresponding to how long an operation takes. This concept seems particularly suited to the higher level descriptions of digital systems where relative time (*e.g.*, with reference to system clocks) is more important than absolute time. Moskowski's thesis shows how a wide variety of digital phenomena can easily be modelled in ITL and relies on equational manipulation to prove equivalences. Moskowski has since produced an imperative programming language called *Tempura* [Moszkowski 84] from a subset of ITL which serves as a simulator for designs coded in the formalism. Examples that have been simulated with it include a small array of RAM cells and a stream driven parser.

Influenced by Moskowski and the VERITAS project of Hanna [Hanna 84], Gordon has replaced his Logic for Sequential Machines with a version of Higher Order Logic [Gordon 85]. HOL seems to meet many of the requirements of hardware verification, because both behavior (functional and temporal) and structure can be represented in a clear manner. Like LSM, it has been embedded in the LCF theorem proving environment, which has allowed several real VLSI designs to be verified, including portions of the Cambridge Ring Chip [Gordon 84].

Rem [Rem 84] advocates a slightly different approach to modelling concurrent systems—one using *traces*. Traces are sequences of atomic actions similar to those that form the basis of CCS-like languages. Instead of defining special construction operators, however, Rem generates traces by means of the language of regular expressions. Because of this, he notes that designs described by traces can be realized as compositions of finite state machines, thus giving a structural flavor to the formalism. Conditions for delay-insensitive communications are given, which he argues can be used to determine which portions of a circuit should be grouped together.

Closely related to the idea of traces is that of *streams*. Sheeran [Sheeran 83] introduces a variant of the functional programming language FP, called μFP, that operates on streams of objects rather than the single objects of the original language. The language is built around the notion of functional composition which she extends to include a notion of state. The compositional nature yields succinct descriptions with a natural interpretation as networks of functional units. This interpretation is exploited in constructing a simple floor planner that reflects the semantics of a design in the layout that it produces. Semantics preserving transforms are used to produce different layout arrangements. μFP deals only with simple synchronous systems and cannot handle the more general timing requirements needed by many digital designs.

Another researcher who has applied streams to hardware descriptions is Kloos [Kloos 86]. He provides numerous agents for operating on streams as well as operators for composing the agents sequentially and in parallel. His approach is more general than that of Sheeran, with applications to register transfer, gate level and switch level systems demonstrated.

The work of Kelly [Kelly 84] on the CRITTER project is an interesting attempt at automatically critiquing digital designs. An applicative language that deals with a wide variety of temporal effects (including statistical and worst/best case estimates of delay times) as well as functional information is used as input to an expert system driven equation simplifier. Operating conditions for a design are derived from the estimated delays of its components and illegal or marginal results are reported together with suggestions of their cause. The system can also be used as a multi-level and symbolic simulator wherein the function of a block can be derived in terms of its input variables. Although quite powerful, it seems to have difficulties with feedback and the treatment of state information is complicated.

Johnson [Johnson 84] shows how synchronous hardware can be synthesized from simple recursion equations. The idea is to produce implementations that are correct by construction, rather than verifying them after they have been designed.

Finally, Chen [Chen 83] applies fixed-point semantics to a model of processes that includes both spatial and temporal information. With it as a basis, she examines some systolic algorithms and a means of representing networks of MOS transistors at the switch level. She also discusses an implementation of her formalism as a multilevel simulator which is embedded in a high level programming language.

Many of the approaches discussed above have been successful because they have concentrated on limited types of digital designs (*e.g.*, purely synchronous systems). In particular, little attention has been paid to nondeterministic phenomena and asynchronous systems. The former appear when random external events, such as interrupts in a microprocessor, can influence the behavior of a design. The latter hold promise for designs that need to operate at a high speed. Asynchronous (or self timed) blocks have the property that they can pass along an output value as soon as it is calculated and then begin work on the next input. Synchronous systems, in contrast, require that each block wait for a global clock event before processing the new input values. All blocks therefore function at the speed of the slowest. This thesis seeks to address some of these issues.

1.2 Topics Covered in the Thesis

Chapter 2 covers the first component that was considered necessary for a verification system: a description language. It introduces the calculus that will serve as a basis for the material developed in succeeding chapters. The calculus is designed for describing the interactions through time of objects that function in parallel. Hardware is by nature highly parallel—explicit steps must usually be taken to impose sequentiality—so it seems appropriate to apply the calculus to hardware problems.

The calculus portrays behavior in terms of (partial) orderings of events in time. These orderings are expressed as equations constructed using the operators of the calculus. There are only a small number of core operators, so the complete framework can be described quite compactly when compared to most hardware description or programming language definitions. Building on the core operators, some derived operators are presented that make expressions syntactically simpler and easier to understand. Certain constructs arise quite commonly when describing hardware, so these are assigned their own notation for convenience.

Once a feel for the nature of the calculus has been developed, consideration is turned to deciding just how events should relate to the physical phenomena that we are seeking to model. Two different approaches are presented and their merits and flaws discussed, thus ending the second chapter.

One of the greatest concerns of hardware designers is time. Time manifests itself in circuits as delays, which are notorious for causing glitches and erratic malfunctions. The basic calculus, as described in Chapter 2, has no notion of quantitative time built in, so the third chapter presents several different ways of obtaining this type of information—without changing the underlying semantics in any way. Operators are developed that allow some ambiguity in describing relationships between events, loosening the rather precise view of temporal ordering imposed by the basic calculus. Delays were mentioned as a source of problems for designers, so the chapter goes on to consider several different types. Each approximates different physical properties, hence the variety.

Being able to describe objects mathematically is not very useful if no analysis can be done on them. Chapter 4 shows how simple properties of the calculus can be exploited to generate simulations of a design's behavior through time. The method is refined further to obtain an increase in efficiency without complex optimizations. Finally, some ideas for analyzing the results of simulations are presented. This concludes the discussion of the third area identified as requiring support in a verification system: a method of experimenting on designs.

Simulation alone is not adequate for building confidence in the correctness of a design. It can show that the portions effected by test stimuli function incorrectly, but it cannot show conclusively that they function properly unless all possible input sequences are applied. This is where verification in the form of mathematical proof enters the picture. A specification is written that captures the desired behavior of the design and the constructed implementation is shown to be in some sense equivalent. One way to do this is to manipulate the two sets of equations using the laws of the calculus until they agree. This is not always possible, however, since specifications will be designed for a particular use, whereas implementations are often quite general. A microprocessor, for example, can be used to implement the logical NOT function as can an inverter, but the constraints on their use are quite different. A notion of context dependent equivalence is therefore developed and presented in two alternative forms. Both are useful for determining if two systems can ever be considered equivalent and if so, what constraints must be placed on their use. Together, these techniques support the second area that must be included in a verification system: a method of verifying designs.

Chapter 6 consolidates much of the material discussed in the thesis by working through several examples. The first reveals some of the unexpected problems one encounters when combining functional units and shows how they may be circumvented. The second is a design for a deck of playing cards that is implemented in terms of gate level components in Appendix A. It serves as a good vehicle for demonstrating ways of coping with nondeterministic behavior utilizing the techniques of Chapter 5. The final example shows how the description of a cathode ray tube controller can be simplified by partitioning it in both time and space. The controller's behavior is explored through simulation and a modification is considered that improves its performance.

Finally, the thesis concludes with a summary of the results obtained. Areas needing further work are identified and some possible links with other work considered. A software system for manipulating expressions is briefly described as well as various ideas for its enhancement. The system proved highly useful in checking several of the examples that were worked in Chapter 6.

Chapter 2

The CIRCAL Calculus

This chapter begins by presenting a brief and informal introduction to the calculus CIRCAL (CIRcuit CALculus), which will be used as the basis for the ideas developed in this thesis. Technical issues are covered cursorily in Section 2.6 and the interested reader is encouraged to look up the references mentioned there for more information.

The calculus is derived from the work done by Milne and Milner [MM 79] and from the material in [Milne 77]. It is quite similar in approach to CCS [Milner 80], SCCS [Milner 83], and other frameworks such as CSP [Hoare 78]. It is designed for describing the behavior of systems composed of concurrently executing agents. Behavior is described in terms of communications between agents and between agents and the environment. Communications will also be referred to as *signals* and *events* depending on what they represent. Similarly, agents will be called *processes* to emphasize their concurrent nature.

The chapter begins by presenting the basic operators of the calculus. It then goes on to examine some derived operators and their properties. Occasionally, their derivation will seem unmotivated until they are applied in a later chapter. Finally, some structuring syntax and representation style is discussed and a few examples are worked.

2.1 The Structure of Processes

Processes communicate with each other and the environment through labelled *ports* (also called *channels*). The labels of these ports form the *sort* of the agent. A sort can be thought of as the set of all possible communications that an agent may ever participate in, no matter what state it evolves into. Sorts are static and may contain labels that will never actually be used in a communication. This is similar to having pins on an integrated circuit package that are not connected to the internal devices.

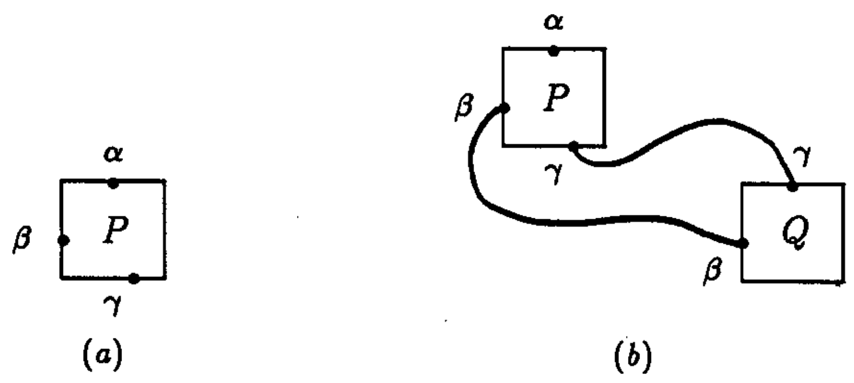


Figure 2–1: (a) A process with sort $\{\alpha, \beta, \gamma\}$.
(b) Two statically connected processes

Processes will often be pictured as boxes with ports on the periphery (Figure 2–1(a)). No ordering is implied by the placement of the ports nor by the placement of boxes with respect to one another. Boxes communicate with each other by connecting similarly labelled ports with an arc (Figure 2–1(b)). The connections are *static* because they indicate potential interactions between the processes—some of which may never happen.

Labels are drawn from the set $\mathcal{L}$ of all possible labels, and will typically be ranged over by lower case Roman letters written in a `typewriter` font for clarity in complex expressions. Lower case Greek letters will usually refer to sets of labels.

Labels may be indexed to produce vectors of channels. Each of these is a label in its own right, but we will adopt the convention that a channel label will also refer to all its indexed derivatives. A sort is therefore a (not necessarily finite) subset L of $\mathcal{L}$ and includes all the indices of its labels.

The behavior of a process is described by one or more CIRCAL expressions. Expressions are built up out of other expressions, operators, and atomic state names. These names correspond to particular states in the evolving behavior of a process. They are drawn from the set of allowable process names $\mathcal{P}$. To distinguish them from channel labels, they will be written as uppercase Greek and Roman characters, or as capitalized identifiers. Like labels, they may be indexed to generate families of process names. Names are assigned to expressions using the binary operator $\Leftarrow$.

2.2 The Core Operators

Part of the strength of this and similar calculi is the small number of core operators. The smaller the number of operators, the easier it is to design a complete set such that all fit cleanly and unambiguously together. The following sections describe the primitive combinators of the calculus as well as a few derived operators that will prove useful.

2.2.1 Guarding

A process may be described in terms of the communications that it engages in before evolving into a new (*resultant*) state. In other words, no transition occurs until the required communications have been resolved. The state transition is said to be *guarded* by the communications. Guards are sets of labels drawn from the sort L of the process and placed before the resultant state enclosed in curly brackets. They introduce sequentiality into a series of actions by forcing some actions to occur after or at the same time as others. A similar function is provided by the ';' semicolon operator in imperative programming languages and in temporal logic [Moszkowski 83]. Note that we now have an implicit sense of time. An action can be required to occur after another action, but nothing can be said about the length of the interval between them. It could be one micro-second or ten years.

CIRCAL allows *sets* of labels for guards. The communications named in the set are considered to occur at exactly the same time, no matter how closely they are examined. The sets obey all the normal set-theoretic operations, including union ($\cup$), intersection ($\cap$), difference ($\setminus$), etc. When the set contains more than one element it will be referred to as a *composite label.* The empty set will not be allowed.

Guards may also be superscripted to indicate a sequence of identical events. In addition, their labels may be indexed, allowing one label to refer to a logically grouped set of labels. The labels are included in a sort by specifying the domains of the indices, e.g. $\mathtt{a}_i$ s.t. $i \in \mathcal{D}$. The domains may be infinite, but should be denumerable.

Here are some examples that informally describe the semantics of guarding:

Examples:

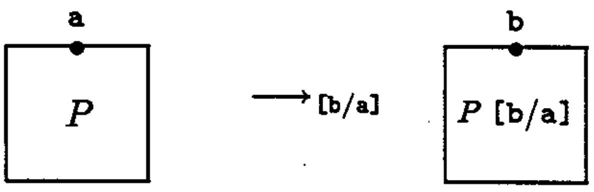
$Q \Leftarrow \{\mathtt{a}\}\{\mathtt{b}\}^3 P$	Means Q accepts an a communication, followed by three b communications, and then evolves into state P.
$R \Leftarrow \{\mathtt{a}\ \mathtt{b}\}\{\mathtt{d}\ \mathtt{e}\}\, R1$	R must interact with both an a and a b communication simultaneously, followed by an e and a d simultaneously before evolving into $R1$.
$S_i \Leftarrow \{\mathtt{b}_i\}\, S_{i+1}$	S_i accepts any communication on b that is indexed by a value equal to its current state variable i, and evolves to a state indexed by the incremented value.

The last example actually defines a family of processes, but since they are closely related by the index variable i can be thought of as *states* of some "meta-process", with i the *state variable* of S.

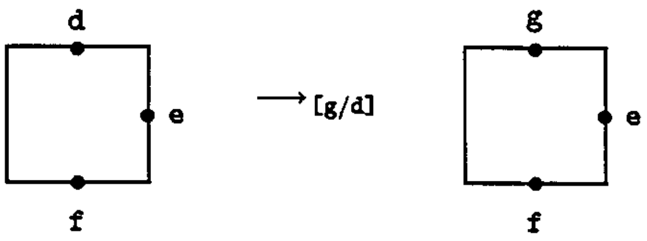
2.2.2 Relabeling

Copies of an expression may be generated by changing its labels and index names using the *relabeling* or *morphism* operator. Relabellings are denoted by the postfix operator $[\rho]$, where ρ is a morphism from labels to labels. Typical morphisms will be of the form b/a, c_i/c meaning that any channel labelled by a will be changed to b and similarly c channels will be changed to c_i channels. The sort of the expression is implicitly changed as well. We shall make heavy use of this operator to instantiate copies of generic components.

Examples:



$$(\{a\}\,P)\ [b/a] \quad = \quad \{b\}\ (P\ [b/a])$$



$$(\{d\ e\}\ \mathit{Q1} + \{f\}\ \mathit{Q2})\ [g/d] \quad = \quad \{g\ e\}\ (\mathit{Q1}\ [g/d]) + \{f\}\ (\mathit{Q2}\ [g/d])$$

Relabelling will also be used to bind free variables in expressions. For example, if a process P is defined as:

$$P(x) \ \Leftarrow\ \{a_x\}\,\{x_x\}\,P(x+1)$$

then

$$P(1) \ = \ P\ [1/\!\!/x] \ = \ \{a_1\}\,\{x_1\}\,(P(1+1)\ [1/\!\!/x])$$

All free occurrences of x are bound to 1. The fact that a binding rather than a relabelling takes place is denoted by the $/\!\!/$ replacement. The effects of the binding hold until a resultant that re-binds the variable. Thus the free x's in $P(1+1) = P(2)$ will get bound to 2 rather than 1.

One important restriction will be made on ρ throughout this thesis, namely that the relabellings be static. In other words, the morphism must not contain any free variables that can change value as the process evolves. If this were allowed, processes could be created dynamically by recursively calling the morphism with the free variables being bound to different values each time (*cf.* [Milner 83, page 292ff.]). Since we are concerned mainly with hardware models, which obviously cannot spawn new components, this restriction is not too limiting. Removing it would require that the results of Chapter 5 be re-investigated.

2.2.3 Deterministic Choice

A process can typically evolve into one of many possible states depending on what communications it has with other processes. By explicitly listing these choices, we can model the possible branchings in its behavior. This is captured by the binary operator '+', called the *deterministic choice* operator. By deterministic, we mean that the choice will somehow be made by external factors in the environment (*e.g.*, the influence of other processes). The arguments of + will be referred to as *branches* for reasons that will become clear in Section 2.5. Both will have the same sort since they are just different possible futures of the same process.

Examples:

$Q \Leftarrow \{\mathtt{a}\}\, Q1 + \{\mathtt{b}\}\, Q2$	means that Q will evolve into $Q1$ if an a event occurs, or into $Q2$ if a b event occurs.
$P \Leftarrow \{\mathtt{c}\}\, P1 + \{\mathtt{c}\ \ \mathtt{d}\}\, P2$	means that P will become $P1$ under a c stimulus, or if c and d occur simultaneously, will become $P2$.

Occasionally the symbol + will be used to indicate the addition operator. The context should make this use obvious.

2.2.4 Nondeterministic Choice

Nondeterministic choice, denoted by the $\oplus$ operator, is similar to deterministic choice, but with a subtle difference. Here the choice is no longer under the control of the environment, but instead is made internally. The system may arbitrarily pick one of the branches and make a silent move to that state. This happens when we are observing the system at a more abstract level of detail than it was originally modeled at. The internal states may make choices based on information that is now lost, and in doing so end up in a state that cannot be predicted from the inputs. Like the deterministic choice operator, both branches have the same sort.

Examples:

$Q \Leftarrow Q1 \oplus Q2$	means Q can spontaneously become either $Q1$ or $Q2$.
$P \Leftarrow \{\mathtt{a}\}\, P1 \oplus \{\mathtt{b}\}\, P2$	P can spontaneously become either $\{\mathtt{a}\}\, P1$ or $\{\mathtt{b}\}\, P2$.

2.2.5 Summation Operators

Often a state may have many possible states that it can evolve to. It is therefore convenient to have a shorthand for the expressing these multiple choices. This is done by defining the deterministic and nondeterministic *summation* operators $\sum$ and $\bigoplus$ as follows:

$$\sum_{i=1}^{n} P_i =_{def} P_1 + \cdots + P_n \tag{2.1}$$

$$\bigoplus_{i=1}^{n} Q_i =_{def} Q_1 \oplus \cdots \oplus Q_n \tag{2.2}$$

Here we depart slightly from the approach in [Milne 83a] by allowing the subscript i of the deterministic choice sum to range over any domain $\mathcal{D}$, not just natural numbers and not necessarily finite. This relatively minor syntactic change will make it possible to introduce *value passing* in communications.

2.2.6 Special Processes

It is possible to define a process that has no communications with the environment; in other words, it simply exists. This can happen when an agent performs a sequence of actions and terminates, or, as we shall see in the next section, when two processes deadlock. It is therefore useful to explicitly indicate such a behavior with the symbol "Δ". Δ has a sort just like any other process and will be subscripted by it to prevent ambiguity, *e.g.*, $\Delta_{\{a,\, b\}}$.

Another special process is the one that may perform any sequence of actions whatsoever. Any attempt at communicating with such a process may or may not succeed. It represents a process that we have absolutely no knowledge about, other than its sort. It will be called the *most nondeterministic* process and will be written as "Ω_L", after the notation used in [deNicola 82]. The CSP operator called CHAOS [Brookes 83] is also very similar. This operator will not be utilized until Chapter 5.

2.2.7 Composition

Up till now we have only discussed constructs for modeling the sequential behavior of a process. Systems, however, are built from many communicating processes and so a combinator is needed to generate the aggregate behavior from the sub-behaviors. This combinator is the binary *Dot Operator* "$\bullet$".

Most of the properties of the operator will not be discussed here. Instead the reader is referred to the table at the end of the section. The composition of deterministic sums, however, is important enough to warrant a closer examination. Law [$\bullet$ +] (also called the Expansion Rule) is defined recursively as follows:

Definition 2.2.1 The Dot Operator.

Given:

$$\begin{aligned} P &\Leftarrow \sum \lambda_i P_i \qquad \text{of sort } L \\ Q &\Leftarrow \sum \mu_j Q_j \qquad \text{of sort } M \end{aligned}$$

Where λ_i and μ_j may be single or multiple labels and i and j are unspecified indices

$$\begin{aligned} P \bullet Q \quad =_{def} \quad & \sum_{\lambda_i \cap M = \emptyset} \lambda_i \, (P_i \bullet Q) && (1) \\ + & \sum_{\mu_j \cap L = \emptyset} \mu_j \, (P \bullet Q_j) && (2) \qquad [\bullet +] \\ + & \sum_{\lambda_i \cap M \,=\, \mu_j \cap L} (\lambda_i \cup \mu_j) \, (P_i \bullet Q_j) && (3) \end{aligned}$$

The sort of $P \bullet Q$ is $L \cup M$. □

This rather formidable collection of summations simply enumerates the outcomes that are possible when two processes are run in parallel. Term (1) specifies all those communications of P that can never synchronize with Q—the *independent* labels of P. If no label in λ_i is in M then P may evolve into P_i independently of what Q does. Similarly, term (2) contributes the independent labels of Q. Term (3) actually embodies two possible outcomes. If both λ_i and μ_j are independent labels, they may both occur at exactly the same time and so both expressions will evolve. Alternatively, if λ_i and μ_j have some labels in common—*dependent* labels—then these labels represent a synchronization on those lines and both expressions will again evolve. If any of the labels in λ_i or μ_j are in the sort of the other process, and there are no corresponding labels in the other guard, then that term can never synchronize and is removed. If all the λ_i and μ_j have dependent labels that don't synchronize, we are faced with the situation commonly called *deadlock*. In other words, one process is trying to communicate with another via a line that is unavailable for synchronization. This is represented by an instance of the deadlock operator Δ with sort $L \cup M$ as described in the previous section.

Note that this definition of composition contrasts with that of the CCS "|" [Milner 80] and the SCCS "×" [Milner 83] operators. The | operator has, on the surface, a similar definition to that of $\bullet$ for single labels (CCS does not support label sets for guards). Labels may either synchronize, resulting in a τ action, or may independently evolve. The key difference is the definition of independent. In CIRCAL, labels are independent if they do not belong to the sort of the other process, whilst in CCS, they are independent if the other process is not indulging in those communications. The connections in CIRCAL can be thought of as being "static" — if the other process is not listening, nothing can happen. In CCS, on the other hand, if the other process is not listening, the process will wait until it does — a "dynamic" concept of communication.

Another important difference is that communications in CIRCAL are multiway. Several processes can participate in the same communication instead of the CCS style of point to point communications. The intended meaning is that a particular event (the communication) must happen to all processes having it in their sorts for any of them to evolve. A parallel can be drawn with the join operation in Petri nets.

The SCCS $\times$ operator is similar to term (3) of the definition of $\bullet$. All guards in SCCS have an implicit synchronizing label that makes them dependent (synchronous). As in CCS, the communications are dynamic, so deadlock is not explicitly introduced.

Multiple compositions can be specified by applying the $\prod$ operator in the same manner as multiple choices are represented by $\sum$. Unlike the summation operator, however, products will always be finite. Infinite numbers of process have no physical analogy and are therefore to be avoided.

Composing processes that have state variables results in a process whose state is determined by the union of the variables of its components. Care must be taken that state variables have unique names, so that an instantiation will have predictable results.

Examples:

$$\begin{aligned}
A(x, y) &\Leftarrow \ldots \\
B(y, z) &\Leftarrow \ldots \\
C(x, y, y', z) &\Leftarrow A(x, y) \bullet B(y', z)
\end{aligned}$$

The state of the composite process C is determined by the union of the state variables of A and B. Notice how the state variable y contributed by B was named y' to prevent a conflict with the variable of the same name contributed by A.

2.2.8 Value Passing

All communications described so far have been in terms of atomic event names (the guard labels). The indexing of labels was mentioned in passing as a means of manipulating arrays of channels. The term "channel", however, seems to imply that something is being sent along it. How then can the ability to pass values between processes be defined?

Given the ability to index labels, it seems reasonable to allow the values of the indices to range through the domain of values that are to be passed through the channel. An event containing a label indexed by one of these values would then correspond to that value being passed. The ability to receive a binary value on channel b, for instance, might be modeled by a choice sum of two labels $\mathtt{b}_1$ and $\mathtt{b}_0$. A change from 0 to 1 would be indicated by a $\mathtt{b}_1$ event and similarly $\mathtt{b}_0$ would signal a 1 to 0 transition. This approach soon becomes tedious, especially when a signal can take on more than two values.

Fortunately, it is possible to use the infinite choice sum operator to capture this type of behavior. A process seeking to input a value on a channel b indexes b by a variable x and sums x over the domain D corresponding to the type of the channel. This would be written as:

$$P \Leftarrow \sum_{x \in D} \{\mathtt{b}_x\} \, P'$$

Any other process that generates a b event indexed by a particular value v will synchronize with the x that matches v, selecting that branch of the summation. The sorts of the two processes are extended to include the subscripted labels so that this synchronization takes place. This can be done by using a sort declaration operator that allows the types of the channels to be declared. For example:

$$P \textbf{ of sort } \{\mathtt{a},\ \mathtt{b},\ \mathtt{c}: \textbf{int},\ \mathtt{d}: \textbf{bool},\ \mathtt{e}\}$$

means that P is a process that has channels a, b, and c indexed by integers; a boolean valued channel d; and a non-typed channel e.

But what happens if the channel synchronizing with an infinite summation is also a summation? Which indices get bound? Using the present notation there is no way to tell which is intended to be the receiver and which the sender.

To indicate the direction of information flow, an arrow is written between the channel's label and its index. The arrows replace the summation over the domain, yielding a clearer and more compact expression. Arrows pointing toward the label denote output channels, whilst arrows pointing toward the index are input channels (the direction of the arrow symbolizes the direction of information flow). Values attached to output events will always bind variables attached to input events. The scope of the binding is the output labels of the guard and the expression being guarded. By convention, unary postfix occurrences of a directionality indicator bind a variable with the same name as the channel. Thus $\{\mathtt{b}\triangleright\}$ is a shorthand way of writing $\{\mathtt{b}\triangleright b\}$.

Examples:

$P(0) \Leftarrow \{\mathtt{a}\triangleright x\}\, P(x+1) + \{\mathtt{b}\triangleleft 1\}\, Q$ means that if $P(0)$ inputs a value 5 on a, it will evolve into $P(6)$. Alternatively, it can output the value 1 on channel b.

$$Square(i : \textbf{integer}) \Leftarrow \{\mathtt{out}\triangleleft(i \times i)\}\, Square(i+1)$$

$$Summer(x : \textbf{integer}) \Leftarrow \{\mathtt{in}\triangleright\}\, \{\mathtt{out}\triangleleft(x + in)\}\, Summer(x + in)$$

The *Square* process produces a series of out events indexed by the squares of the integers beginning from the initial value of i. The direction of the arrow indicates that the squares are being output and thus will bind any free variable attached to an out channel. Similarly in *Summer*, an event on the in channel will bind the variable *in*, which is added to x and output on out.

Care must taken in determining the binding rules for free channel variables to account for the presence of multiway synchronizations. If the processes P and Q are guarded by the same label c with a direction indicated and subscripted by x and y respectively, then there are four possible variations:

1. The c of P is an output label and the c of Q is an input label. This is a proper communication in which y gets bound to the value of x. The scope of the binding includes all resultants of Q until another input event that binds the same variable takes place (see Section 2.2.2).

   $$\{\mathtt{c}\triangleleft x\}\, P \bullet \{\mathtt{c}\triangleright y\}\, Q \quad = \quad \{\mathtt{c}\triangleleft x\}\, (P \bullet Q\ [x /\!/ y])$$

   Notice that the result of the communication is an output event. This happens because $\{\mathtt{c}\triangleleft x\}$ synchronizes with some subset of the summation symbolized by $\{\mathtt{c}\triangleright y\}$, thereby preventing the other possibilities from occurring.

2. The c of P is an input label and the c of Q is an output. This is the symmetric case to the above one.

3. Both are input labels, and consequently must input the same value.

$$\{\mathsf{c}\triangleright x\}\, P \bullet \{\mathsf{c}\triangleright y\}\, Q \quad = \quad \{\mathsf{a}\triangleright z\}\,(P\ [z/\!\!/x] \bullet Q\ [z/\!\!/y])$$

4. Both are output labels and therefore will only synchronize if $x = y$. If $x \in X$ and $y \in Y$ then the values that allow synchronization will belong to $X \cap Y$. Usually, x and y are either functions of the state variables, or variables that were bound by previous input communications.

$$\{\mathsf{a}\triangleleft x\}\, P \bullet \{\mathsf{a}\triangleleft y\}\, Q \quad = \quad \sum_{z\in(X\cap Y)} \{\mathsf{a}\triangleleft z\}\,(P\ [z/\!\!/x] \bullet Q\ [z/\!\!/y])$$

The scope of the binding action of input channels includes any output channels in the same label set, so:

$$\{\mathsf{a}\triangleleft 1\}\, P \bullet \{\mathsf{a}\triangleright x\ \ \mathsf{b}\triangleleft(x+3)\}\, Q \quad = \quad \{\mathsf{a}\triangleleft 1\ \ \mathsf{b}\triangleleft 4\}\,(P \bullet Q\ [1/\!\!/x])$$

The preceding rules seem obvious, but have some subtle pathological cases. Consider the following composition:

$$\{\mathsf{a}\triangleright x\quad \mathsf{b}\triangleleft f(x)\}\, P \bullet \{\mathsf{b}\triangleright y\quad \mathsf{a}\triangleleft g(y)\}\, Q$$

Both processes input a value and instantaneously output a function of it. Clearly, the two will synchronize only for values passed on a and b that are solutions to the simultaneous equations:

$$y = f(x) \qquad \text{and} \qquad x = g(y)$$

If at least one solution exits, the composite process will evolve to:

$$\{\mathsf{a}\triangleleft x'\ \ \mathsf{b}\triangleleft y'\}\,(P \bullet Q) \qquad x' \in \{\, x \mid x = g(f(x))\,\},\quad y' \in \{\, y \mid y = f(g(y))\,\}$$

If there is no solution, the composition deadlocks.

As can be seen from this example, implementing the synchronization semantics can be difficult in the general case. Even if instantaneous calculations are not allowed, there is still the problem of checking the synchronization of channels passing complicated data structures. Determining the result of combining a process that is outputting a list on channel b with another that is also outputting a list on the same channel involves comparing the lists. This is rather expensive for such a primitive operation as value passing. Should this be a problem, one could institute a syntactic requirement that channels outputting complex data be attached only to input channels. That way the equality check need never be made.

Another subtlety of value passing to be aware of is that one is allowed to *deterministically* output two different values on the same channel. For instance, the following expression is perfectly legal:

$$P \Leftarrow \{\mathrm{a}\triangleleft 42\}\, P' + \{\mathrm{a}\triangleleft 17\}\, P'' \tag{2.3}$$

It means something like, "You ask for a 42, I give you a 42. You ask for a 17, I give you a 17." One could argue that it is physically more natural to receive only one of these values, the choice being made nondeterministically. To do this, a new choice operator must be defined that has all the properties of $+$, except that law $[\gamma\ +]$ (see Section 2.2.11) applies only to channel names and ignores the attached values. Since expression like Equation 2.3 are rather contrived, such an operator will not be defined here. Should these types of expressions be required, the nondeterminism must be indicated explicitly.

2.2.9 Recursion and Induction

Some of the examples above contained recursive equations of the form:

$$P \Leftarrow \mathrm{E}(P) \tag{2.4}$$

Where E is an expression using P constructed from CIRCAL operators.

Recursion is the way infinite behaviors are represented and is encountered so frequently that it has a special notation:

$$\mathrm{rec}\ \tilde{X}.\, \mathrm{E}(\tilde{X})$$

$\tilde{X}$ is a vector of free variables that are bound every time the expression recurses. Equation 2.4 can be reformulated in terms of this operator as follows:

$$P = \mathrm{rec}\ X.\, \mathrm{E}(X)$$

Note how it is possible to write an equation of the form:

$$\text{rec } X.\, X$$

that has no useful meaning. We will therefore require that all recursive processes be ***guardedly well defined*** [Sanderson 82]. This means that at least one guard must be interposed between one level of a recursion and the next. The guarded well definedness requirement will apply only to definitions of a processes, since as we shall see in the next section, it is possible to produce this type of recursion by abstracting away the interposed guard.

Because the recursion operator can be used to produce sequences of events, a means of "folding" and "unfolding" these sequences is needed. The following law (called FOLD or UNFOLD as appropriate) shows how this is done:

$$\text{rec } X.\, E \quad = \quad E\,[(\text{rec } X.\, E) /\!\!/ X] \qquad [\text{rec}_1]$$

Remember that the // relabelling operation binds free variables. Thus the law says that a recursive equation can be replaced by a non-recursive one with the recursion variable bound to the original equation. The following equality, for example, holds under this law:

$$\text{rec } X.\, \{\text{a}\}\, X \quad = \quad \{\text{a}\}\, \big(\text{rec } X.\, \{\text{a}\}\, X\big)$$

Now that we have a means of writing infinite behaviors, we also need a means of comparing them. To this end, we follow Hennessy [Hennessy 84] and introduce a simple form of induction called *Fixpoint Induction.*

Definition 2.2.2 Fixpoint Induction

If we have a recursive process P defined as

$$P \Leftarrow \text{E}(P)$$

then if it can be shown that for another process Q

$$\text{E}(Q) = Q$$

we can conclude that

$$P = Q$$

□

The term E(Q) refers to the equation that defines P with all occurrences of the identifier name P replaced by the name Q. If this relabelled equation can be shown equivalent to the expression defining Q's behavior, then we can conclude that P and Q define the same process. A simple example (also due to Hennessy) serves to illustrate this concept. Consider P and Q defined by:

$$\begin{aligned} P &\Leftarrow \mathrm{E}(P) = \{\mathtt{a}\}\,P \\ Q &\Leftarrow \mathrm{F}(Q) = \{\mathtt{a}\}\,\{\mathtt{a}\}\,Q \end{aligned}$$

Intuitively, the two represent the same sequence of events, but how do we show that this is true in the calculus? Clearly, some use must be made of [rec$_1$] and Fixpoint Induction:

$$\begin{aligned} \mathrm{E}(P) &= \{\mathtt{a}\}\,P \\ &= \{\mathtt{a}\}\,\{\mathtt{a}\}\,P && [\text{UNFOLD}] \\ &= \mathrm{F}(P) \end{aligned}$$

Since the two definitions of the processes agree, we can use Fixpoint Induction to conclude that:

$$P = Q$$

The reference to law [rec$_1$] was done through the name [UNFOLD] to indicate the direction in which it was being used.

2.2.10 Abstraction

Because systems tend to be built by composing smaller functional units which in turn are compositions of still smaller blocks, it is important to have a way of looking at the complete behavior at different levels of detail. If an integrated circuit chip is observed at the pin level, for instance, the events associated with internal communications should be hidden from the observer. This is accomplished by applying the *abstraction* operator, written "−". Abstraction implies the removal of labels associated with internal lines, so that the events associated with them are no longer available to the environment. Like the Dot Operator, abstraction on terms having deterministic choice sums is important enough to consider in detail.

The analogue to Law [• +] for abstraction is called [− +] and is also defined as a summation:

$$\begin{aligned}
P &\Leftarrow \sum \lambda_i P_i \\
P - \mathtt{b} &\Leftarrow M \oplus \Big[M + \sum_{\substack{\mathtt{b} \in \lambda_i \\ \{\mathtt{b}\} \neq \lambda_i}} (\lambda_i \setminus \{\mathtt{b}\})\,(P_i - \mathtt{b}) \quad + \sum_{\mathtt{b} \notin \lambda_i} \lambda_i\,(P_i - \mathtt{b}) \Big] \\
M &\Leftarrow \bigoplus_{\lambda_i = \{\mathtt{b}\}} (P_i - \mathtt{b})
\end{aligned}$$

This rather complex expression may be explained briefly as follows:

The two deterministic summation terms are generated by terms in P that either contain b as a <u>member</u> of their guards or do not contain it at all. If b is part of a guard, it is removed since it can no longer be seen. Otherwise, the term evolves as usual.

The term represented by M is generated by terms in P that were guarded by label sets of the form {a} . Clearly, if a has been made internal, we will no longer be able to see if it occurs (a *silent move* is said to have taken place), hence the nondeterministic choice between M and the deterministic sums. The reason that M is also present in the deterministic sum is more subtle. Suppose that the environment never communicates with any of the guards belonging to the deterministic summations. Normally, the term would then evolve to Δ_L, but we know that an internal b communication is always possible. In the absence of any external communications, the internal one will eventually take place forcing the expression to evolve into M. Hence the presence of M in the deterministic choice as well.

Examples:

$$\begin{aligned}
(\{\mathtt{a\ b}\}\, P1 + \{\mathtt{b\ c}\}\, P2) - \mathtt{b} &= \{\mathtt{a}\}\,(P1 - \mathtt{b}) + \{\mathtt{c}\}\,(P2 - \mathtt{b}) \\
(\{\mathtt{a\ b}\}\, Q1 + \{\mathtt{b}\}\, Q2) - \mathtt{b} &= \{\mathtt{a}\}\,(Q1 - \mathtt{b}) \oplus (Q2 - \mathtt{b}) \\
(\{\mathtt{a\ b}\}\, R1 + \{\mathtt{b}\}\, R2 + \{\mathtt{c}\}\, R3) - \mathtt{b} &= \big[\{\mathtt{c}\}\,(R3 - \mathtt{b}) + \{a\}(R1-b) + (R2-b)\big] \oplus (R2-b) \\
(\{\mathtt{a}{\rhd}1\ \mathtt{b}\}\, P1 + \{\mathtt{a}{\rhd}2\ \mathtt{c}\}\, P2) - \mathtt{a} &= \{\mathtt{b}\}\,(P1 \overset{-a)}{+} \{\mathtt{c}\}\,(P2 - a)
\end{aligned}$$

Abstracting several channels is such a common operation that we will allow sets as arguments:

$$P - \{\mathtt{a,\ b,\ c}\} \quad =_{def} \quad P - \mathtt{a} - \mathtt{b} - \mathtt{c}$$

Recursion and abstraction can interact in unexpected ways. Consider the process defined by:

$$P \text{ of sort } \{\mathtt{a}, \mathtt{b}\} \Leftarrow \{\mathtt{a}\}\, P$$

Now hide the a channel to produce

$$\begin{aligned}(P - \mathtt{a}) \text{ of sort } \{\mathtt{b}\} &\Leftarrow (P - \mathtt{a}) \oplus (P - \mathtt{a}) \\ &= P - \mathtt{a}\end{aligned}$$

What is the meaning of this expression? Have we succeeded in reducing the expression to a meaningless equality? The answer is no, because the equation represents a process that is engaging in an infinite number of internal actions and is therefore unable to communicate with the outside world (it is said to be *diverging*). Two possible interpretations can be made of this behavior.

The first claims that since any attempt to communicate with the process on the b will never be answered, the result should be indistinguishable from $\Delta_{\{\mathtt{b}\}}$.

Alternatively, it can be claimed that we have no idea whether this will be an infinite divergence or that at some unknown time in the future a b communication will be offered. Instead of termination, we would have an instance of the most nondeterministic process $\Omega_{\{\mathtt{b}\}}$. (For a more complete discussion of these two interpretations in a slightly different framework see [Brooks 83]).

The first interpretation seems more natural when the original equation for P is considered (how could a b occur?) and will be adopted in this thesis. It can be justified by noticing that

$$\begin{aligned} P - \mathtt{a} &= \text{rec } X.\, X \\ &= \text{rec } X.\,(X + \Delta_{\{\mathtt{b}\}}) && [+\ \Delta] \\ &= \Delta_{\{\mathtt{b}\}} + \big(\text{rec } X.\,(X + \Delta_{\{\mathtt{b}\}})\big) && [\text{UNFOLD}] \\ &= \textstyle\sum_{i=1}^{\infty} \Delta_{\{\mathtt{b}\}} && \text{Induction} \\ &= \Delta_{\{\mathtt{b}\}} && [+\ +] \end{aligned}$$

By taking advantage of the fact that Δ is the identity for $+$ and using induction, we have shown that the infinite recursion is equivalent to an infinite sum of deadlock elements. But $+$ is idempotent, so this in turn is equivalent to a single deadlock.[†] The transformation discussed here only make sense when the

[†]A table of these and other laws is given in the next section

unguarded recursion has resulted from making an action internal. In general, an unguarded recursive equation will have infinitely many solutions.

Notice that under both interpretations, if the sort of P is simply $\{\mathrm{a}\}$, the resulting process would have an empty sort. This type of process cannot interact with any other and hence is an identity element for the Dot Operator:

$$\Delta_{\emptyset} \bullet Q \quad = \quad Q \qquad \text{for any } Q$$

2.2.11 CIRCAL Laws

Many algebraic laws are derivable from the acceptance semantics that defines CIRCAL's basic operators. Table 2–1 is intended as a reference guide to some of the more useful ones. A more detailed discussion of some their derivations can be found in [Milne 83a].

Name	*CIRCAL Expression*	*Comments*
$[+_I]$	$P + P = P$	Idempotency
$[+_C]$	$P + Q = Q + P$	Commutativity
$[+\ +]$	$P + (Q + R) = (P + Q) + R$	Associativity
$[+\ \Delta]$	$P + \Delta = P$	Identity
$[\oplus_I]$	$P \oplus P = P$	Idempotency
$[\oplus_C]$	$P \oplus Q = Q \oplus P$	Commutativity
$[\oplus\ \oplus]$	$P \oplus (Q \oplus R) = (P \oplus Q) \oplus R$	Associativity
$[+\ \oplus]$	$P + (Q \oplus R) = (P + Q) \oplus (P + R)$	$+$ distributes over $\oplus$
$[\oplus\ +]$	$P \oplus (Q + R) = (P \oplus Q) + (P \oplus R)$	$\oplus$ distributes over $+$
$[\gamma\ \oplus\ +]$	$\gamma P + \gamma Q = \gamma P \oplus \gamma Q$	γ is a guard
$[\bullet_I]$	$P \bullet P = P$	Idempotency
$[\bullet_C]$	$P \bullet Q = Q \bullet P$	Commutativity
$[\bullet\ \bullet]$	$P \bullet (Q \bullet R) = (P \bullet Q) \bullet R$	Associativity
$[\bullet\ \oplus]$	$P \bullet (Q \oplus R) = (P \bullet Q) \oplus (P \bullet R)$	Distributes over $\oplus$
$[\bullet\ +]$	See page 19	
$[\bullet\ \Delta]$	$\Delta_A \bullet \Delta_B = \Delta_{A \cup B}$	
$[-_I]$	$P - \mathbf{a} - \mathbf{a} = P - \mathbf{a}$	Idempotency
$[-_C]$	$P - \mathbf{a} - \mathbf{b} = P - \mathbf{b} - \mathbf{a}$	Commutativity
$[-\ \oplus]$	$(\sum P_i) - \mathbf{a} = \sum(P_i - \mathbf{a})$	
$[-\ +]$	See page 28	
$[\mathrm{rec}_1]$	$\mathrm{rec}\, X.E = E\ [\mathrm{rec}\, X.E /\!/ X]$	Called FOLD/UNFOLD
$[\mathrm{rec}_2]$	$(\mathrm{rec}\, X.\gamma X) - \gamma = \Delta_{L \setminus \gamma}$	$\mathrm{Sort}(\mathrm{rec}\, X.\gamma X) = L$
$[\mathrm{rec}_3]$	$(\mathrm{rec}\, X.(E + \gamma X)) - \gamma = \mathrm{rec}\, X.(E - \gamma)$	
$[\mathrm{rec}_4]$	$(\mathrm{rec}\, X.(E \oplus \gamma X)) - \gamma = \mathrm{rec}\, X.(E - \gamma)$	
$[\Leftarrow]$	$(P \Leftarrow E(P)) = \mathrm{rec}\, X.E(X)$	

Note: Law $[\bullet_I]$ applies only to *deterministic* processes.

Table 2–1: Some CIRCAL laws

2.3 Some Derived Operators

2.3.1 Conditionals

The deterministic summation operator can be used to define conditional resultant states in a manner similar to the way it was used to define value passing. Following the approach of [Milner 83], we can define an **if-then-else** construct for choosing between two resultant states based on the value of a boolean selector function.

Definition 2.3.1 The **if-then-else** construct.

$$\textbf{if } b(v) \textbf{ then } P1 \textbf{ else } P2 \quad =_{def} \quad \sum_{b(v)} P1 + \sum_{\neg b(v)} P2$$

Where $b(v)$ is a boolean function of constants, state variables, and/or communicated values. □

A **case** construct can also be derived by replacing the boolean selection function by one that has multiple values.

2.3.2 Hiding

Sometimes it is necessary to remove a channel that is never used by a process. Using the abstraction operator to do this is meaningless since it makes the port internal and thereby introduces the possibility of nondeterminism. Clearly this is not what is desired. How can a totally unused channel contribute to the overall behavior of the process? A new operator, the *hiding* operator [Milne 83a] is therefore defined to handle this case.

Definition 2.3.2 Hiding Operator

$$P \div \mathrm{a} \quad =_{def} \quad (P \bullet \Delta_{\{\mathrm{a}\}}) - \mathrm{a}$$

□

The deadlock operator blocks all activity on the a channel and the abstraction operator removes it from the sort of the process.

Examples:

$$(\{\mathtt{a}\ \mathtt{b}\}\, P1 + \{\mathtt{c}\}\, P2) \div \mathtt{b} \quad = \quad \{\mathtt{c}\}\, (P2 \div \mathtt{b})$$

$$\{\mathtt{a}\}\, Q1 \div \mathtt{a} \quad = \quad \Delta_{\emptyset}$$

2.3.3 Tight Synchronization

Most well constructed systems are regular structures with highly localized communications. This is increasingly true of VLSI designs, in particular, due to the cost of global connections in terms of wiring area and routing complexity. Since communications in CIRCAL are global by default, it is convenient to derive a composition operator that enforces locality. Not only do the resulting descriptions more accurately reflect the intentions of the designer, but they are also be much clearer to read as irrelevant information is removed as soon as possible. Here is the definition of the new composition operator:

Definition 2.3.3 Tight Composition

$$P \text{ has sort } L, \quad Q \text{ has sort } M$$

$$P \parallel Q \quad =_{def} \quad (P \bullet Q) - (L \cap M)$$

□

The abstraction operator is used to remove any communicating lines which will be given by the intersection of the two sorts. This type of synchronization is called *tight synchronization* since it allows only the two processes to participate in any mutual communication(s).

2.3.4 The Array Operator

Another common feature of VLSI circuits is the replication of primitive elements to form higher order components. A stack of depth D, for instance, is created by placing D copies of a single element stack cell end to end. Single bit storage elements are connected in arrays to form $m \times n$ memories. Since the interconnections are highly localized, we can adapt the tight composition operator to generate single or multi-dimensional arrays.

Definition 2.3.4 The Array Operator

$$\underset{i=1}{\overset{n}{\boxed{C}}} P \quad =_{def} \quad \left(\left(\prod_{i=1}^{n-1} P_i \, [C] \right) \bullet P_n \right) - Left(C_{2\ldots n})$$

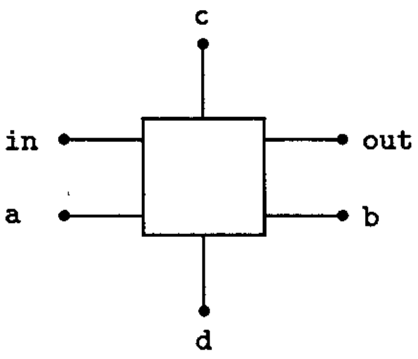
P_i is defined as $P \; [x_i/x, \quad x \in \mathrm{SortOf}(P)]$. □

C, called a *connection set*, is an set of pairs of labels suitable for use by the relabeling operator. Each pair usually specifies an output (input) label that is to be connected to an input (output) label of the next stage. The channels of the last process are left dangling for possible use by the environment. Notice that by default channel labels are indexed. If a channel is to be global to all the processes, it must be explicitly relabelled as such in the connection set. The *Left* function returns a set of all the indexed left members of the relabel pairs. This set is used to hide the connecting channels, except for those of the first stage which must communicate with the outside world (hence the index beginning at 2) and those that are global (unindexed). A sample connection set might look like:

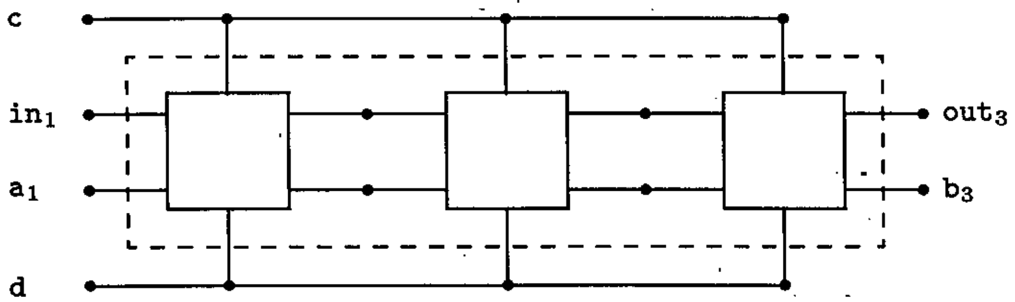
$$\{ \mathtt{in}_{j+1}/\mathtt{out}_j, \; \mathtt{a}_{j+1}/\mathtt{b}_j, \; \mathtt{c}/\mathtt{c}_j, \; \mathtt{d}/\mathtt{d}_j \}$$

Applying it to the box in Figure 2–2(a) yields the array shown in Figure 2–2(b). Note how labels c and d form global communication lines.

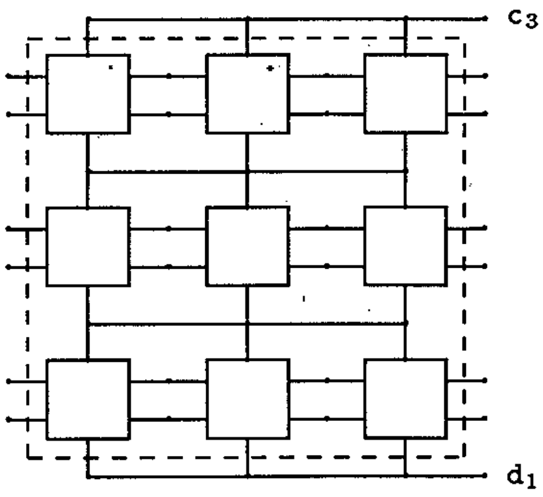
The array operator can be applied an arbitrary number of times to generate multidimensional arrays of processes. Each connecting label must be subscripted by the proper index. Figure 2–2(c) shows how the previous vector of processes



(a) Sample Process P.

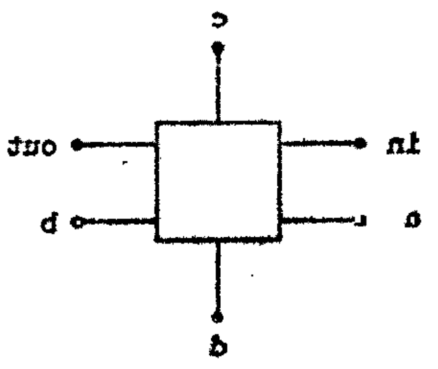


(b) $\underset{j=1}{\overset{3}{\boxed{C}}}\, P_j$

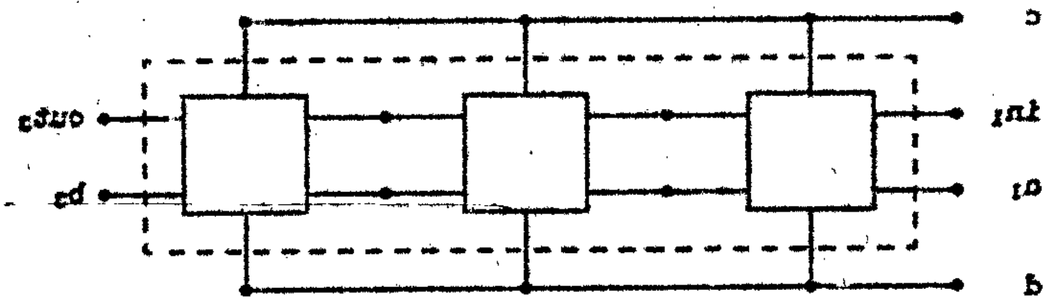


(c) $\underset{i=1}{\overset{3}{\boxed{D}}}\, \underset{j=1}{\overset{3}{\boxed{C}}}\, P_{j,i}$

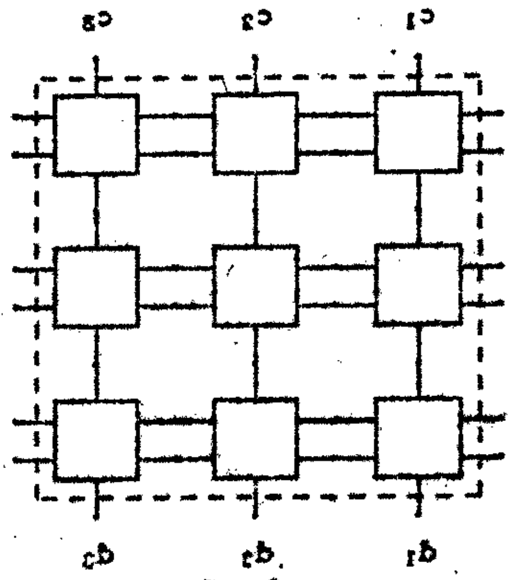
Figure 2–1: Sample applications of the Array Operator



(a) Sample Process P.

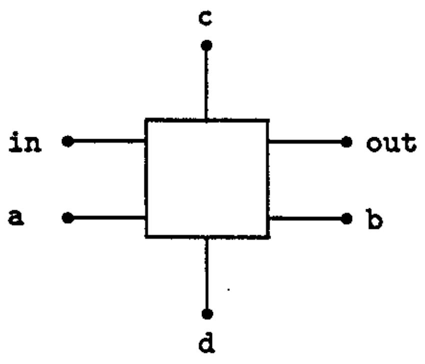


(b) $\boxed{C}_{j=1}^{3} P_j$

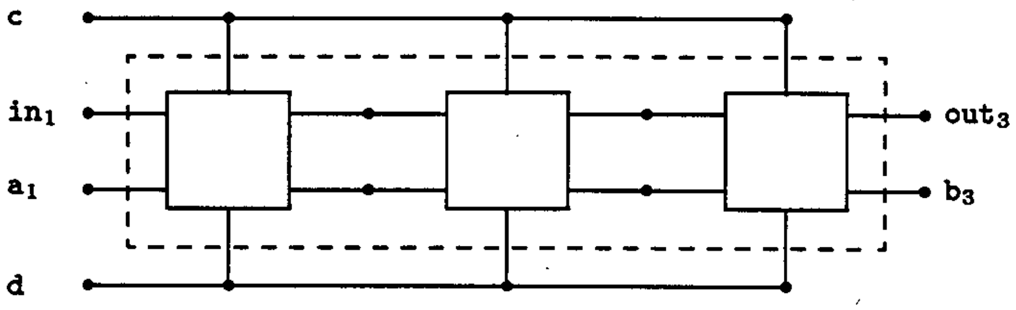


(c) $\boxed{D}_{i=1}^{3} \boxed{C}_{j=1}^{3} P_{ji}$

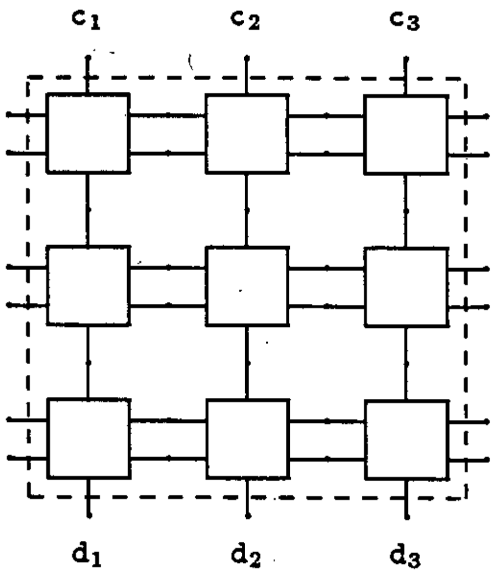
Figure 2–2: Sample applications of the Array Operator



(a) Sample Process P.



(b) $\boxed{C}_{j=1}^{3} P_j$



(c) $\boxed{D}_{i=1}^{3} \boxed{C}_{j=1}^{3} P_{j,i}$

Figure 2–2: Sample applications of the Array Operator

can be extended into a two-dimensional array. The connection set D is defined as:

$$D = \{\mathtt{d}_{i+1}/\mathtt{c}_i\}$$

2.3.5 The "Any Actions" Operator

As we shall see later, it is frequently necessary to synchronize with any subset of a set of actions. A process, for example, may wish to wait for any of or any combination of a number of events to occur before evolving. For convenience, we define an operator that generates the possibilities for us:

$$\mathrm{ANY}(L)P =_{def} \sum_{\gamma_i \in \mathrm{pow}^-(L)} (\cup\gamma_i)\, P$$

The notation pow(L) stands for the powerset (set of all subsets) of the set of label sets L and pow$^-$ is the powerset with the emptyset excluded (pow(L) $\setminus \emptyset$). Consequently, the guards of P will range over all subsets of L, with the exception of the empty set. The union on the guard is needed to "flatten" subsets of the form $\{\{\mathtt{a}\}, \{\mathtt{b\ c}\}\}$ into label-sets like $\{\mathtt{a\ b\ c}\}$. The end result is a deterministic sum of all possible combinations of the events in the generating set.

Notice that all the actions have the same resultants. A more general operator over process sets as well as label sets could be defined, but is not needed in this thesis, so only this restricted case will be considered.

Examples:

$$\begin{aligned}
\mathrm{ANY}(\{\mathtt{a}\})P &= \{\mathtt{a}\}\, P \\
\mathrm{ANY}(\mathtt{a},\mathtt{b})P &= \{\mathtt{a}\}\, P + \{\mathtt{b}\}\, P + \{\mathtt{a\ b}\}\, P \\
\mathrm{ANY}(\{\mathtt{a}\},\{\mathtt{b\ c}\})Q &= \{\mathtt{a}\}\, Q + \{\mathtt{b\ c}\}\, Q + \{\mathtt{a\ b\ c}\}\, Q
\end{aligned}$$

The outer level of braces of the set of label-sets has been removed in the last two examples for clarity.

The Any-Actions operator has the interesting property of acting like an identity for the Dot Operator. Consider a label-set γ that is a subset of the restricted powerset of a set of label-sets A. Then:

$$\mathrm{ANY}(A)\, P \bullet \gamma\, P = \gamma\, P$$

since ANY(A) generates a deterministic choice sum that includes γ as one of its branches. By Law [$\bullet$ +], γ will synchronize with this branch and remove the others.

2.4 Peripheral Considerations

2.4.1 Operator Precedences

To prevent ambiguity in interpreting CIRCAL expressions and to decrease the usage of parenthesis, each of the core operators is assigned a precedence. Table 2–2 lists all the core operators with their symbols and precedences. The highest precedence is 0; the lowest 3.

Operator	Symbol	Precedence
Relabel	[ ]	0
Dot	$\bullet$	1
Choice	$+$	2
Nondeterminism	$\oplus$	2
Abstraction	$-$	2
Definition	$\Leftarrow$	3

Table 2–2: Precedence Table

2.4.2 Sort Determination

It is often necessary to be able to determine the sort of an arbitrary expression. The Dot Operator, in particular, needs to know the sorts of its two arguments in order to perform channel synchronization. Usually, the sort of a process will be declared explicitly with the **ofsort** operator, but sometimes this is inconvenient. A system that expands equations, for example, will often be presented a mixture of operators and atomic state names. Having to indicate the sort of each expression is a needless waste of effort. For these applications a recursive function is presented in Figure 2–3 that can be used to determine a minimum sort for an expression.

$$\begin{aligned}
\mathrm{SortOf}(\ \lambda E\ ) &= \lambda \cup \mathrm{SortOf}(\ E\ )\\
\mathrm{SortOf}(\ \Delta_L\ ) &= L\\
\mathrm{SortOf}(\ E_1 \bullet E_2\ ) &= \mathrm{SortOf}(\ E_1\ ) \cup \mathrm{SortOf}(\ E_2\ )\\
\mathrm{SortOf}(\ E_1 + E_2\ ) &= \mathrm{SortOf}(\ E_1\ ) \cup \mathrm{SortOf}(\ E_2\ )\\
\mathrm{SortOf}(\ E_1 \oplus E_2\ ) &= \mathrm{SortOf}(\ E_1\ ) \cup \mathrm{SortOf}(\ E_2\ )\\
\mathrm{SortOf}(\ E - \mathtt{c}\ ) &= \mathrm{SortOf}(\ E\ ) \setminus \{\mathtt{c}\}\\
\mathrm{SortOf}(\ E\ [\rho]\ ) &= \mathrm{SortOf}(\ E\ )\ [\rho] \text{ where } \rho \text{ is a relabel list}
\end{aligned}$$

Figure 2–3: Algorithm for determining the sort of an expression

Notice that there is no way to stop unbounded recursion. If an expression is recursive, the algorithm will loop forever. The loop can be broken by using the **ofsort** declaration operator that assigns an atom and its expression a sort. The assigned sort must include at least those labels used in the expression and may also include others if necessary. The following rule extends the sort determination function to cope with declarations:

$$\mathrm{SortOf}(\ (P\ \mathbf{ofsort}\ L\ \Leftarrow\ E)\ ) = L$$

Usually a sort will be declared when a process is first defined using the $\Leftarrow$ operator, or it will be obvious from its expression. The above algorithm is intended mainly for mechanical expansion systems that have to determine the sort of an expression from the context. That is why the rules for the choice operators have been included; even though the subterms E_1 and E_2 should have the same sort, it might not be determinable from their defining expressions.

2.5 Synchronization Trees

In [Milner 80], Milner presents an elegant way of picturing behaviors as rooted trees. Each node in the tree corresponds to a state in the evolution of a process; the root node being the first state. The arcs connecting nodes are labelled by non-empty subsets of $\mathcal{L}$ the set of all label names and represent the label-sets of guards. Multiple branches represent a (deterministic) choice of possible outcomes at that point. Expressions can be pictured quite naturally as trees, as Figure 2–4 shows.

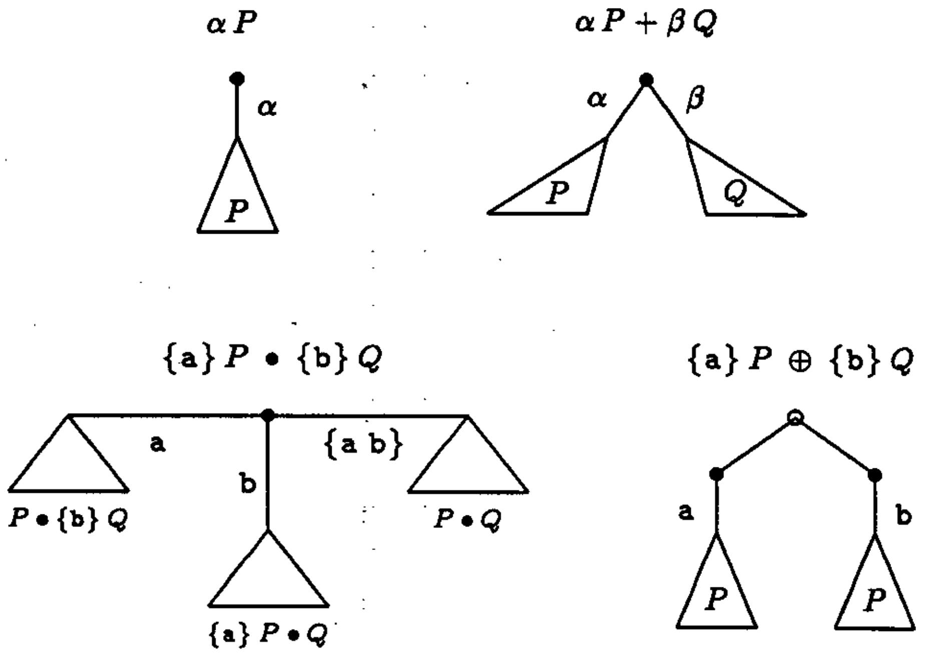


Figure 2–4: Some sample trees

Laws [• +] and [− +] allow all expressions to be represented as summations. Therefore, the expression that describes a process can be pictured as a tree of sequences of events. Finite trees have instances of the termination operator Δ as leaves, whilst recursive terms give rise to infinite trees.

A slight extension is needed to our informal tree semantics to cope with nondeterministic choices. The nodes described above and pictured by a solid circle (•) refer to purely deterministic choice, so we introduce a different type of node pictured by an open circles (∘) for nondeterministic choices. Branches leaving these nodes are not labelled since they stand for hidden moves. Trees having no nondeterministic nodes represent *deterministic* processes.

Definition 2.5.1 Deterministic Processes [Milne 84b].

A process is said to be *deterministic* if it and all its resultants never face a non-deterministic choice. Therefore an expression E describing the process must be of the form:

$$E \Leftarrow \sum_{\substack{i \in I \\ \gamma_j \neq \gamma_k}} \gamma_i \, E_i \qquad \text{and } E_i \text{ is deterministic}$$

□

The $\gamma_j \neq \gamma_k$ qualifier on the summation disallows nondeterministic choices of the form $\gamma P + \gamma Q$. The deadlock operator Δ_L is included by the degenerate case of $i \in \emptyset$.

The choices available to a process when it is in a given state corresponding to a particular node can be collected in a set of sets of actions (and actions are sets of labels!) called the *initial actions set.* A function on expressions can be defined that generates such sets. For example:

$$\underline{\text{initials}}(\{\mathtt{a}\}\, P + \{\mathtt{b}\}\, Q) = \{\{\{\mathtt{a}\}, \{\mathtt{b}\}\}\}$$

Nondeterministic choices require the extra level of nesting, since they are represented by a set of possible deterministic choices:

$$\underline{\text{initials}}((\{\mathtt{a}\}\, P + \{\mathtt{b}\}\, Q) \oplus \{\mathtt{c}\}\, R) = \{\, \{\{\mathtt{a}\}, \{\mathtt{b}\}\}, \{\{\mathtt{c}\}\}\, \}$$

Related to the initial actions set is the *refusal set.* This is the set of actions that belong to the process's sort, but are not found in the initial actions; in other words, the actions that are not part of the choice summation of the current state. Any attempt to communicate with them will deadlock.

Definition 2.5.2 Initial and Refusal Sets

Given a process defined by:

$$P \text{ ofsort } L \Leftarrow \bigoplus_{i \in I} \sum_{\gamma_i \in \mathcal{D}_i} \gamma_i P_i$$

Then the initial actions set of P is calculated by the <u>initials</u> function as follows:

$$\underline{\text{initials}}(P) \quad =_{def} \quad \{ \mathcal{D}_i, \quad i \in I \}$$

and the refusal actions set by:

$$\underline{\text{refusals}}(P) \quad =_{def} \quad \{ x_i \mid x_i = (\text{pow}^-(L) \setminus \mathcal{D}_i), \quad i \in I \}$$

□

The index sets for the deterministic summation contain label-sets (guards), so the <u>initials</u> function will produce a set of sets of label-sets, the members of which correspond to the branches of the nondeterministic summation. The <u>refusals</u> function produces a similar set, except that the members are sets of all possible initial actions ($\text{pow}^-(L)$) with the actual initials removed. Here is a brief example to make this clearer:

$$\begin{aligned}
P \text{ of sort } \{\texttt{a}, \texttt{b}, \texttt{c}\} &\Leftarrow \{\texttt{a}\}\, P' \oplus \{\texttt{b}\}\, P' \\
\underline{\text{initials}}(P) &= \Big\{ \{\{\texttt{a}\}\}, \{\{\texttt{b}\}\} \Big\} \\
\underline{\text{refusals}}(P) &= \Big\{ \{\{\texttt{b}\}, \{\texttt{c}\}, \{\texttt{b c}\}, \{\texttt{a c}\}, \{\texttt{a b c}\}\}, \quad \checkmark\{a b\} \\
&\qquad \{\{\texttt{a}\}, \{\texttt{c}\}, \{\texttt{b c}\}, \{\texttt{a c}\}, \{\texttt{a b c}\}\} \Big\} \quad \{a b\}
\end{aligned}$$

P is a nondeterministic process with two branches, *i.e.*, the nondeterministic index set I has two members $\mathcal{D}_1 = \{\{\texttt{a}\}\}$ and $\mathcal{D}_2 = \{\{\texttt{b}\}\}$. Thus the initial action set of P is a set containing $\mathcal{D}_1$ and $\mathcal{D}_2$ as members. The members of the refusals set are those possible actions not in $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively.

These functions will be used extensively in Chapter 5.

2.6 A Brief Introduction to Acceptance Semantics

The material in this section is on the technical side and can safely be glanced through. It will not be used until Chapter 5, where several proofs require recourse to this level of detail. Further information on acceptance semantics and the proofs of various CIRCAL laws using it can be found in [Milne 83a].

A natural way of finding out how something behaves is to experiment on it. In our framework processes are modelled by the communications that they can participate in. Experiments then take the form of applying stimuli to a process's ports and observing the results. Stimuli are communications of subsets of the sort of the process. They can be *accepted*, in which case the process evolves into another, or *rejected* and the meta-symbol $\star$ produced. This symbol is disjoint from both $\mathcal{L}$, the set of all possible label names and $\mathcal{P}$, the set of all process names. It simply indicates the refusal.

The notion of acceptance can be formalized as a family of *acceptance relations*, indexed by sort and having type:

$$\mathcal{P}_L \times 2^L \quad \longrightarrow \quad \mathcal{P}_L \cup \{\star\}$$

$\mathcal{P}_L$ is the set of all processes of sort L and 2^L is the set of all subsets of L minus the empty set. For $P, Q \in \mathcal{P}_L$ and $\alpha \subseteq L$, the relation is written as

$$P \quad \xrightarrow{\alpha} \quad Q$$

meaning that P accepts α and becomes Q. If P were to refuse a communication β, it would be written as:

$$P \quad \xrightarrow{\beta} \quad \star$$

The *accepts* operator $\longrightarrow$ is a relation and can therefore have multiple values. This is how meaning is given to nondeterminism. A nondeterministic process P can, for example, evolve to both $\star$ and Q under the same stimulus.

Terms are considered equivalent if they have the same sort, accept the same stimuli, and refuse the same stimuli. They must also continue to do so through out their evolution. To capture this notion of comparison, we define an equivalence that is the intersection of ascending indexed relations $\sim_n$. It is always true that $P \sim_0 Q$ for $P, Q \in \mathcal{P}_L$. The equivalence $\sim$ is then defined as $\bigcap_n \sim_n$, where:

$$\begin{array}{llll}
P \sim_{n+1} Q & \iff & \forall\, \alpha \subseteq L,\ \alpha \neq \emptyset & \\
\quad a) & & P \xrightarrow{\alpha} \star \text{ implies } Q \xrightarrow{\alpha} \star & \\
\quad b) & & Q \xrightarrow{\alpha} \star \text{ implies } P \xrightarrow{\alpha} \star & \\
\quad c) & & P \xrightarrow{\alpha} P' \text{ implies } \exists Q' \text{such that } Q \xrightarrow{\alpha} Q' \text{ and } (P' \sim_n Q') & \\
\quad d) & & Q \xrightarrow{\alpha} Q' \text{ implies } \exists P' \text{such that } P \xrightarrow{\alpha} P' \text{ and } (Q' \sim_n P') &
\end{array}$$

The equivalence $\sim$ is a *congruence* with respect to the operators of CIRCAL [Milne 83a]. This means that for all contexts C that can be constructed:

$$P \sim Q \quad \text{implies} \quad C[\![P]\!] \sim C[\![Q]\!]$$

A context is simply an expression with a "hole" in which any CIRCAL term may be placed. This is the equivalence that is implied when we write "=".

2.7 Packaging behaviors

High-level programming languages provide numerous methods (*e.g.*, classes, records, objects, etc.) for packaging data to ease its manipulation and maintain consistency. Behavior expressions encapsulate a great deal of information about a system, so it is valuable to adopt similar techniques in structuring them.

A low level component is typically described in terms of a number of CIRCAL expressions. Each expression corresponds to a state of the behavior and is often assigned using the $\Leftarrow$ operator to an atomic name. In a complicated system built from many components, it is all too easy to intermix th~~i~~ese expressions and in doing so obscure their intended functionality. Changing the description of a component could also prove difficult. Each related expression would have to be tracked down and modified.

To avoid these pitfalls, we introduce a minor syntactic addition called the *part* to our framework. A part is a vector of related state names or assignments that unite to describe the complete behavior of some object. It is similar in application to the procedure construct in most block-structured programming languages. The syntax is as follows:

part *Name* {*sort*} $\langle$

$$\begin{array}{l} \quad \textit{State1} \ \Leftarrow \ \ldots \\ \quad \textit{State2} \ \Leftarrow \ \ldots \\ \qquad \vdots \\ \quad \textit{StateN} \ \Leftarrow \ \ldots \end{array}$$

$\rangle$

The processes associated with the state names can be referenced from outside the part's body by the identifiers *Name.State1*, *Name.State2*, and so forth. By convention, the part is considered to start in the first state of the vector; using the identifier *Name* in an expression is the same as referencing *Name.State1*.

The sort declaration *sort* is a set of channel labels with optional type declarations. Many Hardware Description Languages represent connectivity by passing channel names as parameters to a part. Distinct parts which have a common label as a parameter are considered to be joined on the channel referenced by the parameter. A similar thing can be done here using a *positional morphism.* Instead of writing

Count [n1/out, n2/preset, n3/clear, n4/clock]

when we wish to use the part in a system, we can write

Count [n1, n2, n3, n4]

if the part was declared as

part *Count* {out: **byte**, preset, clear, clock}

The positional morphism is not a set but rather a vector, since the position of a label indicates what channel it names.

Positions in the vector containing an underscore (__) denote hidden channels, *i.e.*, channels that are removed using the hiding (÷) operator. Positional morphisms gives the **part** construct a flavour similar to the commercial Hardware Description Language MODEL [Lattice 82].

The expressions describing the behavior of each state can reference channels not declared in *sort*. These channels are considered to be local to the part's body and cannot communicate with outside processes. Consequently, an abstraction operator is implicitly applied to the first state (and therefore all succeeding ones) that removes the local channels from view when the part's definition is expanded.

Most operators, with the exception of composition, distribute over the body of a part, changing the sort if necessary. Composing part G having n states with H having m results in a part with potentially $n \times m$ states. Some of the states will probably be unreachable when the system is "run", so in practice the composition might be implemented as a lazy evaluation.

Example: To illustrate the utility of the **part** construct, consider a system built from two parts P and Q:

$$S \Leftarrow P \bullet Q$$

The parts are defined as follows

part P $\{\mathtt{start}, \mathtt{a}, \mathtt{b}\}$ $\langle$

$$\begin{aligned} \mathit{Init} &\Leftarrow \{\mathtt{start}\}\, \mathit{P1} \\ \mathit{P1} &\Leftarrow \{\mathtt{a}\}\, \mathit{P2} \\ \mathit{P2} &\Leftarrow \{\mathtt{b}\}\, \mathit{Init} \end{aligned}$$

$\rangle$

part Q $\{\mathtt{start}, \mathtt{c}, \mathtt{d}\}$ $\langle$

$$\begin{aligned} \mathit{Init1} &\Leftarrow \{\mathtt{start}\}\, \mathit{Q1} \\ \mathit{Init2} &\Leftarrow \{\mathtt{start}\}\, \mathit{Q2} \\ \mathit{Q1} &\Leftarrow \{\mathtt{c}\}\, \mathit{Q2} \\ \mathit{Q2} &\Leftarrow \{\mathtt{d}\}\, \{\mathtt{e}\}\, \mathit{Init1} \end{aligned}$$

$\rangle$

By our convention, this is simply an abbreviation for:

$$S \Leftarrow P.\mathit{Init} \bullet (Q.\mathit{Init1} - \mathtt{e}).$$

The e channel is implicitly abstracted away since it not referred to in the outer sort declaration.

Now suppose, for some unknown reason, we wish to change Q so that it starts up in state *Q.Init2*. In the non-abbreviated form, this means going back to the definition of S and changing it to

$$S \Leftarrow P.Init \bullet (Q.Init2 - \mathrm{e})$$

In the abbreviated version, on the other hand, one simply exchanges the positions of *Q_Init1* and *Q_Init2* in the part declaration. In an elaborate version of S that references Q many times, this can save a considerable amount of effort, as well as reducing the opportunities for introducing errors. A degree of modularity has been introduced and with it a way of localizing changes that need to be made to a system's behavior.

Here is a simple part that inputs a starting value, outputs every integer between this value and 0, and then terminates.

Example:

```
part CountDown {start, c:int} (
    Start    ⇐ {start▷x} Count(x)
    Count(x) ⇐ if x = 0 then Δ{start, c}
               else {c◁x} Count(x − 1)
)
```

2.8 Modeling Systems

Constructing a behavioral model of a system is not simply a matter of identifying the states that it can evolve into and listing the transition events, but also involves choosing a *modeling style*. The calculus provides operators for describing and manipulating behaviors in a certain framework, but says nothing about mapping real-world behaviors onto this framework. This section considers some of the general issues that arise when trying to construct a model of a real-world object.

2.8.1 Representation Style

The behavior of most real-world systems is based in the interactions of continuously varying phenomena. A discrete event oriented calculus such as CIRCAL needs to make some approximations in order to deal with these continuous values. This is done by sampling them at some granularity of time to produce a sequence of value-events. Two possibilities exist for representing these sequences in the calculus. The first method simply writes the sequence as a list of guards that continually broadcast the current value of the parameter. In the second method, only changes to the value of the parameter are indicated. Parameters that change infrequently can be represented much more compactly using this method.

To see the difference between the two approaches, consider a model of a boolean signal line that continuously outputs either a 0 or a 1 based on a controlling event (the first method):

$$\begin{aligned} \mathit{Line}(1) &\Leftarrow \{\texttt{control}\}\,\mathit{Line}(0) + \{\texttt{out}\triangleleft 1\}\,\mathit{Line}(1) \\ \mathit{Line}(0) &\Leftarrow \{\texttt{control}\}\,\mathit{Line}(1) + \{\texttt{out}\triangleleft 0\}\,\mathit{Line}(0) \end{aligned}$$

The line and a similar one are then connected to an ideal boolean OR box (Figure 2–5). The box is defined by:

$$\mathit{OR} \Leftarrow \{\texttt{out}\triangleright x \quad \texttt{out2}\triangleright y \quad \texttt{sig}\triangleleft(x \vee y)\}\,\mathit{OR}$$

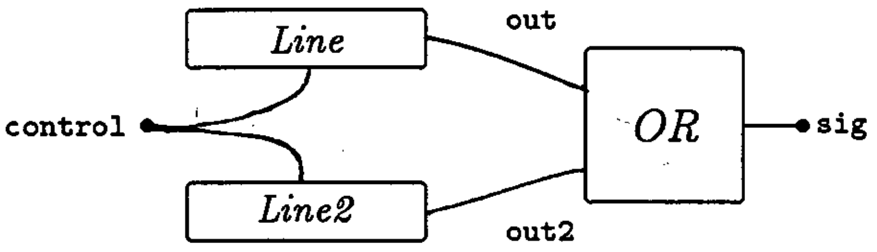


Figure 2–5: OR box with two value generating boxes attached

In the absence of a control signal, *Line* continually outputs its current value to *OR* which uses this in turn to continually (and instantaneously) calculate its logical OR with another value given on the out2 line.

On the surface, this appears to be perfectly acceptable. The definitions are compact and easy to understand. Further thought, however, reveals some problems. If a value has already been communicated to another agent, sending it again cannot really be considered an "event" worth noting. The receiving process must be able to accept the communication at all times or deadlock will result. This means that size of process descriptions increases as the number of channels increases, even if most of the channels have no relevance to a particular state. In addition, if one is interested in the change of a value, as is frequently the case in digital systems, a state variable is required to remember the previously communicated value so that it can be compared with the current ones. It rapidly ~~gets~~ becomes cumbersome to detect~~ing~~ value changes on a number of channels. A final objection to this approach can be seen by noting that the $\vee$ computation should only have to be performed when the value of a signal changes, not at every instant. This could, for example, be detrimental to the performance of a software implementation of the OR box.

Accordingly, we modify the *OR* box example so that the state of the line is remembered and only changes to this value are accepted as input.

$$\begin{aligned}
Line(1) &\Leftarrow \{\mathtt{control}\}\,\{\mathtt{out}\triangleleft 0\}\,Line(0)\\
Line(0) &\Leftarrow \{\mathtt{control}\}\,\{\mathtt{out}\triangleleft 1\}\,Line(1)
\end{aligned}$$

$$\begin{aligned}
OR(0,0) &\Leftarrow \{\mathtt{out}\triangleright 1\ \ \mathtt{sig}\triangleleft(1\vee 0)\}\,OR(1,0)\\
&\quad+\{\mathtt{out2}\triangleright 1\ \ \mathtt{sig}\triangleleft(0\vee 1)\}\,OR(0,1)\\
&\quad+\{\mathtt{out}\triangleright 1\ \ \mathtt{out2}\triangleright 1\ \ \mathtt{sig}\triangleleft(1\vee 1)\}\,OR(1,1)\\
OR(0,1) &\Leftarrow \{\mathtt{out}\triangleright 1\ \ \mathtt{sig}\triangleleft(1\vee 1)\}\,OR(1,1)\\
&\quad+\{\mathtt{out2}\triangleright 0\ \ \mathtt{sig}\triangleleft(0\vee 0)\}\,OR(0,0)\\
&\quad+\{\mathtt{out}\triangleright 1\ \ \mathtt{out2}\triangleright 0\ \ \mathtt{sig}\triangleleft(1\vee 0)\}\,OR(1,0)\\
OR(1,0) &\Leftarrow \text{Symmetric case to the above}\\
OR(1,1) &\Leftarrow \{\mathtt{out}\triangleright 0\ \ \mathtt{sig}\triangleleft(0\vee 1)\}\,OR(0,1)\\
&\quad+\{\mathtt{out2}\triangleright 0\ \ \mathtt{sig}\triangleleft(1\vee 0)\}\,OR(1,0)\\
&\quad+\{\mathtt{out}\triangleright 0\ \ \mathtt{out2}\triangleright 0\ \ \mathtt{sig}\triangleleft(0\vee 0)\}\,OR(0,0)
\end{aligned}$$

At the cost of a considerable increase in complexity, we now have a purely event driven model. The four possible states of the *OR* are explicitly listed in a rather naïve fashion. In addition, the **sig** output line is asserted for every change in the inputs, even when the calculated value is the same as the one last output. This shows that the above equation for the OR box is still not a true event driven model. What changes must be made to accomplish this?

The processes that drive the inputs to the OR box, *Line* and *Line2*, only output value changes. This means that we can input the values without worrying about absorbing spurious ones. To prevent such spurious values from being produced on the **sig** channel, a conditional guard is used:

$$\begin{aligned}
OR(in,\ in2,\ last) \ \Leftarrow\ & \{\mathtt{out}{\triangleright}x \ \ \mathbf{if}\ g(x, in2, last)\ \mathbf{then}\ \mathtt{sig}{\triangleleft}(x \vee in2)\} \\
& \qquad OR(x,\ in2,\ (x \vee in2)) \\
& + \{\mathtt{out2}{\triangleright}y \ \ \mathbf{if}\ g(y, in, last)\ \mathbf{then}\ \mathtt{sig}{\triangleleft}(in \vee y)\} \\
& \qquad OR(in,\ y,\ (in \vee y)) \\
& + \{\mathtt{out}{\triangleright}x \ \ \mathtt{out2}{\triangleright}y \ \ \mathbf{if}\ g(x, y, last)\ \mathbf{then}\ \mathtt{sig}{\triangleleft}(x \vee y)\} \\
& \qquad OR(x,\ y,\ (x \vee y)) \\
g(x,\ y, z) \ \ =_{def}\ & (x \vee y \neq z)
\end{aligned}$$

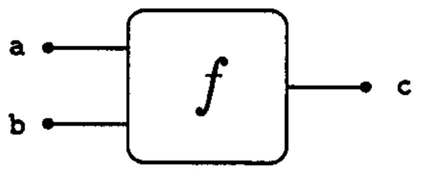
A definite improvement! The equation has been made considerably smaller at the expense of some new notation. Putting the conditional that tests whether or not to output a value inside the guard is legal since it amounts to adding a predicate to the summations that define the input channels. Even with this new trick, however, the expression is still much more complex than the continuously communicating version. This demonstrates that the event interpretation is not very good for capturing the behavior of purely functional units. On the other hand, the continuous interpretation is just as unwieldy to use for representing events such as clock edges. Most of the systems that will be examined later on will be event based rather than functionally based, so the former is the interpretation that will be adopted.

2.8.2 Generic Box Components

Many hardware components can be thought of as black-boxes that take inputs, perform some function on them, and output the results. The Boolean combinational elements AND, OR, NOT, and their derivatives are prime examples.

If we examine the description for an OR box given in the previous section, we notice that the only thing that determines its function is the presence of the $\vee$ operator. An AND Box could be generated by simply changing this to a $\wedge$, <u>without</u> changing the CIRCAL expression in any way. This suggests that the expression describes a two-input one-output *black-box*, that can be *personalized* to produce a required functional element.

If the function calculated by the box is ignored and the structure alone is considered, we see that it consists of an enumeration of all possible input events. This brings to mind the "Any Actions" operator ANY discussed in Section 2.3.5. If we combine the use of this operator and change the box to have some unspecified delay, a very simple expression results. A picture of the box and the simplified CIRCAL expression describing its behavior are shown in Figure 2–6. Note the use of the notation that causes input channels to bind a variable with the same name as the channel. Thus $\{\mathtt{in1}\triangleright\}$ would input a value on channel in1 and bind it to the variable *in1*. This notation can greatly simplify complex device descriptions.



(*a*)

$$
\begin{aligned}
&\mathit{Box2\_1}(\mathit{in1}, \mathit{in2}, \mathit{out}) \Leftarrow \\
&\qquad \mathrm{ANY}(\{\mathtt{in1}\triangleright\}, \{\mathtt{in2}\triangleright\}) \\
&\qquad\quad (\textbf{if } f(\mathit{in1}, \mathit{in2}) \neq \mathit{out} \textbf{ then } \{\mathtt{out}\triangleleft f(\mathit{in1}, \mathit{in2})\}) \\
&\qquad\qquad \mathit{Box2\_1}(\mathit{in1}, \mathit{in2}, f(\mathit{in1}, \mathit{in2}))
\end{aligned}
$$

(*b*)

Figure 2–6: Generic two-input one-output box. (*a*) Spatial properties (*b*) CIRCAL description.

The equation in Figure 2–6(*b*) has a subtle feature that should be examined more closely. The Any-Actions operator generates a choice sum of all possible input events on the two channels. Now, what will happen if the state variables initially have the values $in1 = 1$, $in2 = 2$ and $out = f(1,2)$, and a value of 3 is communicated to the `in1` channel? The value will get bound to the *in1* variable, *replacing* its previous value of 1. Similarly, communications on `in2` will rebind *in2*.

The box is personalized by renaming the channels and the transfer function f. In this manner, we obtain the CIRCAL expression for any two argument function. Unlike the OR box discussed in the previous section, however, the value takes time to calculate and is output some unspecified time after an input change.

The personalization can be taken a step further if an n-input generic box is defined. Not only can the transfer function be easily selected, but so can the number of inputs. Here is the description of such a box:

$$\begin{aligned} BoxN\_1(\tilde{x}, out) \;\Leftarrow\; & \mathrm{ANY}(\{\mathtt{in}_i\triangleright\}, i = 1 \ldots n) \\ & \quad (\text{if } f(\tilde{in}) \neq out \text{ then } \{\mathtt{out}\triangleleft f(\tilde{in})\}) \\ & \quad\quad BoxN\_1(\tilde{x}, f(\tilde{in})) \\ \text{where } \tilde{in} \;=\; & in_1, \ldots, in_n \end{aligned}$$

The two-input box had three $(2^2 - 1)$ terms in the choice expression that described its behavior. An n-input box will have $2^n - 1$ possible input combinations. This means that the behavior of an 8-input AND gate, for instance, would contain 256 terms in its choice-sum! In the absence of the derived operators and the shorthand notation, these values would have to be enumerated by hand. This illustrates the importance of being able to easily introduce new ways of representing the large amount of information that is needed to describe even the simplest components in any detail. Mechanical manipulation aids soon become vital since the complexity of expressions grows exponentially.

2.8.3 Design Style

There are three approaches to designing a system: top-down, bottom-up, and a mixture of the two. Top-down means that the system is specified in a macroscopic form and then gradually broken down into sub-systems, sub-sub-systems, and so forth until the primitive component level is reached. This is the approach generally advocated by structured programming proponents and to a lesser extent by hardware designers. It is successful mainly when the design style is highly stylized, such as in synchronous register transfer systems [Winkel 80].

Bottom-up is precisely the opposite. Primitive components are combined to yield a higher level module which can in turn be combined with other modules or components to form a yet higher level entity. This design style is considered bad practice, but is still quite common.

In practice, neither approach is followed rigidly. Even the best top-down designs are influenced by "low-level" considerations. Availability and/or cost (on some arbitrary scale) of bottom-level components can greatly affect global design considerations. As constraints are satisfied, the composition of a system may change radically. Being able to change the design at any level as easily and quickly as possible is vital.

Both styles have one thing in common, namely that they are modular. Modules are collections of elements grouped together to form a particular functional unit. Since elements may include other modules, the end result is a tree structure of module relationships whose bottom-most entries ("leaves") are the primitive components. The two design styles are simply different methods for generating this tree. This approach, as opposed to completely specifying the system at one level, is known as *hierarchical design*. In recent years, many people have demonstrated the benefits of designing hardware in such a fashion [Rowson 80, Buchanan 80, Mead 80]. This methodology has been particularly successful in managing the complexity of VLSI layout design. Partitioning a system into functional blocks is the same whether or not the end result is layout or a behavioral description. We will therefore describe our systems in as hierarchical a fashion as possible. Whether the hierarchy is constructed bottom-up or top-down is a purely personal choice.

2.9 Summary

This chapter has introduced a calculus for describing the behavior of concurrent systems in terms of sequences of events. The basic calculus, due to Milne [Milne 83a], draws upon a small number of operators to provide a rich framework for reasoning about the interactions of processes. As the examples in Milne's paper show, however, use of these operators alone to describe even simple concepts can lead to unnecessarily complicated expressions. Consequently, some new operators were introduced and restrictions were placed on the use of existing ones (*e.g.*, dynamic relabelling is not allowed). The new operators include, among others, a mechanism for passing values through channels and a way of accepting any of a number of communications. These two are crucial in managing the exponential explosion in possible inputs to large processes. Along with the new operators, a syntactic construct was defined that eases the manipulation of sets of equations describing a single object (the **part** construct).

With the framework established, consideration was turned to developing a representation for real-world phenomena in an event-driven manner suitable for use in CIRCAL. The case was made for events indicating value changes, rather than events that continuously assert the value of a channel. An example that contrasted the two approaches also led to the concept of *generic boxes*. The boxes provide a general structure for han$\overset{l}{d}$ing the events needed to drive a functional unit. The actual function calculated by the box is orthogonal to its interaction with other processes, so it can be defined by relabelling the template process.

Putting these ideas together suggests the following guidelines to describing behaviors:

- CIRCAL expressions describing the behavior of a single component are packaged up into one object using the **part** construct.

- Only changes in values are represented as events, thus upholding the character of the calculus. Stable values will not be signalled continuously.

- Locality will be exploited as much as possible to generate modular descriptions. The results will be combined in a hierarchical fashion.

- Generic box components will be used as often as possible for describing primitive functional units. The boxes provide descriptions of n-input, m-output components with a transfer function f. They are personalized using the relabelling operator to provide a particular functional block.

2.9.1 Contributions of this Chapter

Several additions were made to the CIRCAL calculus of Milne. These included a mechanism for passing values along channels, a conditional construct and an operator for generating arrays of interconnected processes. Some ambiguities in existing operators were resolved, such as the instantaneous communication of interdependent functions using the Dot operator and the semantics of hiding a singleton guard in a recursive equation. Two forms of packaging behaviors were then developed (the **part** and generic boxes concepts) in order to simplify the creation of behavioral descriptions. Finally, two modelling styles were compared and contrasted, resulting in the adoption of a purely event driven approach.

Chapter 3

Time, Clocks and Delays

Towards the end of the previous chapter, we encountered several fairly large expressions that described the behavior of basic combinational elements. Although notation was developed that simplified these expression, they are still quite complicated considering the simplicity of their function. Consequently, there must be a compelling reason for choosing an event-driven representation over other calculi—such as Boolean algebra—which produce far more succinct descriptions. The reason is, quite simply, time. Describing behavior in terms of events leads naturally to reasoning about the relationships between these events in time. Considering such relationships allows one to ask performance-related questions about a design (*e.g.,* How much time will it take for this system to calculate an answer? Will this module provide a valid output during a single clock cycle?). Performance issues are particularly important in hardware designs, since they can determine whether a design will function at all.

The chapter begins by showing how explicit time may be introduced into the calculus. Several approaches are developed and their application to particular kinds of problems discussed. Building on the basic approach, consideration is turned to multiple levels of clocks and temporal intervals in general. With time comes the concept of delay, and so several notions of delay are examined next. The effects of delay in certain common feedback arrangements (storage elements) are considered by way of example.

3.1 Time

Recall that the calculus only orders events in time. There is no way of quantifying when they occur in relation to each other. Guarding imposes a simple *before* and *after* ordering on the events in a sequence. For instance the expression $\{\texttt{a}\}\{\texttt{b}\}P$ implies that if the event a occurs then it will do so before b and nothing more. Reasoning about real world events, particularly hardware ones, often requires a means of measuring the passage of time between them. One way to accomplish this is to mark the real time axis with a series of events called *ticks*. The interval between ticks is usually assumed to be uniform and have an arbitrary length ϵ. The ticks are generated by a central process call the *Universal Clock*.

The Universal Clock sends tick events to all timeable processes in the *process universe*. A process universe is the collection of all processes that have the same tick label in their sorts (*i.e.*, ones that are begin examined at the same granularity of time). All actions produced by these processes must occur "at the exact same instant" as a tick. Statements about the separation of actions in time may then be made by counting the number of ticks that occur between them. The length of the interval between ticks ϵ must be chosen to be smaller than the smallest separation between events of interest. Figure 3–1 shows how an arbitrary function of time is sampled by ticks labeled t, including a transient value that was missed because the granularity of time was too great.

The Universal Clock concept presents a convenient way of comparing otherwise asynchronous events. Moore used a similar synchronizing technique to deal with asynchronous transitions in state diagrams [Langdon 74].

Milne [Milne 83b] shows how adding a Universal Clock to the calculus involves no fundamental changes. A tick label, t, is introduced into the sort of all processes by forcing each communication to belong to a label-set containing t. For example, to indicate the fact that event b occurs two ticks after event a, we might write:

$$\begin{aligned} Q &\Leftarrow \{\texttt{a t}\}\{\texttt{t}\}\{\texttt{b t}\}\,Q \\ \mathit{UniversalClock} &\Leftarrow \{\texttt{t}\}\,\mathit{UniversalClock} \end{aligned} \tag{3.1}$$

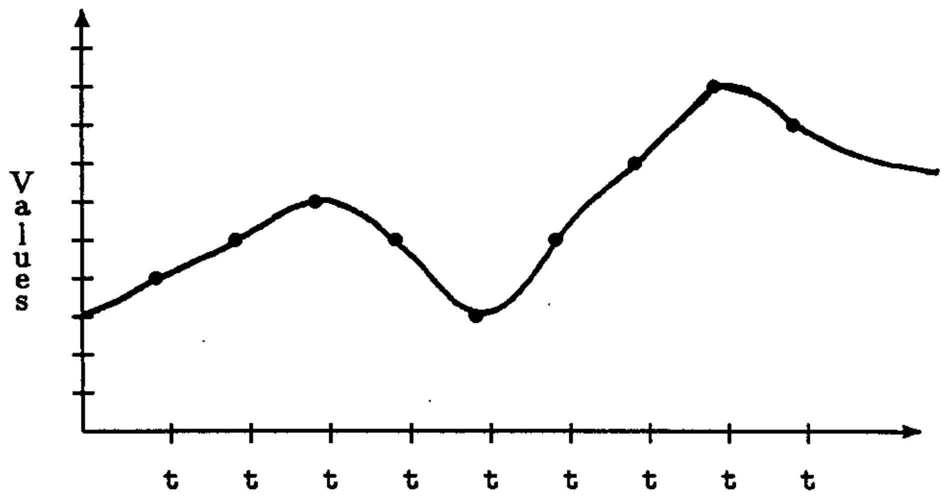


Figure 3–1: Sampling the real time axis.

The *UniversalClock* process is an example of what will be called a *virtual process.* Virtual processes are never actually implemented but are used instead to represent extra information about the system. In this case, *UniversalClock* simply generates the stream of t events that force the progression of the rest of the system. Strictly speaking, its presence is not required since guards containing a t will synchronize when composed using the Dot Operator.

If the passage of time is abstracted away, the original asynchronous nature of the calculus should reassert itself. Therefore, as a crosscheck on this representation of time, let us see what happens when the ticks in Equation 3.1 are removed using the abstraction operator:

$$\begin{aligned} R &\Leftarrow Q - \mathtt{t} & \\ &= (\{\mathtt{a\ t}\}\,\{\mathtt{t}\}\,\{\mathtt{b\ t}\}\,Q) - \mathtt{t} & \text{defn. } Q \\ &= \{\mathtt{a}\}\ (\{\mathtt{t}\}\,\{\mathtt{b\ t}\}\,Q) - \mathtt{t} & [-\,+] \\ &= \{\mathtt{a}\}\,\{\mathtt{b}\}\,(Q - \mathtt{t}) & [-\,+] \text{ twice} \\ &= \{\mathtt{a}\}\,\{\mathtt{b}\}\,R & \text{defn. } R \end{aligned}$$

Precisely the desired result. The information that b occurred two ticks after a has been lost, leaving only the before/after ordering of the asynchronous calculus.

What happens when a process wishes to wait for an event? If the communication a is offered on a certain tick and there are no accepting communications at that same tick, deadlock could result. An "idler loop" is needed to absorb ticks until the a communication is accepted. This is implemented with the deterministic choice operator as follows:

$$Q \Leftarrow \{\mathtt{a\ t}\}\, P + \{\mathtt{t}\}\, Q \tag{3.2}$$

Note that this is similar to the "delay until" or "wait for" operator δ in SCCS [Milner 83], but a more mneumonic name will be used as well as a subscript to indicate which tick label is being waited upon:

$$Q \Leftarrow \mathrm{WAIT}_{\mathtt{t}}(\, \{\mathtt{a\ t}\}\, P\,)$$

The intended meaning is that Q will idle, absorbing ticks (perhaps forever), until an a communication takes place.

Definition 3.1.1 The "Wait For" Operator

$$\mathrm{WAIT}_{\gamma}(\, P\,) \quad =_{def} \quad \mathrm{rec}\ X.\, (\gamma\, X + P)$$

□

As another check on this scheme for representing time, let us see what happens if we remove explicit time from a process that is waiting for an event. Consider Equation 3.2 with t abstracted away:

$$\begin{aligned} Q - \mathtt{t} &= (\{\mathtt{t}\}\, Q + \{\mathtt{a\ t}\}\, P) - \mathtt{t} \\ &= (Q - \mathtt{t}) \oplus ((Q - \mathtt{t}) + \{\mathtt{a}\}\, (P - \mathtt{t})) && [- +] \end{aligned}$$

We have encountered unguarded recursion again! Last time we encountered it (Section 2.2.10), it was equated with deadlock since all actions by the process were happening internally, blocking all external communications. The situation here, however, is a bit different in that the recursion appears in two different summations. Considering the nondeterministic summation first, we find that:

$$\begin{aligned} Q - \mathtt{t} &= (Q - \mathtt{t}) \oplus R \\ &= (Q - \mathtt{t}) \oplus R \oplus R && [\mathrm{UNFOLD}] \\ &= \sum_{i=1}^{\infty} R && \text{Induction} \\ &= R && [\oplus_I] \end{aligned}$$

As before, the recursion was turned into an infinite summation and the idempotency property of $\oplus$ exploited to collapse the sum. This result will be referred to as law [rec$_4$]. A similar argument is used to remove the recursion in the deterministic summation to produce law [rec$_3$]:

$$\begin{aligned} Q - \mathtt{t} &= (Q - \mathtt{t}) + \{\mathtt{a}\}\,(P - \mathtt{t}) && \text{Above} \\ &= \sum_{i=1}^{\infty} \{\mathtt{a}\}\,(P - \mathtt{t}) && \text{Induction} \\ &= \{\mathtt{a}\}\,(P - \mathtt{t}) && [+_I] \end{aligned}$$

An asynchronous expression results as expected. Any communication on the a channel will "pre-empt" the internal ticks and cause the process to evolve.

3.1.1 Time in CIRCAL and SCCS

We now digress slightly to explore the similarities and differences between CIRCAL and the closely related calculus SCCS. SCCS (Synchronous Calculus for Communicating Systems) has a concept of time roughly similar to the one presented here. All actions are assumed to occur simultaneously with an implicit tick action (hence the "Synchronous"). Points in time when no events happen are indicated by the special action **1**. Like CIRCAL, SCCS has a guarding combinator, which is written as a colon prefixed by a label and followed by a resultant.

Examples:

$a : b : P$	Perform a followed by b in the next unit of time before becoming P.
$a : \mathbf{1} : b : P$	Perform a, delay one unit of time, and then continue as above.
$P \Leftarrow \mathbf{1} : P$	Idle indefinitely

Other operators also have their counterparts in SCCS. Choice, written as +, is very similar to CIRCAL's deterministic choice operator. Parallel composition $\times$, however, has a quite different effect from the Dot Operator. It distributes over SCCS's + and has the following properties for actions:

$$a : P \times b : Q = ab : (P \times Q) \qquad a : P \times \mathbf{1} : Q = a : (P \times Q)$$

Where $ab : R$ is equivalent to writing {a b} R in CIRCAL. Notice that $\times$ does not rely on the sorts of its arguments to determine synchronization, unlike the Dot Operator. An expansion rule for $\times$ with respect to $+$ can be written as:

$$\begin{aligned} P &\Leftarrow \sum_i \alpha_i : P_i \\ Q &\Leftarrow \sum_j \beta_j : Q_j \\ P \times Q &=_{def} \sum_{i,j} \alpha_i\beta_j : (P \times Q) \end{aligned} \tag{3.3}$$

Contrast this with the expansion rule for the dot operator:

$$\begin{aligned} R &\Leftarrow \sum_i \alpha_i\, R_i \text{ of sort } L \\ S &\Leftarrow \sum_i \beta_j\, S_j \text{ of sort } M \\ R \bullet S &=_{def} \sum_{\alpha_i \cap M = \emptyset} \alpha_i\, (R_i \bullet S) \\ &\quad + \sum_{\beta_j \cap L = \emptyset} \beta_j\, (R \bullet S_j) \\ &\quad + \sum_{\alpha_i \cap M = \beta_j \cap L} (\alpha_i \cup \beta_j)\, (R_i \bullet S_j) \end{aligned}$$

In the synchronized process-universe discussed in the previous section, all actions belong to a label-set containing the tick label t. The sorts of the two processes, L and M, will also contain t, so $L \cap M \neq \emptyset$. Furthermore, $\alpha_i \cap \beta_j \neq \emptyset$ for any two guards α_i and β_j (both contain the tick label). Using these facts, a simplified version of $\bullet$ (written with the tick label as a subscript) can be defined as:

$$R \bullet_{\mathtt{t}} S = \sum_{\alpha_i \cap M = \beta_j \cap L} \alpha_i \cup \beta_j\, (R_i \bullet S_j) \tag{3.4}$$

This equation bears a striking resemblance to Equation 3.3. Both have the same form of a simultaneous action followed by a single resultant. Carrying this analogy toofar, however, is dangerous. In SCCS, events are not broadcast as they are in CIRCAL. Instead, each action has a "direction" so that when two complementary labels synchronize, a single point-to-point communication takes place. Should an event happen on the same tick as several instances of its complements, the choice of which one actually synchronizes is nondeterministic. This communication semantics is characterized by its *dynamic* nature, unlike that of CIRCAL which determines synchronization based on static sorts. Because of this, deadlock is not explicitly represented in SCCS. Compare these two examples:

$$(a : b : P) \times (b : a : Q) \ = \ ab : ab : (P \times Q) \qquad \text{(SCCS)}$$
$$(\{\mathtt{a}\ \mathtt{t}\}\,\{\mathtt{b}\ \mathtt{t}\}\,R) \bullet_{\mathtt{t}} (\{\mathtt{b}\ \mathtt{t}\}\,\{\mathtt{a}\ \mathtt{t}\}\,S) \ = \ \Delta_{\{\mathtt{a},\,\mathtt{b},\,\mathtt{t}\}} \qquad \text{(CIRCAL)}$$

The SCCS process P has no clue that Q will eventually perform an a event, so it happily produces one at the same time that Q is producing the b event. R, on the other hand, knows that S has the potential to do an a, so it waits. S, meanwhile, knows that R can potentially produce a b, so it too waits. The result is deadlock.

Both approaches to modeling synchronous process-universes yield similar results, modulo communication semantics. This is as it should be since CIRCAL and SCCS come from similar roots. The important difference, from the point of view of this quick comparison, is that the event mechanism in SCCS is truly global. All events happen at the same time as a universal tick.[†] As we shall see in a later section, CIRCAL is a bit more flexible than this. More than one tick label can be defined to yield multiple process-universes, each with a different granularity of time. This simplifies the description of systems having a several levels of clocking, as is often the case in digital designs.

3.1.2 Enumerated Time

Sometimes an application will need to generate an event at a particular time in the future. An alarm clock, for example, might wish to signal a wake up call at time $t = 100$ as measured by some arbitrary time units. One way to do this is to count all the ticks that occur from the time when the alarm is initialized. The wake-up signal can then be generated after a hundred have passed. This is needlessly complicated since it involves adding an extra state variable to each such process to keep track of the passage of ticks. A more elegant solution is to

[†]Actually, an asynchronous calculus (ASCCS) can be derived from SCCS that gets around this. The result resembles the aynchronous nature of raw CIRCAL.

let the Universal Clock to do the time keeping for us by attaching an integer tag value to each tick as it is broadcast. Waiting for a particular time then simply involves synchronizing with a tick subscripted by the appropriate tag value.

A mechanism—similar to the "Wait For" WAIT_x operator—is needed to indicate the passage of time while waiting for the desired tick. There are two ways of doing this. The first involves modifying the wait-for operator so that it absorbs all unwanted tag values. The second method relies on a clever definition of the sort of the waiting process. If the sort excludes all ticks with a tagged value less than that required, the semantics of the Dot Operator will cause the process to wait when composed with the Universal Clock. This happens because the Universal Clock contains the desired tick in its sort, but must pass through all the preceding tag values before reaching it. These tagged ticks will be independent of the waiting process's sort and will therefore occur without its knowledge. This method has the advantage of being concise, but because it relies on the static sort, cannot be used for events that happen dynamically (*e.g.,* at time $t + 100$).

To illustrate the two approaches and to compare them with the plain tick representation of time, we now consider a simple example of an alarm clock that produces a wakeup signal at time $t = 100$ (arbitrary units). First we need to define the Universal Clock that broadcasts the tagged ticks:

$$
\begin{array}{l}
\textbf{part } \textit{UniversalClock}\ \{\texttt{startclock}, \texttt{t} : \mathcal{N}\}\ \langle \\
\qquad \begin{array}{rcl}
\textit{StartClock} & \Leftarrow & \{\texttt{startclock}\}\ \textit{BIGBEN}(0) \\
\textit{BIGBEN}(\textit{time}) & \Leftarrow & \{\texttt{t}\triangleleft\textit{time}\}\ \textit{BIGBEN}(\text{succ}(\textit{time}))
\end{array} \\
\rangle
\end{array}
$$

The universe's time begins with a value of 0 when the `startclock` signal is generated. For then onward, the current time is broadcast on the `t` channel, which has as its type the natural numbers ($\mathcal{N}$).

Now we see how the alarm clock would be defined in a pure tick universe. This is done by ignoring the tags attached to the ticks and counting them "by hand."

$$\textbf{part } \mathit{Alarm\_pt}(0)\ \{\texttt{wakeup},\ \texttt{t} : \mathcal{N}\}\ \langle$$
$$\mathit{Alrm}(x) \Leftarrow \textbf{if } x = 100 \textbf{ then } \{\texttt{t}\triangleright\ \ \texttt{wakeup}\}\ \mathit{Alrm}(\mathrm{succ}(x))$$
$$\textbf{else } \{\texttt{t}\triangleright\}\ \mathit{Alrm}(\mathrm{succ}(x))$$
$$\rangle$$

The guard $\{\texttt{t}\triangleright\}$ will synchronize with any tagged tick since it is short-hand for $\{\texttt{t}\triangleright t\}$.

Using the first method discussed above, any tick not labelled by the tag 100 is ignored:

$$\textbf{part } \mathit{Alarm\_m1}(0)\ \{\texttt{wakeup},\ \texttt{t} : \mathcal{N}\}\ \langle$$
$$\mathit{Alarm} \Leftarrow \{\texttt{t}\triangleright 100\ \ \texttt{wakeup}\}\ \mathit{Alarm}$$
$$+\ \{\texttt{t}\triangleright(i \neq 100)\}\ \mathit{Alarm}$$
$$\rangle$$

Finally, using the second method, the sort of the alarm process is declared to include only a tick with a tag value of 100:

$$\textbf{part } \mathit{Alarm\_m2}\ \{\texttt{wakeup}, \texttt{t}_{100}\}\ \langle$$
$$\mathit{Alarm} \Leftarrow \{\texttt{t}\triangleright 100\ \ \texttt{wakeup}\}\ \mathit{Alarm}$$
$$\rangle$$

All the representations do nothing after the wake-up signal has been generated. Should further actions be required, *Alarm_m2* would have to be modified to receive all ticks with tags greater than 100.

A functioning system is obtained by composing *Alarm_xx* (where *xx* is replaced by *pt*, *m1*, or *m2*) with the virtual clock, and generating a `startclock` event. In all cases, the wakeup signal will be generated 100 ticks later.

$$\mathit{WakeUpAt100} \Leftarrow \{\texttt{startclock}\}\ \Delta_{\{\texttt{startclock}\}} \bullet \mathit{Alarm\_xx} \bullet \mathit{UniversalClock}$$

It is interesting to note that *UniversalClock* need not be defined to generate sequential integer tick tags. Any single valued function may be used in place of the successor function, including piecewise linear ones. The usefulness of this is not clear, but it might simplify the task of describing a system in which events occur in clusters separated by large intervals of time.

3.1.3 Multiple Clocks

As we have seen, controlling the occurrence of events by means of a Universal Clock is quite straightforward in CIRCAL. Digital designers make extensive use of clocks,[†] often of many levels and interrelationships, to synchronize operations in a circuit [Winkel 80]. To support this design philosophy, we too must be able go beyond the limitations of a single clock to model arbitrary clocking schemes.

Further consideration of the Universal Clock method raises the question of why must the clock be universal? If a system can be separated into subsystems with different clocks, then each one may be placed in its own "universe" with its own concept of time. The system's concept of time should then be derivable from those of its parts.

Example: Suppose we have a system clock, φ_1, that dictates the order of the events a, b, c. Furthermore, φ_1 occurs simultaneously with every second t tick generated by the Universal Clock[‡]. A typical sequence of actions would then look like:

$$P \Leftarrow \{\varphi_1\ \mathtt{a}\ \mathtt{t}\}\,\{\mathtt{t}\}\,\{\varphi_1\ \mathtt{b}\ \mathtt{t}\}\,\{\mathtt{t}\}\,\{\varphi_1\ \mathtt{c}\ \mathtt{t}\}\,\{\mathtt{t}\}\,Q$$

With the Universal Clock's presence implied by the t's.

Now we forget the period of φ_1 by hiding the ticks that were used to measure it. Here is what happens:

$$P - \mathtt{t} = \{\varphi_1\ \mathtt{a}\}\,\{\varphi_1\ \mathtt{b}\}\,\{\varphi_1\ \mathtt{c}\}\,(Q - \mathtt{t}) \tag{3.5}$$

The result is a system in which events are controlled purely by φ_1; in other words a new Universal Clock has been obtained that generates ticks labeled by φ_1. This shows that a system controlled by a hierarchy of clocks can be viewed at a higher level by simply abstracting away the lower level ticks.

[†]Real clocks, not the abstract Universal Clock that was introduced to enable measurement of the passage of time

[‡]We say that φ_1 has a period of 2 and a frequency of 1/2 $\frac{1}{\mathrm{tick}}$ (tocks?).

Proceeding in the opposite direction—top down—is not quite so easy. Given Equation 3.5 with $P - \mathtt{t}$ replaced by R and $Q - \mathtt{t}$ by S, how can the 2 t's per φ_1 relationship be recaptured? One way is to define a secondary process that embodies this relationship and compose it with R:

$$\begin{aligned} \Phi &\Leftarrow \{\varphi_1\ \mathtt{t}\}\,\{\mathtt{t}\}\,\Phi \\ R &\Leftarrow \{\varphi_1\ \mathtt{a}\}\,\{\varphi_1\ \mathtt{b}\}\,\{\varphi_1\ \mathtt{c}\}\,S \end{aligned} \tag{3.6}$$

$$R \bullet \Phi = \{\varphi_1\ \mathtt{a}\ \mathtt{t}\}\,\{\mathtt{t}\}\,\{\varphi_1\ \mathtt{b}\ \mathtt{t}\}\,\{\mathtt{t}\}\,\{\varphi_1\ \mathtt{c}\ \mathtt{t}\}\,\{\mathtt{t}\}\,(S \bullet \Phi)$$

By multiple applications of law [• +].

The Φ in Eqn. 3.6 is an example of what will be called a *clock process*, which implements a *clock relationship.*

Definition 3.1.2 Simple Clock Relationships and Processes
Given two tick actions t and u belonging to $\mathcal{L}$, we say that they form a *clock relationship* if n u events occur for every t event and t happens simultaneously with a u. The relationship is embodied by a *clock process* calculated by the following function of type $\mathcal{L} \times \mathcal{N} \times \mathcal{N} \times \mathcal{L} \longrightarrow \mathcal{P}$:

$$\mathrm{Clk}(\mathtt{t}, l, r, \mathtt{u}) \quad =_{def} \quad \mathrm{rec}\ X.(\{\mathtt{u}\}^l\,\{\mathtt{t}\ \mathtt{u}\}\,\{\mathtt{u}\}^{r-1}\,X)$$

Where $l + r = n$. □

It is sometimes desirable to start two parallel clock processes at different points in their cycles, hence the *phase* of the relationship can be changed by varying the two parameters l and r. The parameter l is the number of u ticks that must pass before the very first t tick when the system starts up. Thereafter, the period of the clock is the expected $r + l$. By way of example, here are two clock processes with the same period, but are they start up out of phase by two t ticks:

$$\begin{aligned} \mathit{CLOCK1} &= \mathrm{Clk}(\varphi_1, 0, 10, \mathtt{t}) \\ \mathit{CLOCK2} &= \mathrm{Clk}(\varphi_2, 2, 8, \mathtt{t}) \end{aligned}$$

The case when l is zero (the relationship is said to be *zero phase*) is so common that it is represented by a short form of the function *e.g.,*

$$\mathrm{Clk}(\mathtt{t}, 10, \mathtt{u})$$

If a zero phase relation has a period of one ($r = 1$) both t and u will occur simultaneously as can be seen from the definition:

$$\mathrm{Clk}(\mathtt{t},1,\mathtt{u}) \quad = \quad \mathrm{rec}\ X.(\{\mathtt{t}\ \mathtt{u}\}\, X)$$

In this case, the clocking process could be removed completely and all processes connected to the u tick renamed to use t instead.

Clock processes are used to relate the tick events that govern the passage of time in two different process universes. The two universes are composed with the clock process using the Dot Operator to produce a single universe whose time is measured by the smallest of the two granularities. If the clock process were not part of the composition, the two notions of time would interleave and generate an explosion of terms. One would expect this collapsing procedure to be transitive, as the following property shows.

Proposition 3.1.3 *Clock processes with zero phase difference are transitive when composed.*

$$\mathrm{Clk}(\mathtt{t},n,\mathtt{u}) \quad \bullet \quad \mathrm{Clk}(\mathtt{u},m,\mathtt{v}) \quad \text{implies} \quad \mathrm{Clk}(\mathtt{t},\, n\times m,\, \mathtt{v})$$

Proof: The above three relationships can be expressed as processes by applying the definition of Clk.

$$\begin{aligned} P &= \mathrm{rec}\ X.(\{\mathtt{t}\ \mathtt{u}\}\,\{\mathtt{u}\}^{n-1}\, X) \\ Q &= \mathrm{rec}\ Y.(\{\mathtt{u}\ \mathtt{v}\}\,\{\mathtt{v}\}^{m-1}\, Y) \\ R &= \mathrm{rec}\ Z.(\{\mathtt{t}\ \mathtt{v}\}\,\{\mathtt{v}\}^{nm-1}\, Z) \end{aligned}$$

Composing P and Q and abstracting away the u channel should result in R. Invoking laws [UNFOLD] and [$\bullet$ +] results in:

$$\begin{aligned} P \bullet Q &= (\{\mathtt{t}\ \mathtt{u}\}\,\{\mathtt{u}\}^{n-1}P) \ \bullet\ (\{\mathtt{u}\ \mathtt{v}\}\,\{\mathtt{v}\}^{m-1}Q) \\ &= \{\mathtt{t}\ \mathtt{u}\ \mathtt{v}\}\,\{\mathtt{v}\}^{m-1}(\{\mathtt{u}\ \mathtt{v}\}\,\{\mathtt{v}\}^{m-1})^{n-1}(P \bullet Q) \end{aligned}$$

Hiding u and applying a bit of elementary algebra reduces the expression considerably:

$$\begin{aligned} P \bullet Q - \mathrm{u} &= \{\mathtt{t}\ \mathtt{v}\}\,\{\mathtt{v}\}^{\,m-1}(\{\mathtt{v}\}\,\{\mathtt{v}\}^{\,m-1})^{n-1}(P \bullet Q - \mathrm{u}) \\ &= \{\mathtt{t}\ \mathtt{v}\}\,\{\mathtt{v}\}^{\,m-1}\,(\{\mathtt{v}\}^{\,m})^{n-1}\,(P \bullet Q - \mathrm{u}) \\ &= \{\mathtt{t}\ \mathtt{v}\}\,\{\mathtt{v}\}^{\,mn-m+m-1}(P \bullet Q - \mathrm{u}) \\ &= \{\mathtt{t}\ \mathtt{v}\}\,\{\mathtt{v}\}^{\,mn-1}(P \bullet Q - \mathrm{u}) \end{aligned}$$

Applying the law [FOLD] to the last equation produces:

$$\mathrm{rec}\ W.(\,\{\mathtt{t}\ \mathtt{v}\}\,\{\mathtt{v}\}^{nm-1}\,W\,)$$

This is identical to the equation for R, so by Fixpoint Induction we can conclude that they are equivalent. □

The clock processes discussed above were called *simple* for a good reason. The clocks were fixed in the way that they behave with respect to each other. More complicated relationships can be envisaged, including ones that deal with more than two tick labels and others that change through time. The former type, for example, can be used to model an implementation phenomenon called *clock skew.*

Clock skew refers to a variation in the arrival time (as measured by some global time standard) of clock events to different parts of a system. In VLSI circuits this is usually caused by high capacitance and/or long clock lines [Mead 80]. A simple form of this effect can be modelled by partioning the system into separate process-universes, each having a universal clock which is slightly phase shifted ("skewed") with respect to that of its neighbor. A global clock with a finer granularity is used to measure the amount of skew.

Example: Suppose that a circuit consists of two modules *INPUT* and *OUTPUT*, both controlled by a system clock φ with a period of 4 t. Unfortunately, when the modules were placed on silicon, an automatic router wired the clock generator circuitry to *INPUT* with a short metal line (low capacitance) and to *OUTPUT* with a very long diffusion wire (high capacitance and hence a delay). The delay associated with the long diffusion wire is 2 t.

The circuit can be broken down into two process-universes, one consisting of the clock generator and *INPUT* (clocked by the φ label), the other of *OUTPUT*

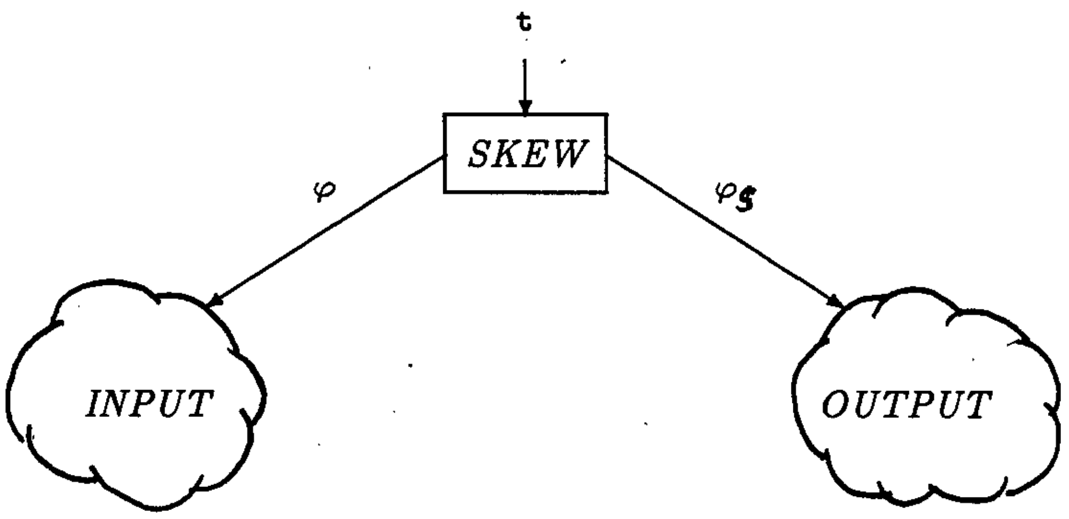


Figure 3–2: A model of clock skew .

(clocked by the φ_s label). Figure 3–2 shows how the pieces fit together, including a clock process *SKEW* that establishes a relationship between the two universes. The *SKEW* process is defined by a function similar to the Clk function:

$$\begin{aligned}\text{Skew}(\varphi_1, \varphi_2, \mathtt{t}, p, s) &=_{def}\\ &\text{rec } X.(\{\varphi_1\ \mathtt{t}\}\,\{\mathtt{t}\}^{(s-1)}\,\{\varphi_2\ \mathtt{t}\}\,\{\mathtt{t}\}^{(p-s-1)}\,X)\end{aligned}$$

The period of the clocks is p in units of t and the amount of skew is s. For this particular example, the clock process would be:

$$\text{Skew}(\varphi, \varphi_s, \mathtt{t}, 4, 2) \quad =_{def} \quad \text{rec } X.(\{\varphi\ \mathtt{t}\}\,\{\mathtt{t}\}\,\{\varphi_s\ \mathtt{t}\}\,\{\mathtt{t}\}\,X)$$

Although the skew is unrealistically large with respect to the clock period, it does serve to illustrate how the function operates. □

Example: Real world clocks can suffer from other problems besides skew. Two clocks that start in phase may slowly drift out of phase due to small variations in their timebases. This is known as ***clock drift*** and can be modelled by having a relationship that changes over time. Suppose that φ_1 and φ_2 start out perfectly in phase with a period of n ticks of the t universal clock:

$$R \;\Leftarrow\; \{\varphi_1\ \varphi_2\ \mathtt{t}\}\,\{\mathtt{t}\}^{n-1}\,R$$

Now for every $m \times n$ t ticks, the two drift apart by one tick. This is captured by adding a state variable to R:

$$\begin{aligned} R'(0) &\Leftarrow \left(\{\varphi_1\ \varphi_2\ \mathtt{t}\}\ \{\mathtt{t}\}^{n-1}\right)^m R'(n-1) \\ R'(x) &\Leftarrow \left(\{\varphi_1\ \mathtt{t}\}\ \{\mathtt{t}\}^{(n-1)-x}\ \{\varphi_2\ \mathtt{t}\}\ \{\mathtt{t}\}^{x-1}\right)^m R'(x-1) \end{aligned}$$

For the first $m \times n$ ticks, the process looks like R with the two clocks in perfect synchrony. On the very next tick, x gets set to $n-1$ which means that the clock relation looks like:

$$\{\varphi_1\ \mathtt{t}\}\ \{\varphi_2\ \mathtt{t}\}\ \{\mathtt{t}\}^{n-2}\ldots$$

After a further $m \times n$ ticks, φ_2 becomes two ticks out of phase with φ_1. This continues until eventually the clocks drift back into perfect phase. □

These sections have presented some examples of what must be termed *ideal* clocks. Time is sampled by infinitely narrow (Dirac delta function) events called ticks. Two events sharing a guard label-set with a tick label are considered to happen at exactly the same time even when the tick is hidden. This representation presents some pit-falls in digital design where time is sampled by ticks of a finite width, *i.e.*, step (Heaviside) functions.

To illustrate some of these problems, consider a specification that says: "In state *READ*, events **req** and **ack** occur during φ_1 before progressing to state *RELINQ*". At first glance, one would be tempted to describe this in CIRCAL by writing:

$$READ \Leftarrow \{\mathtt{req}\quad \mathtt{ack}\quad \varphi_1\}\ RELINQ$$

Unfortunately, this says that even if we were to move to a very fine granularity of time, **req** and **ack** would still occur simultaneously with the clock edge. What the specification really meant to say was: "..., events **req** and **ack** occur sometime, we don't care in what order, between the rising edge of φ_1 and the falling edge & φ_1 ...". Time, at the current level of abstraction, is being sampled by the finite width clock φ_1.

With the goal of coping with this type of statement, we now define the *temporal permutation* operators and use them in turn to define *clock intervals*.

3.1.4 Temporal Permutations

In order to make statements of the form: "we don't care in what order events b and c happen," we need a way of capturing this ambiguity in the calculus. Examining the statement closer, we can see that it means that one out of all possible orderings of b and c will be present, but which it is will be unknown. This immediately suggests an application of the nondeterministic choice operator:

$$\{c\}\,\{b\}\,(\ldots) \oplus \{b\}\,\{c\}\,(\ldots) \oplus \{b\ \ c\}\,(\ldots)$$

This expression is called a ***nondeterministic temporal permutation*** of b and c. The concept of permutations is so useful in writing specifications (*e.g.*, the informal example at the end of the last section) that a special operator will be defined to generate them.

Definition 3.1.4 The Nondeterministic Temporal Permutation Operator.

Let Γ be a set of label-sets and $\mathrm{pow}^-(\Gamma)$ be the powerset of Γ with the emptyset removed. Then the nondeterministic temporal permutation of Γ, written as $\textcircled{\pi}(\Gamma)P$, is defined by:

$$\begin{aligned} \textcircled{\pi}(\emptyset)\,P &=_{def} P \\ \textcircled{\pi}(\Gamma)\,P &=_{def} \sum_{\gamma_i \in \mathrm{pow}^-(\Gamma)} \cup\gamma_i\,[\textcircled{\pi}(\Gamma \setminus \gamma_i)\,P] \end{aligned}$$

Where $\Gamma \setminus \gamma_i$ removes the label-sets in γ_i from Γ. □

The union is required when γ_i is used as a guard since it is a set of label-sets whereas guards are just label-sets. Thus, $\cup\{\{a\}, \{a\ b\}\} = \{a\ b\}$.

The result of applying $\textcircled{\pi}$ to a set of composite labels (guards) is a tree-like structure of nondeterministic choices. The top level of the tree is a nondeterministic sum of all combinations of the guards in the set (similar to what is produced by the "Any Actions" operator). The next level below a particular guard is a similar tree formed from the set with those elements removed that formed the guard. Thus, as one progresses deeper in the tree, the number of branches will decrease until finally a single branch will lead to the resultant P. Here is a small example to consolidate this idea:

$$\textcircled{\pi}(\{\texttt{b}\}, \{\texttt{b c}\})\, P \quad = \quad (\{\texttt{b}\}\,\{\texttt{b c}\}\, P) \oplus (\{\texttt{b c}\}\,\{\texttt{b}\}\, P) \oplus (\{\texttt{b c}\}\, P)$$

Figure 3–3 shows part of the expansion of $\textcircled{\pi}(\{\texttt{a b}\}, \{\texttt{b c}\}, \{\texttt{c}\})\, P$ to illustrate the tree-like structure. The top level summation runs horizontally across the page, while the resultants of each branch run downward. A possible sequence can be examined by picking a top level guard, followed by one at the next level and so on down the page until a P is encountered.

```
        {a b}         ⊕{b c}⊕{c}⊕{a b c}⊕{a b c}⊕{b c}⊕{a b c}
{b c}⊕  {c}  ⊕{b c}     ⋮     ⋮     {c}       ⋮       ⋮      ⋮
 {c}    {b c}   P                   P
  P      P
```

Figure 3–3: Partial expansion of $\textcircled{\pi}(\{\texttt{a b}\}, \{\texttt{b c}\}, \{\texttt{c}\})\, P$

The nondeterministic temporal permutation operator as defined above produces sequences where the events may occur simultaneously. Sometimes it is more useful to produce sequences of purely interleaved events, without allowing simultaneity (*e.g.*, see the second example in Chapter 6). Another version of the $\textcircled{\pi}$ operator can be created to do this by a simple change to the above definition. It will be called the *distinct* permutation operator since it preserves the individual events.

Definition 3.1.5 The distinct nondeterministic permutation operator.

Let Γ be a set of label-sets. Then the distinct nondeterministic temporal permutation of Γ, written as $\textcircled{\pi}_D(\Gamma)P$, is defined by:

$$\begin{aligned} \textcircled{\pi}_D(\emptyset)\, P &=_{def} P \\ \textcircled{\pi}_D(\Gamma)\, P &=_{def} \sum_{\gamma_i \in \Gamma} \gamma_i[\textcircled{\pi}(\Gamma \setminus \gamma)\, P] \end{aligned}$$

□

It is possible, with some restrictions, to define deterministic analogues of the above operators. Simply changing the nondeterministic choice sums to deterministic sums is not good enough. Consider $\Gamma = \{\{\texttt{a b}\},\ \{\texttt{b c}\},\ \{\texttt{c d}\}\}$. Two subsets of $\mathrm{pow}^{-}(\Gamma)$ are $A = \{\{\texttt{a b}\}, \{\texttt{c d}\}\}$ and $B = \{\{\texttt{a b}\}, \{\texttt{b c}\}, \{\texttt{c d}\}\}$. But $\cup A = \cup B = \{\texttt{a b c d}\}$ which would produce a nondeterministic choice by law $[\gamma\ \oplus\ +]$ if the two were used as guards in a deterministic choice. To prevent this, it is necessary that Γ have the property:

$$\forall x, y \in \mathrm{pow}^{-}(\Gamma).\ \cup x \neq \cup y \tag{3.7}$$

Some legal sets are:

$\{\{\texttt{a b}\},\ \{\texttt{b c}\}\}$ $\{\{\texttt{in}\triangleright x\ \texttt{t}\}, \{\texttt{out}\triangleleft\texttt{l t}\}\}$ $\{\{\texttt{a}\},\ \{\texttt{b}\},\ \{\texttt{c}\},\ \{\texttt{d}\}\}$

Some illegal sets include:

$\{\{\texttt{a b c}\},\ \{\texttt{b c}\}\}$	($\{\texttt{a b c}\} \cup \{\texttt{b c}\}$ is the same as $\{\texttt{a b c}\}$)
$\{\{\texttt{d}\},\ \{\texttt{e}\},\ \{\texttt{d e}\}\}$	($\{\texttt{d}\} \cup \{\texttt{d e}\}$ is the same as $\{\texttt{d e}\}$, etc.)

Here then is the definition of the deterministic version of the permutation operator:

Definition 3.1.6 The Deterministic Temporal Permutation Operator.

Let Γ be a set of label-sets that obeys Equation 3.7 and $\mathrm{pow}^{-}(\Gamma)$ be the powerset of Γ with the emptyset removed. Then the deterministic temporal permutation of Γ, written as $\pi(\Gamma)P$, is defined by:

$$\begin{aligned} \pi(\emptyset)\,P &=_{def}\ P \\ \pi(\Gamma)\,P &=_{def} \sum_{\gamma_i \in \mathrm{pow}^{-}(\Gamma)} \cup\gamma_i.[\pi(\Gamma \setminus \gamma)\,P] \end{aligned}$$

□

A similar definition can be given for the distinct version of this operator.

The expansion for π has a tree-like structure similar to that of $\textcircled{\pi}$ with the nondeterministic choices replaced by deterministic ones. This leads to the following useful theorem:

Theorem 3.1.7 *The deterministic temporal permutation operator π is the identity for the nondeterministic version $\textcircled{$\pi$}$. In particular,*

$$\textcircled{π}(\Gamma)P \bullet \pi(\Gamma)P \quad = \quad \textcircled{π}(\Gamma)P$$

Where Γ satisfies Equation 3.7.

Proof: By induction on Γ. For the base case, choose $\Gamma = \emptyset$. Then, by the definition of $\textcircled{$\pi$}$, $\textcircled{$\pi$}(\emptyset)P = P$ and $\pi(\emptyset)P = P$. Thus the composition is $P \bullet P = P = \textcircled{π}(\emptyset)P$ as required.

For the induction step, let $Q \Leftarrow \textcircled{π}(\Gamma)P$ and $Q_i = \textcircled{$\pi$}(\Gamma \setminus \cup\gamma_i)P$, where $\gamma_i \in \text{pow}^-(\Gamma)$. Similarly, let $R \Leftarrow \pi(\Gamma)P$ and R_j be defined like Q_i. For convenience, also let $G = \text{pow}^-(\Gamma)$.

Now assume that $Q_i \bullet R_i = Q_i$, and show that $Q \bullet R = Q$. Expanding Q on the left hand side by one level produces:

$$Q \bullet R \quad = \quad \left(\underset{\gamma_i \in G}{\textcircled{Σ}} \cup\gamma_i\,(Q_i) \right) \bullet R$$

R is a deterministic sum, so it can be distributed over the nondeterministic sum using law $[\bullet\ \oplus]$ resulting in:

$$Q \bullet R \quad = \quad \underset{\gamma_i \in G}{\textcircled{Σ}} \left((\cup\gamma_i)\, Q_i \bullet \sum_{\lambda_j \in G} (\cup\lambda_j)\, R_j \right)$$

Both γ_i and λ_j are subsets of G. Moreover, because of the restriction of Equation 3.7, there is only one λ_j such that $\cup\gamma_i = \cup\lambda_j$. Since the sorts of Q_i and R_j are the same, law $[\bullet\ +]$ can be used to conclude that this λ_j will be the only label-set to synchronize with $\cup\gamma_i$, allowing the summation to be removed completely:

$$Q \bullet R \quad = \quad \underset{\gamma_i \in G}{\textcircled{Σ}} (\cup\gamma_i)\ (Q_i \bullet R_i) \quad = \quad \underset{\gamma_i \in G}{\textcircled{Σ}} (\cup\gamma_i)\ (Q_i)$$

by the Induction Hypothesis. The last equation is just the expansion of Q, so we can conclude that $Q \bullet R \;=\; Q$ as required. □

This theorem shows how the π operator can be used to describe a process that is able to accept an arbitrary sequence drawn from a given set of events. Such a process would typically receive communications from another process that outputs them in some unknown order, perhaps using the $\textcircled{\pi}$ operator.

Expanding permutation expressions so that they can be composed or otherwise manipulated soon results in a morass of symbols, because the number grows exponentially with the size of the generating set. Deriving algebraic properties of the operators is vital for their effective use. The following theorem presents one such property that will be needed in a later chapter.

Theorem 3.1.8 *The composition of two deterministic temporal permutation expressions is the permutation of the union of the two generating sets if each set is independent of the other process's sort and both resultants are guarded by the same label-set:*

$$\begin{aligned} P \textbf{ of sort } L &\Leftarrow \pi(\Gamma)\,\gamma\,P' \\ Q \textbf{ of sort } M &\Leftarrow \pi(\Lambda)\,\gamma\,Q' \\ P \bullet Q &= \pi(\Gamma \cup \Lambda)\,\gamma\,(P' \bullet Q') \end{aligned}$$

if

$$\Gamma \cap M \;=\; \Lambda \cap L \;=\; \emptyset$$

and $\Gamma \cup \Lambda$ *obeys the requirements of Equation 3.7.*

Proof: Sketch of an inductive argument. The base case of $\Gamma = \Lambda = \emptyset$ clearly holds by the definition of π, and $P \bullet Q = \gamma\,(P' \bullet Q')$. If both Γ and Λ have one element, applying law $[\bullet\ +]$ produces all permutations of those elements, yielding the same expression as is produced by $\pi(\Gamma \cup \Lambda)$:

$$\begin{aligned} P &\Leftarrow \alpha\,\gamma\,P' \\ Q &\Leftarrow \beta\,\gamma\,Q' \\ P \bullet Q &= \alpha\,\beta\,\gamma\,(P' \bullet Q') + \beta\,\alpha\,\gamma\,(P' \bullet Q') + (\beta \cup \alpha)\,\gamma\,(P' \bullet Q') \\ &= \pi(\alpha, \beta)\,\gamma\,(P' \bullet Q') \end{aligned}$$

This can be generalized by observing that adding another member to either Γ or Λ results in that label-set interleaving with the existing sequences by the application of law [$\bullet$ +]. All sequences must rendezvous at the γ guard, so neither P nor Q can recurse before the sequence is completed. This completes the induction step. □

Example: Given P of sort {`a, b, c`} and Q of sort {`c, d, e`} defined as:

$$\begin{aligned} P &\Leftarrow \pi(\{\mathtt{a}\},\{\mathtt{b}\})\,\{\mathtt{c}\}\,P \\ Q &\Leftarrow \pi(\{\mathtt{d}\},\{\mathtt{e}\})\,\{\mathtt{c}\}\,Q \end{aligned}$$

Then the composition is given by:

$$P \bullet Q \quad = \quad \pi(\{\mathtt{a}\},\{\mathtt{b}\},\{\mathtt{d}\},\{\mathtt{e}\})\,\{\mathtt{c}\}\,(P \bullet Q)$$

□

3.1.5 Clock Intervals

The motivation for defining the temporal permutation operators was the need for a way to represent events happening in any order during a particular period of time. Many digital systems are designed in a *register transfer* fashion wherein the outputs of blocks of combinational elements are saved in registers before being passed to the next block. The input values of the registers are transferred to the outputs by a global system clock, whose period is greater than the longest path delay through any of the blocks. In this way, each block will always have stable inputs and feedback loops may be constructed without fear of uncontrollable oscillation. Since the order in which input values to a register from a combinational block arrive is immaterial (all will be stable when the clock triggers the transfer), it makes sense to indicate this freedom in the specifications for the blocks. Many different implementations of a particular combinational function are possible, each of which will probably generate output values in a different order. Requiring a particular ordering in the specification prevents equally valid (and perhaps better) implementations from being considered.

To capture this requirement, we now define operators for producing *clock intervals.* Clock intervals allow one to indicate that a number of actions will occur in some unknown order between two delimiting events (*e.g.*, the rising and falling edges of a global system clock). Two varieties of operator will be defined. The first generates an *open* interval, meaning that the events that happen during the interval cannot do so simultaneously with the delimiting events. The second is similar, but produces a *closed* interval in which the internal events may overlap the delimiting ones. Open intervals are safer to use, since it is considered bad practice to allow digital values to change at the same time as a clock edge.

Definition 3.1.9 Open Clock Intervals

The open clock interval containing the set of label-sets Γ and delimited by events α and β that do not belong to any of the label-sets, is defined as:

$$\{\alpha\}\ \textcircled{π}(\Gamma)\ \{\beta\}\ P$$

and written as:

$$\mathrm{Int}(\,\alpha \mid \ \ \Gamma \ \ \mid \beta\,)\, P$$

□

The idea is that the beginning event of the interval, α, occurs followed by all possible sequences of the label-sets in Γ, then the closing event β.

Example: Suppose a block of combinational logic is fed values during phase one of the system clock, as indicated by the phi being high. The effects of the input changes percolate through the block until eventually they reach to output lines. These output lines will change value at different times according to the number of combinational elements between them and the inputs. As long as they all stabilize by the time phi goes low, however, we will not be interested in these delays. This type of behavior is modelled by an open clock interval:

$$\mathrm{Int}(\,\{\mathtt{phi}\triangleleft 1\} \mid \ \{\mathtt{out}_1\triangleleft x_1\}, \ldots, \{\mathtt{out}_n\triangleleft x_n\} \ \mid \{\mathtt{phi}\triangleleft 0\}\,) \ldots$$

Typically the $\mathtt{out}_i$ will be connected to a register that will accept the changes in any order, perhaps using the π operator:

$$REG \ \Leftarrow\ \pi(\{\mathtt{out}_1\triangleright\}, \ldots, \{\mathtt{out}_n\triangleright\}) \ldots$$

□

Defining a closed interval is trickier. Not only can the events occur in any order, but they can also occur simultaneously with the delimiting events. A modification of the definition of ⓟ is required.

Definition 3.1.10 Closed Clock Intervals.

Given the same notation as Definition 3.1.4 and letting the delimiting events be α and β, the closed clock interval containing Γ is defined as:

$$\begin{aligned}
F(\emptyset,\{\lambda\},\beta,P) &= (\lambda\cup\beta)P \oplus \lambda\beta\, P\\
F(\emptyset,\Gamma,\beta,P) &= \sum_{\gamma_i\in\mathrm{pow}^-(\Gamma)} \cup\gamma_i\, F(\emptyset,\Gamma\setminus\gamma_i,\beta,P)\\
F(\alpha,\Gamma,\beta,P) &= \alpha\, F(\emptyset,\Gamma,\beta,P) \oplus \sum_{\gamma_i\in\mathrm{pow}^-(\Gamma)} \alpha\cup(\cup\gamma_i)\, F(\emptyset,\Gamma\setminus\gamma_i,\beta,P)
\end{aligned}$$

The notation $\{\lambda\}$ stands for a set of label-sets containing one element λ. This is used to combine the last element in the sequence being generated with the β terminating event. Closed clock intervals will be written as:

$$\mathrm{Int}(\,\alpha,\ \ \Gamma,\ \ \beta\,)\,P$$

□

The vertical bars in the notation for open intervals are intended to show that the delimiting events are separated from the contents of the interval. Similarly, the comma's in the notation for closed intervals show that the events may intermingle.

Intervals may be cascaded so that the terminating event of one becomes the initial of the other. Here is how an open interval immediately followed by a closed interval would be written:

$$\mathrm{Int}(\,\{\mathtt{a}\}\mid\ \Gamma_1\ \mid \mathrm{Int}(\,\{\mathtt{b}\},\ \ \Gamma_2,\ \ \{\mathtt{c}\}\,)\,)$$

The first is delimited by a and b events, and the second by b and c.

3.2 Delay Elements

With the introduction of time, comes the concept of delay, and with delay comes timing problems. Circuits that are designed to work in an ideal world where input values arrive simultaneously often cease to do so when they are held up by differing lengths of time. Spurious and even completely wrong output values can result, which in turn can cause attached components to malfunction. Coping with timing problems is probably the single largest issue faced by the designer of high performance circuits. As we shall see in a later section, delays can introduce unsuspected and usually undesirable features into even the simplest behavior. Not all forms of delay, however, are harmful. Some hardware devices depend on delay for their very operation.

Delay manifests itself in many forms due to the variety of physical phenomena that cause it. We begin by discussing what are considered the most common forms that arise in digital systems. The discussion is followed by sample descriptions of the various delays in CIRCAL.

Digital systems map the continuous analog world of voltages and currents onto a discrete—usually binary—set of values. The primitive components that make up digital systems are designed to conduct current when a controlling voltage is greater than a certain threshold. Below this threshold they cease to conduct and appear as an open switch. Delays can then arise in two different ways.

The first, switching delay, is by far the most prevalent. It corresponds to the amount of time it takes the primitive element (transistor, relay, etc.) to change state from conducting to non-conducting or vice-versa. The output of a circuit might depend on a number of such switches and so will not present a valid result until all the devices have (in some sense) stabilized. The voltages controlling the switches take a finite amount of time to reach the *switching threshold* due to the resistances, inductances and capacitances present in the circuit. The time it takes for a controlling voltage to rise from zero to the threshold is called the *turn on time*. Similarly, the time it takes for the voltage to fall from a high value is called the *turn off time* and may or may not be the same as the turn on time.

The second type of delay, interconnection (propagation) delay, arises from the physics of charge flow. Electrons have a certain maximum speed (eight inches in one nano-second [Winkel 80]) that they can travel through even the best conductor. Signal propagation through non-ideal wires is modeled by *transmission line* theory and will not be considered here. Instead, propagation delays will be modeled by *pure* delays (see below). The pure delay effect of interconnection lengths tend to be far outweighed by the effects of switching delay. As feature sizes are being scaled down, however, the gap is closing [Mead 80].

3.2.1 Pure Delay

The simplest delay that one can imagine is to force an event to occur one time unit later than when it normally would. This is called *Unit Delay* and is modeled by a two-state part as follows:

part *UnitDelay* $\{\mathtt{a},\ \mathtt{b},\ \mathtt{t}\}$ $\langle$

$$UD \Leftarrow \mathrm{WAIT}_{\mathtt{t}}\big(\{\mathtt{a}\triangleright v \quad \mathtt{t}\}\, UD'(v)\big)$$
$$UD'(l) \Leftarrow \{\mathtt{t} \quad \mathtt{b}\triangleleft l\}\, UD + \{\mathtt{a}\triangleright v \quad \mathtt{b}\triangleleft l \quad \mathtt{t}\}\, UD'(v)$$

$\rangle$

This can be generalized to a D delay unit by composing D instances of *UnitDelay* and hiding the connecting lines:

$$\mathit{NDelay} \Leftarrow \underset{i=1}{\overset{D}{\boxed{C}}}\ \mathit{UnitDelay}(l_i)$$

$$= \Big(\mathit{UnitDelay}\,[\mathtt{b}_1/\mathtt{b}] \bullet \prod_{i=2}^{D} \mathit{UnitDelay}\,[\mathtt{b}_i/\mathtt{a} \quad \mathtt{b}_{i+1}/\mathtt{b}]\Big) - (\mathtt{b}_1,\ldots,\mathtt{b}_{D-1})$$

The Array Operator is used to cascaded D instances of a unit delay using the connection vector $C = [\mathtt{b}_i/\mathtt{a}_{i+1} \quad \mathtt{t}/\mathtt{t}_i]$. Breuer [Breuer 76] calls this *transport delay.*

Both *UnitDelay* and *NDelay* are forms of what Unger [Unger 69] calls *pure delay.* A pure delay is one that simply causes an event to occur some time after it was signalled. For time-varying signals and a delay of D, this means that $f(t)$ is transformed into $f(t-D)$.

The definition of *NDelay* given above is, to say the least, unwieldy! The longer the required delay, the more processes are needed to model it. An alternative formulation can be made that stores the values being delayed in a queue. The values are shifted left one place in the queue with every tick of the universal clock until they reach the head, from which they are output.

```
$NDelay(q)$  $\Leftarrow$  if $(\mathrm{hd}(q) \neq \perp)$ then
                   $\{\mathtt{b}\triangleleft \mathrm{hd}(q) \quad \mathtt{t}\}\ NDelay(\mathrm{shift}(q,\perp))$
                 $+ \{\mathtt{a}\triangleright x \quad \mathtt{b}\triangleleft \mathrm{hd}(q) \quad \mathtt{t}\}\ NDelay(\mathrm{shift}(q,x))$
               else
                   $\{\mathtt{t}\}\ NDelay(\mathrm{shift}(q,\perp))$
                 $+ \{\mathtt{a}\triangleright x \quad \mathtt{t}\}\ NDelay(\mathrm{shift}(q,x))$
               fi
```

Where hd(q) is the first element of list q.
$\perp$ is an undefined element not in the domain of x.
shift(q, v) removes the head of q and appends v to the tail.

Notice that the delay factor D is implicitly specified by the length of queue q. The domain of the elements of q is left unspecified, but must be the same as that of the input variable x. Every entry in the queue is assumed to be initialized to $\perp$.

Both the representation using a queue and the one made from cascaded unit delays may be used interchangeabley. One stores information in the state variables of several processes, whilst the other achieves the same effect by using a more complicated data structure. The choice of one representation over the other depends on how it is to manipulated. In an environment in which process creation is expensive, for instance, the second model might be preferable.

3.2.2 Inertial Delay

An *inertial* delay, as defined in [Unger 69], is one that responds to an input change only if it has persisted for a certain period D. If another input event

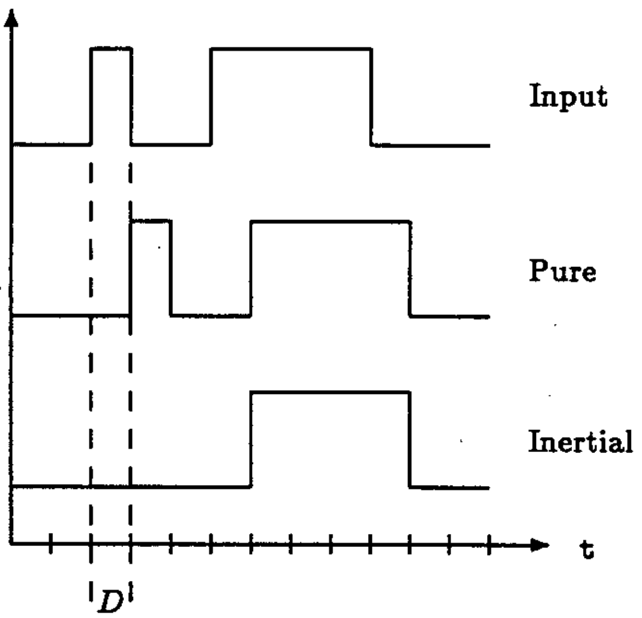


Figure 3–4: Comparison of delay types (From [Unger 69]). The delay in both cases is one t tick

occurs during D, the previous one is forgotten. Figure 3–4 shows how this type of delay ignores rapid input value changes (spikes). The CIRCAL description of an inertial delay is even simpler than that of a pure delay because the intermediate transitions do not have to be stored:

part *IDelay* $\{\texttt{a},\ \texttt{b}\}$ $\langle$

$$\begin{aligned} ID &\Leftarrow \text{WAIT}_{\texttt{t}}\big(\{\texttt{a}{\rhd}x\ \ \texttt{t}\}\ ID'(x,D)\big). \\ ID'(v,1) &\Leftarrow \{\texttt{b}{\lhd}v\ \ \texttt{t}\}\ ID + \{\texttt{a}{\rhd}x\ \ \texttt{b}{\lhd}v\ \ \texttt{t}\}\ ID'(x,D) \\ ID'(v,c) &\Leftarrow \{\texttt{t}\}\ ID'(v,c-1) + \{\texttt{a}{\rhd}x\ \ \texttt{t}\}\ ID'(x,D) \end{aligned}$$

$\rangle$

$D > 0$ is the required delay time in units of t. Many variations on this simple description are possible. An intermediate value, for example, can be output whilst the passage of time is being counted. In a three valued logic, this could correspond to the line taking on the "unknown" value.

Inertial delays find applications in protecting spike sensitive circuits from a potentially hostile environment and in modeling switching delay due to capacitive loading. A capacitor takes time to charge up to the switching threshold of a load device. Hence, the load will not switch if the charging voltage falls before the capacitor has charged up sufficiently.

3.2.3 Separate Delays

In real devices, logic values (voltages) often vary in the times it takes to change from one to another. NMOS is a common example of a technology that has widely differing times for a 0 to 1 transition as compared to a 1 to 0 transition. We can see how this happens by considering the NMOS inverter pictured in Figure 3–5.

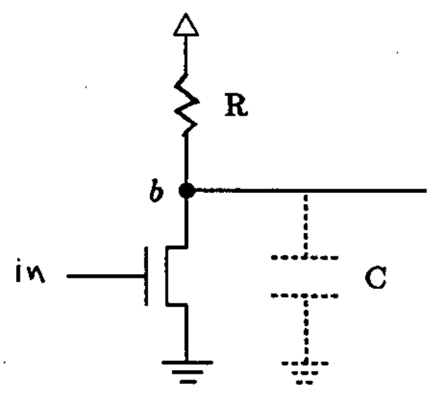


Figure 3–5: NMOS inverter

The capacitor C in the diagram is the combined capacitances of all the load devices. This can be done because MOS circuits have the nice feature that transistor gates form an almost ideal capacitive load [McCarthy 82]. Assume that the input in is at a logic High (Vdd) and that C is discharged. Node b is at ground and power is being dissipated by the pullup R and the on resistance R_{ON} of the transistor. Now in goes low (Gnd) turning off T. The transistor no longer pulls b to ground, so current will flow through R charging C up to Vdd. This take time proportional to the product of R and C (the *rise* time). If in subsequently goes high, turning T on again and pulling b low, the capacitor will discharge through R_{ON}. This resistance is typically much smaller than R (a factor of 4 is common) since R must be fairly large to reduce power dissipation when b is at ground. Hence, the time constant for C to discharge is a quarter of the charging constant, giving the output signal separate rise and fall times.

This behavior can easily be captured by a modification of the inertial delay model given above:

part *SepDel* {a, b} $\langle$

$$\begin{array}{rcl} SD & \Leftarrow & \text{WAIT}_{\texttt{t}}(\{\texttt{a}\triangleright x \;\; \texttt{t}\}\, SD'(x, D(x))) \\ SD'(v,1) & \Leftarrow & \{\texttt{b}\triangleleft v \;\; \texttt{t}\}\, SD \;+\; \{\texttt{a}\triangleright x \;\; \texttt{b}\triangleleft v \;\; \texttt{t}\}\, SD'(x, D(x)) \\ SD'(v,c) & \Leftarrow & \{\texttt{t}\}\, SD'(v, c-1) \;+\; \{\texttt{a}\triangleright x \;\; \texttt{t}\}\, SD'(x, D(x)) \end{array}$$

$\rangle$

Where $D(x)$ maps an input value onto a delay count, *e.g.*, $D(0) = 10$, $D(1) = 40$. The idea is that an input change triggers a value dependent count down until the change is reflected on the output. A simple approximation of the inverter discussed above would use $D(0) = 1$ and $D(1) = 4$ (and would invert the input value x) to show that the fall time is a quarter ofrise time.

This delay is inertial; new changes override propagating values. A pure delay version seems unreasonable since the separate delay times usually arise due to capacitive effects, which are inertial. More accurate approximations can be made by outputting intermediate values based on c, the counter that measures the passage of time, and/or by parameterizing the delay function D with this value.

3.2.4 Generic Boxes with Delay

The concept of generic box components was introduced in the last chapter. They made the generation of similarly structured components much easier by confining the functional personalization to a simple relabelling operation. It is often desirable to have versions of these components that incorporate one of the delay models just discussed. Of course, this can be done by composing a delay element onto the output of a box and hiding the connection, but these constructs are used so frequently that we will define a self-contained version.

Examining the descriptions of the delays given above, we notice that they have a structure very similar to that of the generic boxes. One part of the expression detects input changes and another performs the actual delay operation. By changing the input portion to that of an n-input one-output box, a new library component can be produced. For example, a generic two-input inertial delay box can be described as follows (remembering the requirement that only actual changes in value are output):

```
part $IBOXn\_1(\widetilde{in}, out)$ $\{\mathtt{in}_1, \ldots, \mathtt{in}_n, \mathtt{out}\}$ $\langle$
    $BID(\widetilde{in}, out)$  $\Leftarrow$  $\mathrm{WAIT}_{\mathtt{t}}(\mathrm{ANY}(\{\mathtt{in}_i\ \ \mathtt{t}\})$
                                   (if $f(\widetilde{in}) \neq out$ then $BID'(\widetilde{in}, f(\widetilde{in}), D))$
                                                  else $BID(\widetilde{in}, out))$
    $BID'(\widetilde{in}, out, 1)$  $\Leftarrow$  $\{\mathtt{out}\triangleleft\ \ \mathtt{t}\}\ BID(\widetilde{in}, out)$
                           $+\ \mathrm{ANY}(\{\mathtt{in}_i\ \ \mathtt{out}\triangleleft\ \ \mathtt{t}\})$
                                   (if $f(\widetilde{in}) \neq out$ then $BID'(\widetilde{in}, f(\widetilde{in}), D)\ )$
                                                  else $BID'(\widetilde{in}, out$ [illegible]$))$
    $BID'(\widetilde{in}, out, c)$  $\Leftarrow$  $\{\mathtt{t}\}\ BID'(\widetilde{in}, out, c-1)$
                           $+\ \mathrm{ANY}(\{\mathtt{in}_i\ \ \mathtt{t}\})$
                                   (if $f(\widetilde{in}) \neq out$ then $BID'(\widetilde{in}, f(\widetilde{in}), D)\ )$
                                                  else $BID'(\widetilde{in}, out, c-1))$
$\rangle$
```

The part waits for any changes on the $\mathtt{in}_i$ ports. When an input change happens, the delay count is started provided that the box's function returns a value different from that last output. When the delay count reaches one, this value is finally output. Input changes that happen during this period restart the count.

Figure 3–6(a) shows how generic two-input, one-output delay boxes will be pictured. Figure 3–6(b) shows an inertial type two-input AND gate with delay 10. Note that a particular instance requires three parameters to be specified *viz.*, the function to be calculated by the box, the delay count D and the type of delay (inte tial, pure, and so forth). The first two are defined by relabelling the appropriate parameters and the last by choosing a particular delay box.

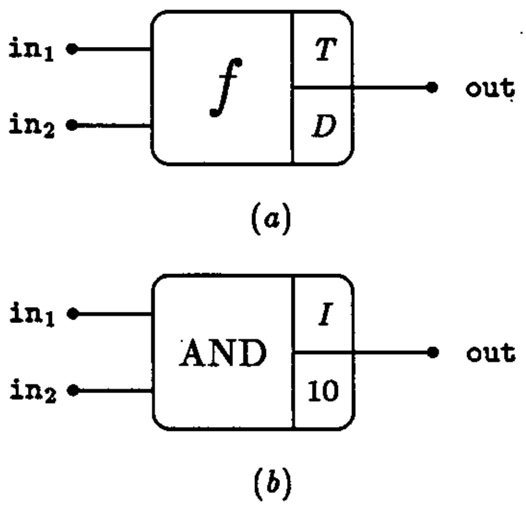


Figure 3–6: (*a*) Delay box. (*b*) Inertial delay AND gate with a delay of 10.

3.3 The Latch (SR Flipflop)

One of the simplest and most useful of the bistable feedback circuits is the Set-Reset Flipflop, also known as the latch. Its operation involves feedback, which opens the door to some interesting aspects of behavior. Because of this, it will be the favorite example circuit throughout the rest of the thesis, much as the ubiquitous shift register cell is used to demonstrate layout design systems.

As the name implies, the flipflop can be in one of two possible states, namely *Set* or *Reset*. The current state is reflected by the output q and its complement $\bar{q}$, called qbar here. A high logic value on q (and low on qbar) indicates the *Set* state, and vice-versa for the *Reset* state. A transition from *Reset* to *Set* is achieved by signalling an s event, called setting the latch. Similarly, an r event resets the state so that q is low and qbar is high. The s and r events are usually represented by low to high voltage transitions. This behavior is reflected in the following CIRCAL description:

part *LATCH* {s, r, q, qbar} (

$$\begin{aligned} \mathit{Reset} &\Leftarrow \{\mathtt{s}\triangleright 1\}\{\mathtt{qbar}\triangleleft 0\}\{\mathtt{q}\triangleleft 1\}\,\mathit{Set} + \{\mathtt{r}\triangleright x\}\,\mathit{Reset} + \{\mathtt{s}\triangleright 0\}\,\mathit{Reset} \\ \mathit{Set} &\Leftarrow \{\mathtt{r}\triangleright 1\}\{\mathtt{q}\triangleleft 0\}\{\mathtt{qbar}\triangleleft 1\}\,\mathit{Reset} + \{\mathtt{s}\triangleright x\}\,\mathit{Set} + \{\mathtt{r}\triangleright 0\}\,\mathit{Set} \end{aligned}$$

)

Note that the ordering of the output events was chosen to match that produced by the implementation considered below. We will return to this point in the next chapter.

One way to construct the latch is by cross-coupling two NOR gates. The gates are formed from generic delay boxes personalized with the NOR function and a pure delay of two units (Figure 3–7).

$$NA \Leftarrow \textit{PBOX2\_1}\ [\text{NOR}/f,\ 2/D,\ \texttt{r}/\texttt{in}_1,\ \texttt{qbar}/\texttt{in}_2,\ \texttt{q}/\texttt{out}]$$

$$NB \Leftarrow \textit{PBOX2\_1}\ [\text{NOR}/f,\ 2/D,\ \texttt{q}/\texttt{in}_1,\ \texttt{s}/\texttt{in}_2,\ \texttt{qbar}/\texttt{out}]$$

part *SRLatch* {s, r, q, qbar} ⟨

$NA(0,1,0,\texttt{q}(0,0)) \bullet NB(0,0,1,\texttt{q}(1,1))$

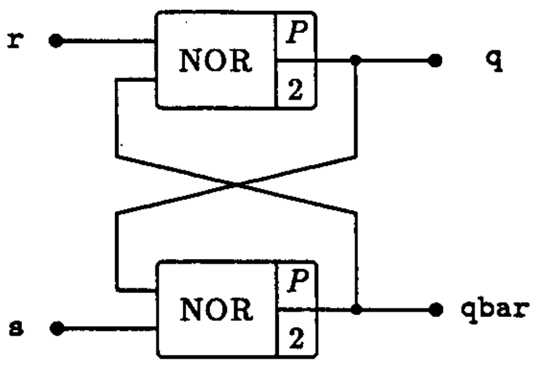
⟩

The *PBOX2_1* are generic boxes with two inputs ($\texttt{in}_1$ and $\texttt{in}_2$), one output (out) and a pure delay of length 2. They calculate the NOR function of any input changes and store this value in a delay queue. The queues of the two gates are initialized to contain all zeros and all ones respectively (the q(0,0) parameter to *NA* and an analogous one to *NB*). The queue is shifted from left to right with new entries appearing on the left. The arguments to the NOR gates are, from left to right, the previous value received on the $\texttt{in}_1$ line, previous value on the $\texttt{in}_2$ line, last value output (out line) and the current value of the delay queue. The part declaration thus describes a latch in the Reset state.

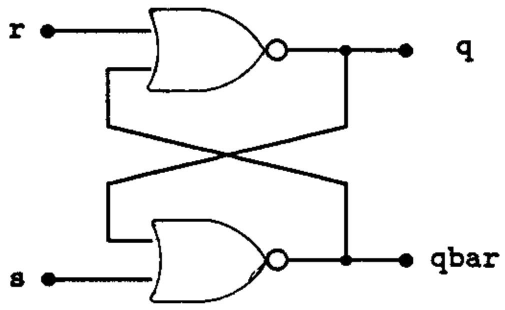
As a foretaste of the material presented in the next chapter, let us examine the operation of the constructed flip-flop. Setting the flip-flop is accomplished by causing the s line to go to 1; in other words, signalling s◁1 to *SRLatch*. By repeatedly applying law [• +] to rewrite the expression, the effects of this experiment will become visible as time ticks away.

$$\textit{TestInput} \Leftarrow \{\texttt{s}\lhd\texttt{1}\}\,\Delta_{\{\texttt{s},\,\texttt{r}\}}$$

$$\begin{aligned}\textit{TestInput} \bullet \textit{SRLatch} = &\\ &\{\texttt{t}\}\ (\textit{TestInput} \bullet NA(0,1,0,\texttt{q}(0,0)) \bullet NB(0,0,1,\texttt{q}(1,1)))\\ &+ \{\texttt{s}\lhd\texttt{1}\ \ \texttt{t}\}\ (\Delta_{\{\texttt{s},\,\texttt{r}\}} \bullet NA(0,1,0,\texttt{q}(0,0)) \bullet NB(0,1,0,\texttt{q}(0,1)))\end{aligned}$$



(a) Box representation.



(b) Equivalent circuit.

Figure 3–7: Constructed SR flip-flop

Expanding the second term six more times yields:

$$\{\mathtt{s} \triangleleft 1 \;\; \mathtt{t}\} \; (\Delta_{\{\mathtt{s},\,\mathtt{r}\}} \bullet NA(0,1,0,\mathtt{q}(0,0)) \bullet NB(0,1,1,\mathtt{q}(0,1))) =$$
$$\{\mathtt{s} \triangleleft 1 \;\; \mathtt{t}\}\{\mathtt{t}\}\{\mathtt{qbar} \triangleleft 0 \;\; \mathtt{t}\}\{\mathtt{t}\}\{\mathtt{q} \triangleleft 1 \;\; \mathtt{t}\}\{\mathtt{t}\}$$
$$(\Delta_{\{\mathtt{s},\,\mathtt{r}\}} \bullet NA(0,0,1,\mathtt{q}(1,1)) \bullet NB(1,1,0,\mathtt{q}(0,0)))$$

Note how the qbar line goes low two ticks before q goes high (Figure 3–8(a) shows these events plotted against occurrences of t). The two output lines are both low during this interval, yielding a transient illegal output value. What will happen if s subsequently goes low during this unstable period? To do this, we must make the *s* events occur at the same time as a tick. Only then can we make sure that they will occur at the desired time. Once the input stimuli have occurred, ticks must continue to be fed down the input lines until the effects have propagated through the circuit. With this in mind, let us redesign our experiment:

$$\mathit{Ticks} \text{ of sort } \{\texttt{s, r, t}\} \Leftarrow \{\texttt{t}\}\, \mathit{Ticks}$$

$$\mathit{TestInput} \text{ of sort } \{\texttt{s, r, t}\} \Leftarrow \{\texttt{s}\lhd\texttt{1 t}\}\{\texttt{s}\lhd\texttt{0 t}\}\, \mathit{Ticks}$$

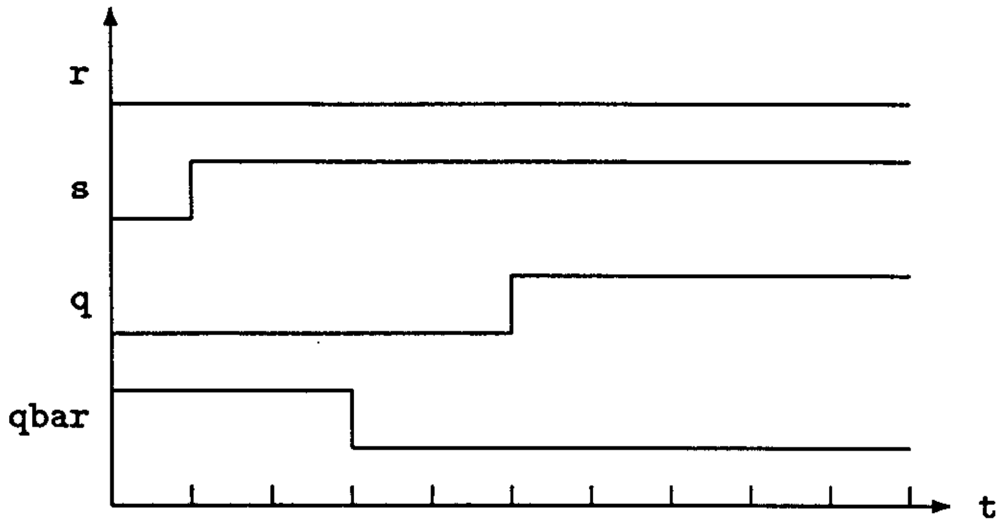
$$\mathit{TestInput} \bullet \mathit{SRLatch} =$$
$$\{\texttt{s}\lhd\texttt{1 t}\}\ (\{\texttt{s}\lhd\texttt{0 t}\}\ \mathit{Ticks} \bullet \mathit{NA}(0,1,0,\mathrm{q}(0,0)) \bullet \mathit{NB}(0,1,0,\mathrm{q}(0,1)))$$

Expanding further yields:

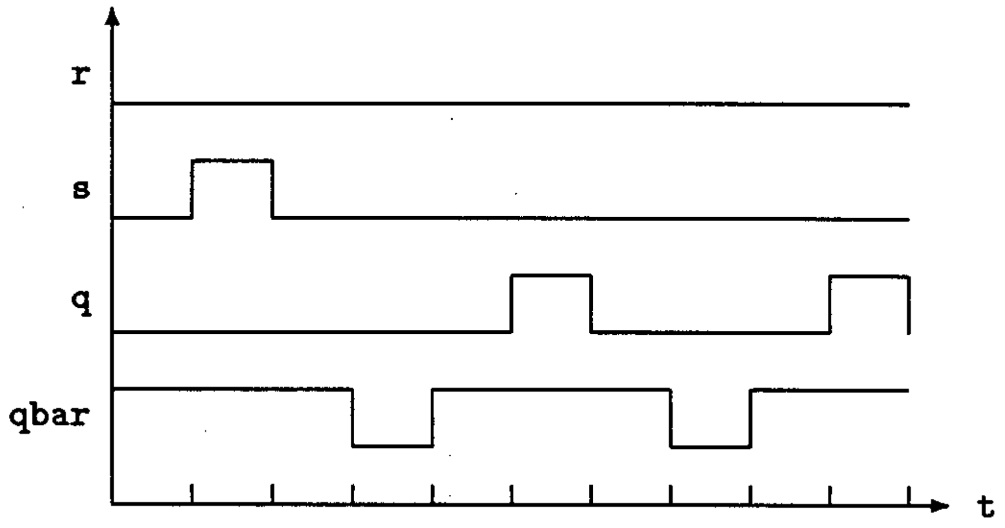
$$\begin{aligned}
&\{\texttt{s}\lhd\texttt{1 t}\}\{\texttt{s}\lhd\texttt{0 t}\}\\
&\quad\{\texttt{qbar}\lhd\texttt{0) t}\}\{\texttt{qbar}\lhd\texttt{1 t}\}\\
&\qquad\{\texttt{q}\lhd\texttt{1 t}\}\{\texttt{q}\lhd\texttt{0 t}\}\\
&\qquad\quad\{\texttt{qbar}\lhd\texttt{0 t}\}\{\texttt{qbar}\lhd\texttt{1 t}\}\\
&\qquad\qquad\{\texttt{q}\lhd\texttt{0 t}\}\{\texttt{q}\lhd\texttt{1 t}\}\\
&\qquad\qquad\quad(\mathit{Ticks} \bullet \mathit{NA}(0,1,0,\mathrm{q}(0,0)) \bullet \mathit{NB}(0,0,1,\mathrm{q}(1,0)))
\end{aligned}$$

The latch has entered an unstable state with the outputs oscillating (see the waveform in Figure 3–8(*b*)). The indication is that the circuit does not react well to input spikes and should be used in an environment in which these do not occur. This is an encouraging sign, since it shows that the model approaches real-world behavior. The outputs of an real latch would probably attain some unknown intermediate value depending on how the circuit is fabricated. The oscillation in our simulation is an artifact of the two-valued logic used here which is trying to take on this intermediate value. If we were to further model the NOR gates with a larger value set , the oscillation would be replaced by the unknown value.

This example shows how a description of a component in CIRCAL can be used to investigate its behavior. Additional methods for conducting the investigation will be encountered in the succeeding chapters.



a) Setting the latch.



(*b*) Spike on the set line.

Figure 3–8: Experimenting on a latch

3.4 Summary

A lot of ground has been covered in this chapter. We began by considering the mechanism due to Milne [Milne 83b] by which the notion of explicit time can be added to the calculus. This involved introducing a tick action into the guards of all processes. The tick action is produced by a Universal Clock process and is used to divide time into uniform fragments. Delays can be measured by counting the number of ticks that occur between events.

An alternative method for telling time was then presented that attached integer time stamps to the ticks. The integer values make it easy to wait for a particular time as, for example, might be required for an alarm clock function.

The tick method of representing time makes it easy to define hierarchies of clocks, each with a different granularity of time. This ability is of particular importance in digital systems where several notions of time can exist (*e.g.*, microcode cycles, register transfers, gate delays, etc.). For clarity, a notation was defined to represent these clock relationships in terms of clock processes.

When designing a system from the top down, knowledge about the characteristics of lower level components is rarely available. Thus, imposing an ordering on particular events in a specification at one level abstraction may not only be meaningless, but may also interfere with an equivalence proof conducted between the specification and a valid implementation at a lower level. Consequently, several operators (the temporal permutation and clock interval operators) were defined to capture this temporal ambiguity.

Finally, some concrete manifestations of time in the form of delay models were presented. The operation of some circuits is heavily influenced by delays in their components, as was demonstrated by constructing a latch from non-ideal NOR gates. Experimenting on the CIRCAL representation of this system with select stimuli revealed that it had the possibility of producing unexpected output values.

3.4.2 Contributions of this Chapter

New topics developed in this chapter included the notion of process universes and the clock relationships between them, tagged universal clock ticks, and the temporal permutation operators. The latter will prove to be highly useful in the following chapters and indeed were already used to define clock intervals in this one. The delays models presented here are neither complicated nor completely new, rather their representation in CIRCAL illustrates the formalism's ability to model a wide range of phenomena.

Chapter 4

Simulating Systems in CIRCAL

4.1 CIRCAL as an HDL

Chapter 2 introduced the CIRCAL calculus due to Milne and went on to present some new operators and syntax. In Chapter 3, we saw how the calculus can be used to represent various aspects of time and also saw some example descriptions of simple digital circuits. The emphasis all along has been on capturing the behavior of hardware and hardware related concepts. Little, however, has been said about what advantage is gained by applying this particular calculus to hardware description. Since the literature abounds with hardware description languages of every kind [Uehara 83, Breuer 75], there must be a compelling reason to introduce yet another into an already crowded field.

Hardware description languages (HDL's) are designed to capture either the physical structure of systems, or their behavior, or sometimes both. HDL's of the first type, such as CIF [Mead 80], SCALE [Marshall 83], and Sticks and Stones [Cardelli 81] are designed primarily for describing the layer masks that are used to fabricate integrated circuits. They provide a vehicle for representing the structure of a finished device with the ultimate purpose of driving fabrication equipment, and hence will not be considered here. CIRCAL is primarily a behavioral HDL, although there is a notion of structure imposed by the **part** construct and the static nature of process interconnections. This structure usually, but not always, has a physical interpretation.

Van Cleemput [VanCleemput 79] identifies four areas that behavioral HDL's can be applied to:

1. As a means of communicating a description of the behavior of a system between designers and clients.

2. Input to a simulator.

3. Input to a formal verification system.

4. Input to a hardware generator (compiler).

CIRCAL, perhaps embroidered with comments, is eminently suitable for the first application. Its well defined and compact semantics go a long way toward ruling out ambiguities. Moreover, new operators are easy to define and integrate with the existing ones. This is an important ability because, as we have seen in the previous chapters, describing the behavior of even simple systems can become complex very rapidly. Replacing common sequences of the basic operators by a new operator not only reduces the notational complexity, but more accurately reflects the designer's intentions.

Even with derived operators, constructing a description of a system in CIRCAL is a difficult task. Is the extra effort of creating the description worth it? The answer is yes, for that is the only way of exploring the functionality of a design short of fabricating it. Building a circuit, whether on silicon or on printed circuit boards, can involve considerable costs both in time and in resources. Engineers tend to iterate their designs, changing features as prototypes reveal problems or shortcomings. As designs become larger, each iteration becomes more expensive. No matter how carefully specified a system may be, performance difficulties and algorithmic problem areas will almost always surface. They must be detected before the design is built. Computer time is far cheaper than the costs associated with most forms of fabrication. It therefore makes sense to retain a design's representation in a "soft" form (*i.e.*, the description language) for as long as possible. Not only are changes easier to introduce, but simulation and verification tools may be applied to detect and eliminate design flaws. Another incentive is that the physical realization may be packaged in a form that does not permit access to the internal components, making testing difficult. Furthermore, a clear line may be draw between the flaws that are introduced by the fabrication process and those that are inherent in the design.

The goal of this chapter is to show how CIRCAL satisfies the second of Van Cleemput's applications—not just as input to a separate simulation program, but as a medium for conducting the simulation itself. We shall see that various simulation mechanisms can be obtained by exploiting properties of the operators of the calculus. A connection with the third application is also established, which will be followed up in a later chapter on verification. The fourth application, as input to a hardware generator, is not discussed in this thesis.

Throughout the chapter, reference will be made to various representation levels. These range from low (detailed information about primitive elements) to high (macroscopic properties of the complete system). The ranging can be continuous, but is usually considered to be divided into the following classes when discussing hardware [Coelho 84]:

- Circuit Level. Circuit elements (such as capacitors, resistors, transistors, etc.) are represented by detailed equations in terms of voltages and currents.
- Switch Level. Transistors are modelled as simple voltage controlled switches.
- Gate Level. Groups of transistors are replaced by the functional unit that they implement. Examples include NAND and NOR gates, counters, and most macrocells.
- Register Transfer Level. Behaviors are described as sequences of operations on data. Intermediate results are held in registers. Data and control busses are described at this level.
- Architectural (Block) Level. The overall structure of the system is represented and global conventions (such as I/O protocols) are specified.

These divisions are the most commonly accepted out of the many that are possible. They represent a convenient way of conveying the level of detail at which a particular system is being modelled.

4.2 The Rôle of Simulation and Verification

The terms *verification* and *simulation* have come to mean many different things. This section compares several of the common meanings and settles on the ones that will be used throughout the rest of the thesis. Loosely speaking, verification will refer to the process of showing that two representations of a system (a specification and an implementation) behave identically for all possible input sequences. The term simulation will be used to refer to the approach of applying a particular stimulus to a system and examining the resulting behavior for errors.

4.2.1 Verification

One way to obtain a high degree of confidence that a design will work properly when fabricated—modulo fabrication errors—is to check each step carefully as it is refined through successive representation levels. At each level, a *specification* (spec) is written detailing the intended behavior at that level. The blocks that form the next level down are then composed and shown to meet the requirements of the specification. The design is said to be *verified* when the implementation satisfies the specification for all possible input sequences. The term *verification* will be applied in this thesis only when the proof of the satisfaction relation is done with formal methods. Verification has a rather different meaning in industry. It is usually used to mean what we will call simulation, *i.e.*, the determination of correct behavior by the exhaustive application of test stimuli (called *test vectors* in hardware circles and *test suites* in software ones).

A point that must always be born in mind is that a circuit that has been shown to meet a spec is not guaranteed to work once it is physically realized, even with no fabrication errors. If a specification incorrectly reflects the required behavior, then so will the implementation. Similarly, an inaccurate model of the primitive components' behavior will yield verified designs that have no basis in reality. The degree of confidence inspired by a verification is in direct proportion to the accuracy of its input and of its underlying model. Verification, even with very simple models of device behavior, is still exceedingly valuable. Eliminating fifty percent of design errors can help immensely. If nothing else, the number of test stimuli that need be applied is drastically reduced.

Care must be taken in constructing the specification of each stage of the design. Too restrictive a specification and the constructed system may never meet it; too general and undesirable implementations may slip through. Trading detail versus ease of use and tractability is something of an art. It reflects the classic space-time tradeoff that is so often encountered in computer science.

Verification is particularly good for showing the functional correctness of an implementation in terms of smaller functional blocks. Most of the successful prototype hardware verification systems to date have been of this variety [Barrow 83, Gordon 83b, Shostak 83]. These systems have yielded some encouraging results, primarily because temporal issues have been dealt with crudely, or not at all. Similar automated theorem proving techniques have been successfully applied to proving simple software systems correct [Hailpern 82, Nelson 81, Anderson 77].

Detailed hardware specifications are hard to construct. Values are not discrete, but are continuous and usually time-varying in a complex manner. Individual components operate in a highly concurrent manner unless strict steps are taken to enforce sequentiality. Most importantly, hardware applications usually have far stronger *performance* requirements than do software ones. An implementation of an adder, for example, may not only have to add two numbers properly, but also to do so in 10 milliseconds and dissipate 10 microwatts of power.

Verification at this level can be extremely difficult and we shall not consider it further until a later chapter.

4.2.2 Compilation

An approach closely related to verification is that of *compilation* or *transformation*. The goal of the transformation process is to map a high level description of a design onto a hardware (or other low-level) realization. If the mapping can be proved correct (in some sense) for all possible input formulations, then much of the difficulty of designing a system would be eliminated. The question of whether or not the circuit will function properly then depends, as in verification, on how well the high level input description matches the designer's intentions.

Extensive efforts have been made by many groups [Deas 83, Bergmann 84, Ayres 83, Lattice 82] to develop so called "silicon compilers". These take a structural specification and turn it into fabrication masks for an integrated circuit chip. In the process they ensure that topological design rules are followed, loading constraints are satisfied (sometimes), and that gross "syntactic" errors have not been made (*e.g.*, connecting power to ground, leaving outputs dangling, attaching a 16-bit bus to an 8-bit register, and so forth). In these respects they have been quite successful, although the mappings that they use have not been proven correct in any formal sense. Some attempts, however, have been made in this direction that point the way for future research. In one paper [Milne 83c], Milne sketches a method (using CIRCAL) for proving a NOR–expression to geometry compiler correct. Sheeran, in her thesis [Sheeran 83], shows how the combinators of a functional language can be used to produce a floorplan corresponding to expressions in the language.

No further discussion of validating the compilation process will be made in this thesis. The approach taken is that a design will be verified and then, perhaps, given to a silicon compiler to implement.

4.2.3 Simulation

As we mentioned previously, the terms *simulation* and *verification* are often used interchangeably, mainly because the industrial state-of-the-art in determining the correctness of a design is through some form of simulation. We differentiate between the two terms as follows. Verification guarantees that a system will function exactly the same as a specification for all possible input stimuli. Simulation, on the other hand, shows that given a particular input (a test vector), the constructed system will respond with a certain output. This can be viewed as experimenting on a system and observing the results.

On the surface, verification seems far superior to simulation. For a simulation to reveal all design flaws, a large number of test vectors must usually be generated. Even then, there is no guarantee that the tests will be exhaustive (although there are methods for automatically generating tests for certain types of faults [Breuer 72]). There are, however, a couple of areas that are readily supported by simulation and not by verification. These are:

1. Pure verification does not support the experimental approach to design. Engineers like to "bread-board" ideas and apply stimuli to see how they function. In this manner they develop a feel for the requirements that must be included in a specification.

2. Even correctly designed systems will still malfunction due to fabrication errors. Test vectors designed to expose these potential errors should be exercised on a simulation of the system to determine their efficacy.

Point 1 concerns the manner in which engineers design systems. A complete system seldom springs fully specified from the void; it is built from a set of ideas that are iteratively refined until a satisfactory result is obtained. Often, the only way of telling if an iteration is a step forward is to see how it functions under select stimuli, in other words, by simulating it. Since there may not yet be a completely determined algorithm, much less a system specification, verification (in the total sense) is useless. Verification during this phase may find application locally in demonstrating properties such as termination and freedom from deadlocks in particular modules.

Even high level programming languages, such as Smalltalk [Goldberg 84] and LISP [Foderaro 80] have found it valuable to provide elaborate debuggers. A debugger is a form of interactive simulator that allows the user to monitor outputs and set inputs. No matter how good the compiler for a high level language is at checking the syntax and semantics of the program given to it, human mistakes will often get through. A constant may be defined incorrectly, a conditional expression may not be properly specified, or whatever. These will only get caught when the program is run and the results examined.

Studies in programming styles [Boehm *et al.* 84] reveal that specifying a system completely before implementing it does not necessarily produce a better product than the prototyping/iterative approach. Prototyping results in an end product that is easier to use and maintain, whilst specification generated products are more comprehensive and better documented. It is important that both styles be available to the designer so as to accommodate different design approaches. Simulation and experimentation are the primary tools of the prototyping approach.

The second point presented above is very important for hardware applications. Next to performance evaluation, fault modelling and analysis are the biggest applications for industrial simulators. Fabricating circuits, particularly on silicon, is an error-prone process. Wires may become shorted, transistors may be stuck open or closed, timing characteristics may vary across the chip, and so forth. Vectors must be generated for test equipment so that it can isolate faulty parts quickly and accurately. Introducing select faults into the simulation of a circuit allows the effectiveness of test vectors to be determined.

With these reasons as motivation, the following sections will examine how simulations can be conducted in the formal framework of CIRCAL. The important thing about remaining within the framework is that a direct link is then established with the verification techniques developed in Chapter 5. The ability to intermix the two approaches at will greatly facilitates the designer's job and should increase the likelihood that a system will work properly when first fabricated.

4.3 Simulation in CIRCAL

In Section 2.5, we saw how the behavior of a system can be represented by a synchronization tree in which arcs are labelled by events and nodes correspond to states in the computation. A path through the tree can be traced by choosing to traverse one of the branches available at each node. The path corresponds to a particular possible behavior of the complete system. When such a path can be determined by feeding a pattern to the system's inputs, we say that the system has been *simulated* for that *stimulus pattern.* In CIRCAL, stimulus patterns are just sequences of events and the process that generates them is attached to the test system using the Dot and relabelling operators (see Figure 4–1).

The simulation itself is conducted by simply expanding the composition of system and stimulus using the properties of the Dot operator. If the synchro-

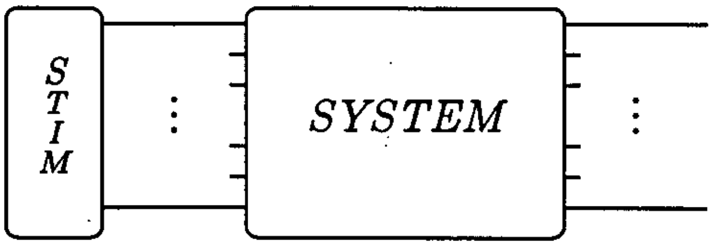


Figure 4–1: Applying a test pattern to a system

nization tree of the system contains only deterministic nodes, the expansion will consisted of repeated applications of law [• +]. The Dot Operator chooses the one branch at each deterministic choice node that matches an input stimulus (see Figure 4–2), and thus generates a sequence of events that reflect the system's response to those inputs. This works only if the input sequence has a sort that is a subset of the sort of the system's sort. In particular it should include all input channels, even if no activity will happen on some of them. If this were not done, some of the inputs could generate independent terms when the Dot Operator is expanded, resulting in a tree instead of sequence being output.

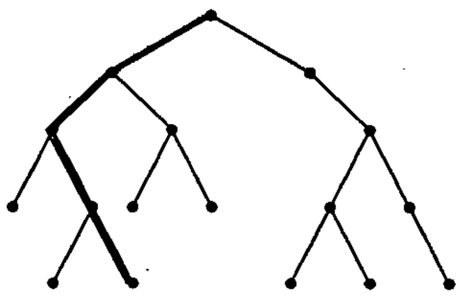


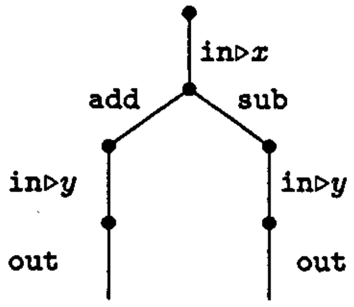
Figure 4–2: Choosing a path through the synchronization tree.

Non-deterministic nodes pose a problem because, by definition, not enough information exists to resolve them. One possible solution if this simulation technique were being mechanized would be to randomly choose a branch and warn the user to that effect. Another is to traverse each branch in turn, thus generating all possible output sequences. A third possibility is to relegate the responsibility for the choice to the user.

Example: Suppose a simple one digit calculator has an input channel that accepts a value in the range 0...9 (*i.e.*, of type **int1**) from some form of keypad decoder. There are also two operations available, addition as signalled by an add event and subtraction by sub. The result of a calculation is displayed on the one digit channel out. The behavior is given by:

$$\textbf{part}\ CALC\ \{\texttt{in, out:int1, add, sub}\}\ \langle$$
$$CALC \Leftarrow \{\texttt{in}\triangleright x\}\,(\{\texttt{add}\}\,\{\texttt{in}\triangleright y\}\,\{\texttt{out}\triangleleft((x+y)\bmod 10)\}\,CALC$$
$$+\ \{\texttt{sub}\}\,\{\texttt{in}\triangleright y\}\,\{\texttt{out}\triangleleft|x-y|\}\,CALC)$$
$$\rangle$$

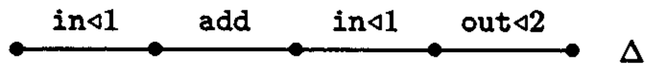
The calculator inputs a number with in▷x, inputs either an add or a sub operation, then another number with in▷y, and finally outputs the result on out. A fragment of the communication tree looks like this:



Now suppose we wish to test if the calculator adds one and one correctly. We therefore define a stimulus process that signals an in◁1 event followed by an add and finally an in◁1 again:

$$ST\ \textbf{of sort}\ \{\texttt{in, add, sub}\} \Leftarrow \{\texttt{in}\triangleleft 1\}\,\{\texttt{add}\}\,\{\texttt{in}\triangleleft 1\}\,\Delta_{\{\texttt{in, add, sub}\}}$$

Composing *CALC* with *ST* and repeatedly expanding produces the following output sequence:



Precisely what was expected! The Dot Operator selected only those paths that synchronized with the stimulus. The out◁2 event was not found in the sort of *ST* and so was allowed to occur independently.

This approach to simulation is quite elegant. Both the behavior of the system and the stimuli that are used to experiment on it are expressed in the same language. Moreover, the simulation itself is conducted by simply applying the expansion rule for the Dot Operator, obviating the need for a separate simulation mechanism.

The next sections show how to construct more elaborate testing schemes that bridge the gap between pure simulation and pure verification.

4.3.1 Symbolic Simulation

Pure simulation involves applying particular data values to the inputs of a circuit. The results of the simulation then hold for those and only those values. For many applications, the number of possible input combinations is very large (there are 2^{64} possible permutations of two 32 bit integers alone) making exhaustive testing very expensive. Frequently, many of these tests are redundant because they reveal no new information about the design. Simulating an implementation of a function that adds two 32 bit integers, for example, is counterproductive if all that results is a truth-table with 2^{64} entries. Who would have the patience to analyze this output?

An alternative approach, called *symbolic simulation*, applies stimuli containing variables rather than constant values to the inputs of the system. The variables pass through the sub-units of the system to produce a composite function on the input and state variables as the output. In the adder example, two variables x and y can applied to the inputs in lieu of 32 bit constants, resulting in a function on these variables that could be shown equal to the addition function using elementary algebra (if it works properly). The full complexity of comparing the system's temporal as well as functional behavior with a specification has been avoided by removing the temporal aspect. This is a first step in the road to proving that the system is equivalent to its specification. Simpler proof techniques can be used than if the full complexity of the synchronization tree were being examined.

Conducting symbolic simulations in CIRCAL is almost as easy as doing pure ones. Recall that value passing along a communication channel was defined in terms of a possibly infinite choice summation of labels (Section 2.2.8). A pure simulation, because it always applies a constant value, selects only one of these branches to produce a single sequence of events. Symbolic simulation, on the other hand, can be done by simply allowing certain trees as a valid result instead of just sequences. These trees are the summations corresponding to the outputting of a function on a channel.

To see how this form of simulation works, let us apply it to a simple circuit consisting of two cascaded adder elements. The adders are defined by personalizing the generic box component defined in Section 2.8.2 with the addition function. The domain of the input ports will be the natural numbers and the adders will be initialized to have previously input and output zeros. Figure 4–3 shows the structure.

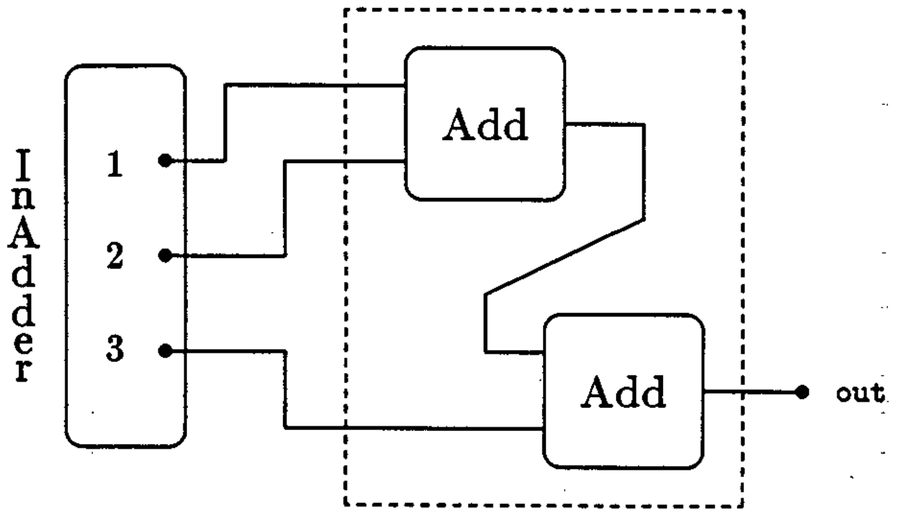


Figure 4–3: Cascaded adders

Here is the CIRCAL description of the system, including a sample symbolic test stimulus:

$$\begin{aligned}
\textit{Adder1} &\Leftarrow \textit{BOX2\_1}\ [+/f,\ \texttt{i1}/\texttt{in}_1,\ \texttt{i2}/\texttt{in}_2,\ \texttt{c}/\texttt{out}] \\
\textit{Adder2} &\Leftarrow \textit{BOX2\_1}\ [+/f,\ \texttt{i3}/\texttt{in}_1,\ \texttt{c}/\texttt{in}_2] \\
\textit{System} &\Leftarrow (\textit{Adder1}(0,0,0) \bullet \textit{Adder2}(0,0,0)) - \texttt{c} \\
\textit{InAdder} &\Leftarrow \{\texttt{i1}\triangleleft x\ \ \texttt{i2}\triangleleft 2\ \ \texttt{i3}\triangleleft 3\}\,\Delta_{\{L\}} \\
L &= \{\texttt{i1},\ \texttt{i2},\ \texttt{i3}\}
\end{aligned}$$

The channel i1 will have a variable applied to it, while the other two will have constants. The resulting output on channel out should be a simple function of the single input variable since there is no internal state. By repeated applications of laws [$\bullet$ +] and [$-$ +], the following output is generated:

$$\begin{aligned}
\textit{InAdder} \bullet \textit{System} &= \{\texttt{i1}\triangleleft x\ \ \texttt{i2}\triangleleft 2\ \ \texttt{i3}\triangleleft 3\}\,\{\texttt{out}\triangleleft(0+3)\}\,\{\texttt{out}\triangleleft((x+2)+3)\} \\
&\qquad \Delta_{\{L\}} \bullet \Big((\textit{Adder1}\,(x,\,2,\,(x+2)) \\
&\qquad\quad \bullet\ \textit{Adder2}\,((x+2),\,3,\,((x+2)+3))) - \texttt{c}\Big) \\
&= \{\texttt{i}\triangleleft x\ \ \texttt{i2}\triangleleft 2\ \ \texttt{i3}\triangleleft 3\}\,\{\texttt{out}\triangleleft 3\}\,\{\texttt{out}\triangleleft(x+5)\} \\
&\qquad \Delta_{\{L\}} \bullet \Big((\textit{Adder1}\,(x,\,2,\,x+2) \\
&\qquad\quad \bullet\ \textit{Adder2}\,(x+2,\,3\,,x+5)) - \texttt{c}\Big)
\end{aligned}$$

We have shown that for all values of x, when i2 is 2 and i3 is 3, the final output of the circuit will be $x+5$. The laws of addition were used to collapse the term $((x+2)+3)$ down to $x+5$. The simulation also showed that the circuit will produce a transient output value due to the delay through the c channel.

The generality of the result can be extended by applying variables to all the input ports. This moves the problem of showing that the circuit does add three numbers from the CIRCAL domain to that of normal arithmetic (the final output value will be $(x+y)+z$). In this case no further simplification is necessary, but in general a simple theorem prover might be able handle the resulting expressions, whereas it would have very little hope of conducting a successful proof in CIRCAL. All the other input possibilities would have to be considered as well as just the case when all three change simultaneously. With symbolic simulation, human intuition is used to select significant input sequences.

4.3.2 Constructive Simulation

In Chapter 2, the case was made for designing systems in a hierarchical fashion. The simulations demonstrated above flatten this hierarchy in order to produce the synchronization tree of the system under test and then go on to prune this tree. Such an approach not only renders useless the effort that went into constructing the hierarchy, but also means that a tree must be constructed most of which will be thrown away. Any implementation of CIRCAL style simulation would therefore require a large amount of temporary storage to hold these trees. Much of this wastage could be eliminated by pruning the tree as it was being built from the hierarchy. Milne [Milne 84a] calls this ***constructive simulation*** because the results are constructed in precisely the same manner as the system. Figure 4-4 shows how the trees of two constituent parts of a system are pruned and composed to produce a simulation of the complete system.

$$S_1 \bullet T_1 \quad \bullet \quad S_2 \bullet T_2 \quad = \quad S \bullet T$$

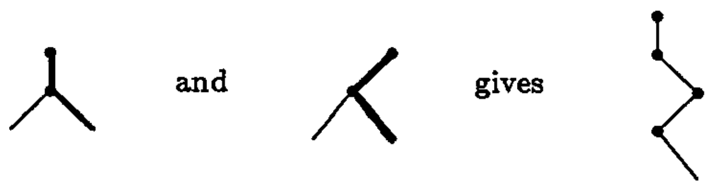


Figure 4–1: Constructing a simulation from locally pruned synchronization trees

The pruning of synchronization trees to produce a simulation is done in CIRCAL by the composing a test stimulus process with the tree. Normally this is done for the synchronization tree that describes a complete system, so how can it be applied to the constituent processes? Two methods are considered here that have *local* and *global* effects respectively.

Local Constructive Simulations

Since test processes are described in the same calculus as the object under test, they too can be constructed in a hierarchical fashion. In this way, the parts of a test stimulus that effect a particular sub-unit of the system are grouped with that sub-unit. Each such splinter of the original stimulus prunes the synchronization of the sub-unit that it is grouped with before the sub-unit is composed with its neighbors. This technique is called *local constructive simulation* since the test patterns are designed with a localized effect in mind. It can be demonstrated by considering a two component system $S \Leftarrow S_1 \bullet S_2$ which is being tested by the test process $T \Leftarrow T_1 \bullet T_2$:

$$\begin{aligned} T \bullet S &= (T_1 \bullet T_2) \bullet (S_1 \bullet S_2) \\ &= (T_1 \bullet S_1) \bullet (T_2 \bullet S_2) \qquad [\bullet_C], [\bullet \bullet] \end{aligned}$$

The sort of T_i must be a subset of the sort of S_i for them to have the desired effect.

Local constructive simulations are very useful when a particular unit of a system is to be simulated at a different level of abstraction from the others. Because the portion of the test stimulus that effects the unit is attached locally, it far easier to change the nature of the stimulus than if it were part of a larger pattern. For example, suppose that a system consisting of five units with byte-wide inputs has been successfully simulated at the block level. Then the designer becomes curious about the behavior of the first unit under the same test patterns when it is considered at the gate level. Accordingly the unit is replaced by its gate level description. At this level the byte-wide input channel appears as eight one-bit channels, necessitating a change in the test patterns. If a local constructive simulation approach were in use, this change would be easily accomplished. Under a normal approach, on the other hand, all references to this channel must be tracked down amongst the stimuli for the other channels.

The ability to simulate a system at mixed levels of abstraction can save tremendously on the execution time of a simulation. Complexity and hence execution time tend to be inversely proportional to the level of abstraction at which a system is represented. This encourages the inclusion of a minimum of low level units during a single simulation run. Using local constructive simulation, the abstraction level of a unit and its test pattern can be easily changed, thus making the use of mixed levels easier. An approach similar to this, called "multi-level simulation", is discussed by Coelho [Coelho 84] in the context of a commercial simulator.

Example: To demonstrate the local constructive simulation approach, we will now apply it to the 3-input adder example used above. Instead of defining a single test agent, *InAdder*, that communicates with all the input ports, we define one such agent for each of the constituent boxes that have inputs (Figure 4–5). As the system is built up by composition of these boxes, the test agents join up to form a single test process. If the simulation were being run on a multiprocessor machine (*e.g.*, a tree machine), each test agent would perform its own local simulation and then pass the results to neighboring processors.

$$\begin{array}{lcl}
\mathit{A1} & \Leftarrow & \mathit{InAdder1} \bullet \mathit{Adder1} \\
\mathit{A2} & \Leftarrow & \mathit{InAdder2} \bullet \mathit{Adder2} \\
\mathit{InAdder1} & \Leftarrow & \{\mathtt{i1}\triangleleft 1 \;\; \mathtt{i2}\triangleleft 2\}\,\Delta_{\mathtt{i1},\,\mathtt{i2}} \\
\mathit{InAdder2} & \Leftarrow & \{\mathtt{i3}\triangleleft 3\}\,\Delta_{\mathtt{i3}} \\
\mathit{System} & \Leftarrow & (\mathit{A1}(0,0,0) \bullet \mathit{A2}(0,0,0)) - c
\end{array}$$

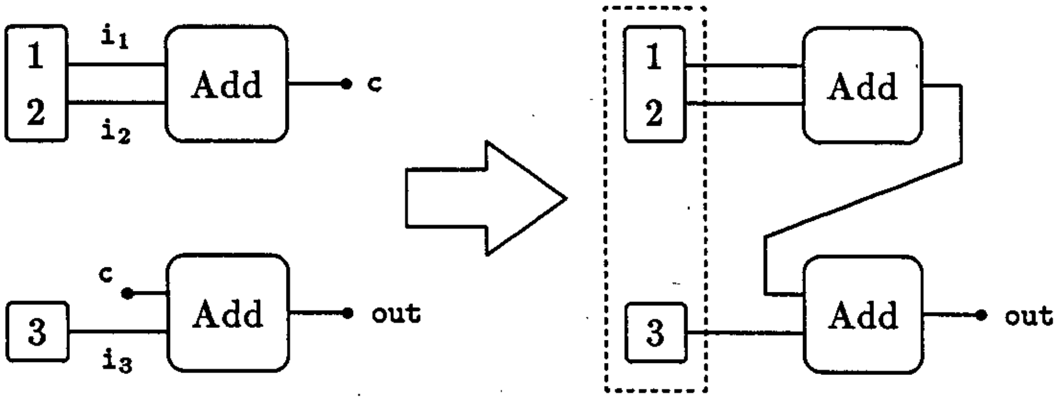


Figure 4–5: Local constructive simulation of a 3-input adder

As before, the simulation is conducted by applying law [• +] multiple times:

$$\begin{aligned}
\mathit{System} \;=\; & \{\mathtt{i3}\triangleleft 3\}\Big((\{\mathtt{out}\triangleleft 3\}\,\mathit{Adder2}(0,3,3) \;\bullet\; \Delta_{\{\mathtt{i3}\}} \\
& \qquad \bullet\; \mathit{InAdder1} \;\bullet\; \mathit{Adder1}(0,0,0)) - \mathtt{c}\Big) \\
& + \{\mathtt{i1}\triangleleft 1 \;\; \mathtt{i2}\triangleleft 2\}\Big((\mathit{Adder1}(1,2,3) \;\bullet\; \Delta_{\{\mathtt{i1},\,\mathtt{i2}\}} \\
& \qquad \bullet\; \mathit{InAdder2} \;\bullet\; \mathit{Adder2}(0,0,0)) - \mathtt{c}\Big) \\
& + \{\mathtt{i1}\triangleleft 1 \;\; \mathtt{i2}\triangleleft 2 \;\; \mathtt{i3}\triangleleft 3\}\Big((\{\mathtt{c}\triangleleft 3\}\,\mathit{Adder1}(1,2,3) \;\bullet\; \Delta_{\{\mathtt{i1},\,\mathtt{i2}\}} \\
& \qquad \bullet\; \Delta_{\{\mathtt{i3}\}} \;\bullet\; \{\mathtt{out}\triangleleft 3\}\,\mathit{Adder2}(0,3,3)) - \mathtt{c}\Big)
\end{aligned}$$

Expanding the third term twice more yields:

$$\{\mathtt{i1}\triangleleft 1\ \ \mathtt{i2}\triangleleft 2\ \ \mathtt{i3}\triangleleft 3\}\,\{\mathtt{out}\triangleleft 3\}\,\{\mathtt{out}\triangleleft 6\}$$
$$((\Delta_{\{\mathtt{i1},\,\mathtt{i2},\,\mathtt{i3}\}} \bullet \mathit{Adder1}(1,2,3) \bullet \mathit{Adder2}(3,3,6)) - \mathtt{c})$$

This shows that the circuit does indeed add 1,2, and 3 to eventually produce 6. As before, a spurious transient output generated before the correct one.

The problem with the above expansion was that we had to realize that expanding the third summand would give us the correct result. When the same simulation was conducted earlier without using the constructive approach, the other summands did not appear. Where have they come from? The answer can be found in the way we defined the two test agents. Each has a sort independent of (disjoint from) the other since they test separate parts of the system. Thus, when they are combined with the Dot Operator, all possible interleavings result:

$$\begin{aligned}\mathit{InAdder1} \bullet \mathit{InAdder2} &= \{\mathtt{i1}\triangleleft 1\ \ \mathtt{i2}\triangleleft 2\}\,(\Delta_{\{\mathtt{i1},\,\mathtt{i2}\}} \bullet \mathit{InAdder2})\\ &\quad + \{\mathtt{i3}\triangleleft 3\}\,(\Delta_{\{\mathtt{i3}\}} \bullet \mathit{InAdder1})\\ &\quad + \{\mathtt{i1}\triangleleft 1\ \ \mathtt{i2}\triangleleft 2\ \ \mathtt{i3}\triangleleft 3\}\,\Delta_{\{\mathtt{i1},\,\mathtt{i2},\,\mathtt{i3}\}}\\ &\neq \{\mathtt{i1}\triangleleft 1\ \ \mathtt{i2}\triangleleft 2\ \ \mathtt{i3}\triangleleft 3\}\,\Delta_{\{\mathtt{i1},\,\mathtt{i2},\,\mathtt{i3}\}}\end{aligned}$$

Since the problem stems from the fact that the individual test agents have disjoint sorts, it is possible to construct an *ad hoc* solution to the problem. If a synchronizing label is introduced into each communication, exactly as was done in Chapter 3 to add absolute time to a system, the interleavings will disappear. This makes sense when we consider that the individual test processes are the decomposition of a single pattern which is applied to all the inputs simultaneously. Each process must wait for the others to finish calculating their portions of the current stimulus before proceding to the next in the test sequence.

Here is an example of how the test processes from the last example can be used to apply a sequence of stimuli to the adders:

$$\begin{aligned}\mathit{InAdder1} &\Leftarrow \{\mathtt{i1}\triangleleft 1\ \ \mathtt{i2}\triangleleft 2\ \ \mathtt{sync}\}\,\{\mathtt{i1}\triangleleft 47\ \ \mathtt{sync}\}\,\{\mathtt{sync}\}\,\Delta_{\{\mathtt{i1},\,\mathtt{i2},\,\mathtt{sync}\}}\\ \mathit{InAdder2} &\Leftarrow \{\mathtt{i3}\triangleleft 3\ \ \mathtt{sync}\}\,\{\mathtt{sync}\}\,\{\mathtt{i3}\triangleleft 17\ \ \mathtt{sync}\}\,\Delta_{\{\mathtt{i1},\,\mathtt{sync}\}}\end{aligned}$$

$$\begin{aligned} \textit{TestSeq} &\Leftarrow \textit{InAdder1} \bullet \textit{InAdder2} \\ &= \{\mathtt{i1}\triangleleft 1 \;\; \mathtt{i2}\triangleleft 2 \;\; \mathtt{i3}\triangleleft 3 \;\; \mathtt{sync}\} \\ &\qquad \{\mathtt{i3}\triangleleft 3 \;\; \mathtt{sync}\} \\ &\qquad\qquad \{\mathtt{i1}\triangleleft 47 \;\; \mathtt{sync}\}\, \Delta_{\{\mathtt{i1},\,\mathtt{i2},\,\mathtt{i3},\,\mathtt{sync}\}} \end{aligned}$$

The synchronization signals have eliminated the interleaving and permited a sequence to be defined in which the values of 1, 2 and 3 are applied to i1, i2 and i3 respectively, followed by a 3 to i3 and then a 47 to i1.

One problem with using local constructive simulation during the design of a system is that the test agents are interspersed with the constituent parts. Removing these agents at a later date if use of the complete behavior is necessary might be difficult. This would be readily accomplished if somehow the names of the test processes could be left syntactically grouped with their target parts, but defined in such a way as to be unable to interact with them. The null-sorted deadlock process $\Delta_{\{\emptyset\}}$ has precisely this inablity to interact any other process, so it can be used to "deactivate" local test agents. The identifier that names the agent is simply defined to be $\Delta_{\{\emptyset\}}$.

Global Constructive Simulation

When constructive simulation was first introduced, it was mentioned that there were two ways of approaching it. As we have just seen, one way is to decompose test patterns into stimuli that locally prune a synchronization tree as it is constructed. Another way is to leave the test pattern intact, but still have it operate locally. This is called *global constructive simulation* because the test pattern is shared by the test systems's constituent parts, and is accomplished by exploiting the *idempotency* property of the Dot Operator.

Idempotency of the Dot Operator—Law $[\bullet_1]$ in Section 2.2.11—simply means that two copies of a deterministic process (see Definition 2.5.1) running in parallel will function exactly as a single instance of that process: $P \bullet P = P$. This can be used to produce as many "copies" of a process as required: $P = P \bullet P = P \bullet (P \bullet P)$. Thus, a copy of the complete test process can be combined with every sub-block of the system using the associative and commutative properties of $\bullet$.

The test vector need never be decomposed, as is done with local constructive simulation.

Using laws [$\bullet \oplus$], [$\bullet_I$] and [$\bullet_C$], we can show why nondeterminism invalidates the idempotency property. For a nondeterministic choice sum $P = Q \oplus R$, the following expression arises:

$$P \bullet P \quad = \quad (Q \oplus R) \bullet (Q \oplus R) \quad = \quad Q \oplus (Q \bullet R) \oplus R \quad \neq \quad P$$

Test patterns should always be deterministic, since they tend to be sequences of actions with no choice sums of any kind, so this problem does not arise.

The three input adder can be used again to contrast this simulation approach with the local version discussed above:

$$\begin{aligned}
\mathit{InAdder} \quad &\Leftarrow \quad \{\mathtt{i1}\triangleleft\mathtt{1} \quad \mathtt{i2}\triangleleft\mathtt{2} \quad \mathtt{i3}\triangleleft\mathtt{3}\}\,\Delta_{\{\mathtt{i2},\,\mathtt{i3}\}} \\
\mathit{TestSystem} \quad &\Leftarrow \quad \mathit{InAdder} \bullet \mathit{System} \\
&= \quad \mathit{InAdder} \bullet ((\mathit{Adder1} \bullet \mathit{Adder2}) - \mathtt{c}) \\
&= \quad \mathit{InAdder} \bullet \mathit{InAdder} \bullet ((\mathit{Adder1} \bullet \mathit{Adder2}) - \mathtt{c}) \\
&= \quad ((\mathit{InAdder} \bullet \mathit{InAdder}) \bullet (\mathit{Adder1} \bullet \mathit{Adder2})) - \mathtt{c} \\
&= \quad ((\mathit{InAdder} \bullet \mathit{Adder1}) \bullet (\mathit{InAdder} \bullet \mathit{Adder2})) - \mathtt{c}
\end{aligned}$$

InAdder can be moved inside the abstraction since it does not contain the c label in its sort and is therefore not affected.

Only the relevant parts of *InAdder* will synchronize with the the inputs of each *Adder* element. Again, the behavior of the components has been limited before the composition of the complete system, reducing intermediate term generation considerably.

It was noted earlier that local constructive simulations are amenable to execution on a multiprocessor machine, with each portion running on its own processor. Simulation in the global constructive style is better suited to a shared memory system in which the test pattern is kept in the central store. Multiple processors could access the shared memory, but this usually involves complicated synchronization mechanisms, so we can conclude that they global approach is preferable for a uniprocessor implementation.

4.3.3 Synchronizing Simulations

When a simulation is run, the designer usually indicates events and/or channels to be monitored. The results of this monitoring are converted to a form— usually wave-forms—suitable for human perusal. The designer then manually examines the data looking for performance errors. In doing so, an implicit comparison is being made with some idea of the expected behavior. Sometimes this idea has actually been formalized by a written specification.

A mathematical framework, such as the one used here, provides an ideal language for writing unambiguous specifications. If the construction of the system is described in the same language, direct comparisons may be made between the two. It seems a pity to have the specification, the system, the test vector, and the simulation results all in the same representation and then rely on a "by eye" comparison with the expected behavior. A far better approach would be to apply the vector to both the specification and the constructed system in parallel and then attach a comparison mechanism to the two sets of outputs.

Recall that the Dot Operator has the property of deadlocking a system if a communication is presented that the intended receiver is not ready to synchronize on and no other action is possible. This property can be utilized to produce a simple comparison mechanism. If the specification produces the same output events as the constructed system, the two will continue to synchronize for as long as the expansion is carried out. If on the other hand there is a discrepancy, the expansion will deadlock at that point. The events leading up to the deadlock can then be examined to see what caused the error. This technique will be called a *synchronizing simulation*, since the specification and implementation must synchronize for the simulation to progress. Figure 4–6 shows how the various parts fit together.

Example: A simple specification can be written for a latch as follows, ignoring for the moment the qbar output line:

part *LATCHSpec* $\{\mathtt{s},\ \mathtt{r},\ \mathtt{q}\}$ $\langle$

$$\mathit{Reset} \Leftarrow \{\mathtt{s}\triangleright 1\}\{\mathtt{q}\triangleleft 1\}\,\mathit{Set} + \{\mathtt{r}\triangleright x\}\,\mathit{Reset}$$
$$\mathit{Set} \Leftarrow \{\mathtt{r}\triangleright 1\}\{\mathtt{q}\triangleleft 0\}\,\mathit{Reset} + \{\mathtt{s}\triangleright x\}\,\mathit{Set}$$

$\rangle$

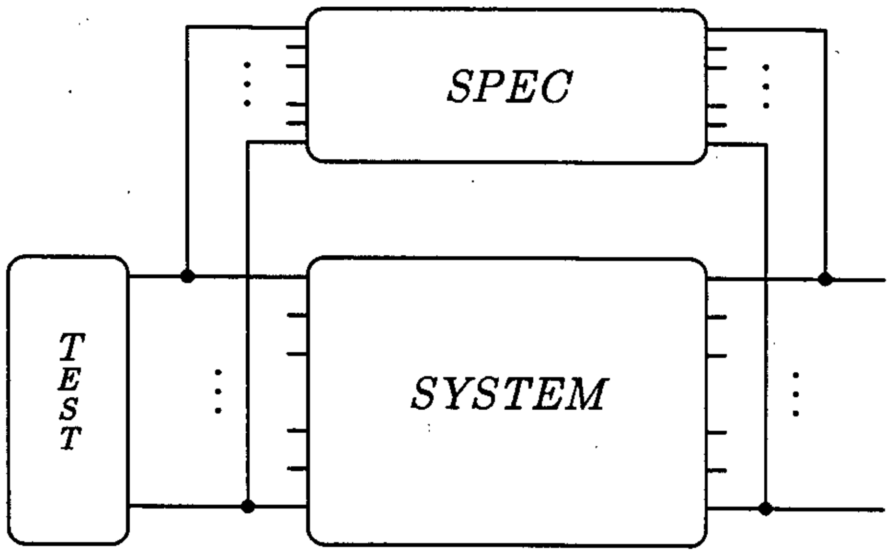


Figure 4–6: Simulating a system and its specification in parallel

For an implementation of this specification, we refer back to the cross-coupled NOR gates discussed on page 86. It has the same sort as *LATCHSpec*, except for the qbar line, and the two may be placed in parallel using the Dot Operator to conduct a synchronizing simulation. To verify that both perform the *set* operation properly (given that they start in the *Reset* state), we apply a test agent that generates a s◁1 signal and expand.

$$\textit{TestInput} \textbf{ of sort } \{\texttt{s, r, t}\} \;\Leftarrow\; \{\texttt{s}\lhd\texttt{1}\}\{\texttt{t}\}^4\Delta_{\{\texttt{s, r, t}\}}$$

$$\begin{aligned}
\textit{System} &\Leftarrow \textit{TestInput} \bullet \textit{LATCHSpec} \bullet \textit{LATCHImp} \\
&= \{\texttt{s}\lhd\texttt{1}\ \ \texttt{t}\}\{\texttt{qbar}\lhd\texttt{0}\ \ \texttt{t}\}\{\texttt{q}\lhd\texttt{1}\ \ \texttt{t}\}\{\texttt{t}\}\{\texttt{t}\} \\
&\qquad (\textit{NA}(0,0,1,1) \bullet \textit{NB}(1,1,0,0) \bullet \textit{Set} \bullet \Delta_{\{\texttt{s, r, t}\}}) \\
&= \{\texttt{s}\lhd\texttt{1}\ \ \texttt{t}\}\{\texttt{qbar}\lhd\texttt{0}\ \ \texttt{t}\}\{\texttt{q}\lhd\texttt{1}\ \ \texttt{t}\}\{\texttt{t}\}\{\texttt{t}\}\Delta_{\{\texttt{s, r, t, q, qbar}\}}
\end{aligned}$$

Examining the output sequence reveals that it does not deadlock until explicitly forced to do so by the test agent. The specification has synchronized with the implementation at every step, implying that it has been satisfied. We have thus shown that *LATCHImp* satisfies *LATCHSpec* when both start in the reset state, an s event is signalled, and time is allowed to progress for four ticks.

The *TestInput* agent requires a closer examination. It outputs a test stimulus and then feeds a sequence of ticks down the input channels. This sequence stimulates the "wait for input" portion of the implementation while the results of the first input percolate through. It acts as an implicit Universal Clock with a lifetime of four ticks. Note that the test agent cannot be applied to the specification alone since it does not satisfy the requirement that its sort be a subset of the target's sort (the specification has no notion of explicit time in the form of a t label). The result would be all possible interleavings of the t's and the output events—a complex tree that is hard to examine. Alternatively, we could have defined *TestInput* to be an asynchronous agent of the form $\{s \lhd 1\}\ \Delta_{\{s,\ r\}}$. This would compose well with the spec, but causes problems when the implementation is present. Its sort doesn't contain t and once again there would be an explosion of interleavings.

Another point to notice is the method of terminating the expansion. The length of the simulation is specified as the stimulus plus four ticks, equalling five ticks total before the expansion deadlocks. This period had to be determined *a priori* as being sufficiently long to obtain a stable output. An alternative method would be to define a termination agent that monitors the sequence being generated and deadlocks the expansion when it considers the outputs to have stabilized. This is more elegant, but has the dangerous feature that an oscillating output would prevent termination (unless, of course, the agent were sophisticated enough to detect this case).

Observe that we carefully avoided including the qbar event in the specification. At first one might be tempted to specify that the q and qbar events occur simultaneously, as conceptually they should. Unfortunately, there is no way to construct a latch that exhibits such behavior. The delay through the feedback loop will always cause one event to precede the other. The order is usually immaterial from the point of view of the specification, especially if the circuit is only part of a larger system. Unless knowledge exists about the structure of the actual implementation (*i.e.*, whether q or qbar occurs first), a specification cannot be produced that will match it. This is not much of a problem when human judgement is used to examine the simulation results, but makes synchronizing simulation as it now stands almost impossible to automate.

4.3.4 Imprecise Specifications

One way to avoid incorporating implementation details into a specification is to allow a small amount of "fuzziness". Usually, this fuzziness takes the form of ignoring the temporal ordering of certain events. If the uncertainty is represented by a deterministic choice sum of all possible orderings, the correct ordering will then be chosen when the specification is composed with the implementation. The spec can then be rewritten to remove the ambiguity so that it more closely mirrors the behavior of the implementation. The latch specification, for example, can be modified to include the qbar event either before or after q as follows:

part *LATCHSpec* {s, r, q, qbar} ⟨

$$
\begin{aligned}
\textit{Reset} \Leftarrow\ & \{\mathtt{s}\triangleright 1\}\,(\{\mathtt{qbar}\triangleleft 0\}\,\{\mathtt{q}\triangleleft 1\}\,\textit{Set} + \{\mathtt{q}\triangleleft 1\}\,\{\mathtt{qbar}\triangleleft 0\}\,\textit{Set}) \\
+\ & \{\mathtt{r}\triangleright 1\}\,\textit{Reset} + \{\mathtt{r}\triangleright 0\}\,\textit{Reset} \\
\textit{Set} \Leftarrow\ & \{\mathtt{r}\triangleright 1\}\,(\{\mathtt{q}\triangleleft 0\}\,\{\mathtt{qbar}\triangleleft 1\}\,\textit{Reset} + \{\mathtt{qbar}\triangleleft 1\}\,\{\mathtt{q}\triangleleft 0\}\,\textit{Reset}) \\
+\ & \{\mathtt{s}\triangleright 1\}\,\textit{Set} + \{\mathtt{s}\triangleright 0\}\,\textit{Set}
\end{aligned}
$$

⟩

The modified part, when placed in the test system described in the last section, produces the same output sequence as the original with qbar occurring before q. Once this ordering has been revealed by the simulation, the other choice branch can be removed from *LATCHSpec*, yielding a specification that correctly reflects the implementations behavior, yet is much simpler.

The idea of generating all possible orderings should seem familiar; it was discussed in Chapter 3 when dealing with the temporal permutation operators. In that chapter, the operators were used to define intervals during which we don't have an exact knowledge of event ordering. The usage here is similar, except that we wish the environment (in this case the implementation) to chose one of the orderings. The above preliminary spec can be rewritten using the deterministic temporal permutation operator π as follows:

part *LATCHSpec* {s, r, q, qbar} (

$$\begin{aligned} \mathit{Reset} &\Leftarrow \{\mathtt{s}\triangleright 1\}\,\pi(\{\mathtt{qbar}\triangleleft 0\}, \{\mathtt{q}\triangleleft 1\})\,\mathit{Set} \\ &\quad + \{\mathtt{r}\triangleright 1\}\,\mathit{Reset} + \{\mathtt{r}\triangleright 0\}\,\mathit{Reset} \\ \mathit{Set} &\Leftarrow \{\mathtt{r}\triangleright 1\}\,\pi(\{\mathtt{q}\triangleleft 0\}, \{\mathtt{qbar}\triangleleft 1\})\,\mathit{Reset} \\ &\quad + \{\mathtt{s}\triangleright 1\}\,\mathit{Set} + \{\mathtt{s}\triangleright 0\}\,\mathit{Set} \end{aligned}$$

)

Strictly speaking the nondeterministic permutation operator should be used, since this indicates that we do not care at all about the order of the output events. Unfortunately, the tools for dealing with nondeterminism are not yet available, so we must remain with the deterministic version for now. A later chapter will show how this problem can be solved properly.

Designers of formal specifications for programming languages have encountered the need for a similar permutation operator (called ARBITRARY-PERMUTATION in [Anderson 77]). A common problem is specifying what the value of an expression should be without specifying the order in which is calculated. For example, the value of the expression $a + b + c$ (where + stands for addition) should not require that a be evaluated before b and c. This would place an unnecessary burden on the implementors of the language with no real benefit. Instead, the result should be specified as simply "the sum of the values of a, b and c".

4.3.5 Probes, Taps and Transformers

One of the most important applications for timing simulators is in revealing transient events that occur during the operation of a circuit. Spikes, oscillations and other anomalies can cause a circuit to function in a totally unexpected manner. Hardware designers have traditionally built prototypes of designs and used various tools, such as test meters, oscilloscopes and logic analyzers to investigate malfunctions. These useful aids are not available when a design is still in "soft" form, *i.e.*, represented by a software description, nor when it has been committed to silicon with many of the internal nodes unprobeable. Ideally, one would wish to have a means of statically determining the existence of these anomalies, but in general this an extremely difficult task. Methods exist that automatically check for certain limited forms of hazards, such as critical races, in boolean combinational circuits [Friedman 77]. Others forms of hazard detection, such as spike detection, are incorporated directly into switch and gate level simulators [Breuer 75, Newton 81]. These approaches are effective, but limited. To achieve a high degree of confidence that a design is free of error inducing transients, an extensible toolkit of detectors is required. In other words, it would be desirable to have software analogues of the hardware debugging aids.

We have seen that the system, its specification and its test stimuli can all be represented in the same framework. Why not do the same for the output analysis tools? In this section, a few such constructs are developed and their application demonstrated.

Probes: It is often useful to be able to attach some form of monitoring device to a node. A voltage monitor could either communicate voltage changes at a node to the user or, alternatively, warn of illegal values. Glitches (undesirable voltage transients) and other momentary phenomena can be detected with a spike detector. These are just some examples of a construct that we shall call a software *probe*. Probes are passive watchers that observe events on a particular line or set of lines. They may use a history mechanism to decide if an event is of interest based on the ones that preceded it. In short, any behavior that can be specified by a sequence can be detected. Probes should only indulge in output communications with the environment—usually to report the detection of an error—and not interact with the system under observation.

Probes must, naturally, be introduced into the system under test at the same level of the hierarchy as the channel(s) being observed. The probe's input ports are attached by simply renaming them to have the same labels as the channels.

Example: Suppose we wish to detect the occurrence of a spike on a voltage line. A spike is arbitrarily defined as a voltage transition that occurs less than two time units after another transition. A "window" into the past that will catch the spikes is kept by counting ticks that occur since the last input transition. If this count is two or less when the line changes again then a spike has been detected and a `spikeDetected` error is signalled.

$$\begin{array}{l}
\textbf{part}\ \mathit{Detector}\ \{\texttt{in, t, spikeDetected}\}\ \langle \\
\quad \mathit{Detecting}(0) \Leftarrow \mathrm{WAIT}_{\texttt{t}}(\{\texttt{in}\triangleright x\ \ \texttt{t}\}\ \mathit{Detecting}(2)) \\
\quad \mathit{Detecting}(c) \Leftarrow \{\texttt{t}\}\ \mathit{Detecting}(c-1) \\
\qquad\qquad\qquad + \{\texttt{in}\triangleright x\ \ \texttt{spikeDetected}\ \ \texttt{t}\}\ \mathit{Detecting}(2) \\
\rangle
\end{array}$$

The in line accepts a value of any type, since a change of any kind can be used to signal the spike.

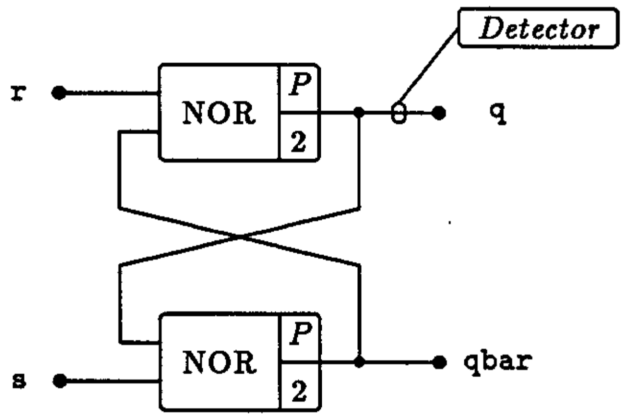
We can use the detector to monitor the q output of the RS latch used above to see what happens when a spike is applied to the s line. The structure of the system is given in Figure 4–7(a) and a plot of the events generated during its expansion for 10 ticks is shown in Figure 4–7(b). The NOR gates have a propagation delay of two time units and the complete system is defined by:

$$\begin{array}{l}
\mathit{Ticks}\ \textbf{of sort}\ \{\texttt{s, r, t}\} \Leftarrow \{\texttt{t}\}\ \mathit{Ticks} \\
\mathit{Input} \Leftarrow \{\texttt{t}\}\{\texttt{s}\triangleleft 1\ \ \texttt{t}\}\{\texttt{s}\triangleleft 0\ \texttt{t}\}\ \mathit{Ticks} \\
\mathit{System} \Leftarrow \mathit{Input} \bullet \mathit{LATCHImp} \bullet \mathit{Detector}\ [\texttt{q/in}]
\end{array}$$

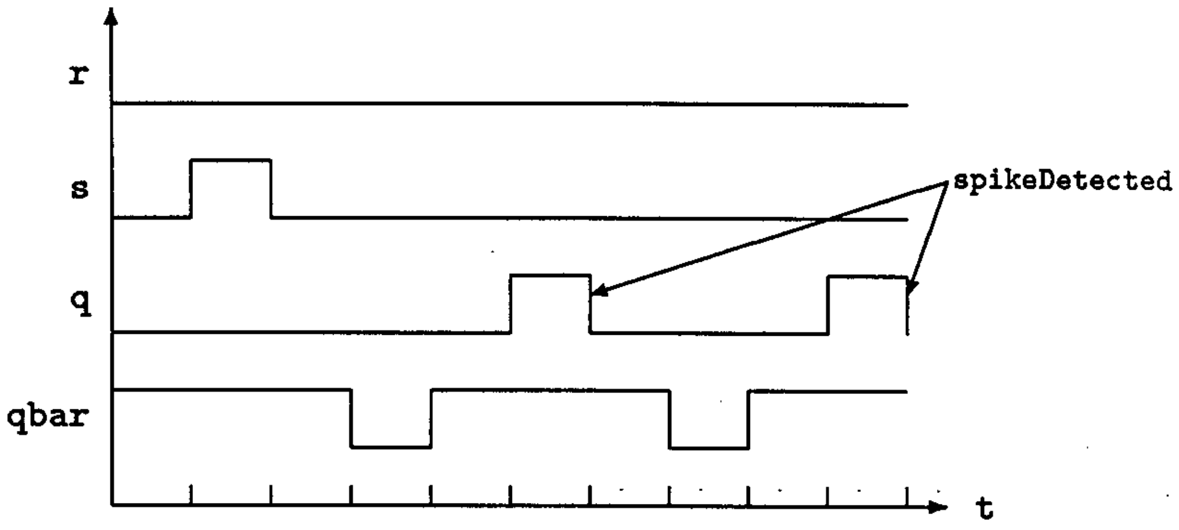
Taps: Counterparts to probes are *taps*. Rather than passively observing, taps "splice in" to a communication link and have the ability to modify the values being passed. They can be used to inject test stimuli into a port in much the same manner as a hierarchical test agent, but have the added ability to construct the stimulus based on the previous traffic through the port. Illegal values can be trapped, replaced by a legal one and either reported to the user or the simulation can be halted and a debugger process enabled.

A very simple tap that checks if integers passing along a channel are less than zero, signals the fact to an error log, and changes the value to zero is given below:

$$\begin{array}{l}
\mathit{IntChecker} \Leftarrow \{\texttt{in}\triangleright(x : \textbf{integer})\}\ \textbf{if}\ x < 0\ \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad \{\texttt{errorlog}\triangleleft x\ \ \texttt{out}\triangleleft 0\}\ \mathit{IntChecker} \\
\qquad\qquad\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad\qquad\qquad \{\texttt{out}\triangleleft x\}\ \mathit{IntChecker}
\end{array}$$



(*a*) Circuit.



(*b*) Output waveforms.

Figure 4–7: Spike detector applied to a latch

Transformers: A sub-class of taps are *transformers*. Transformers input a value and instantaneously map it to another value. They are primarily used for type conversions between different representations of the same value.

Quite often in mixed level descriptions, data of one type must be passed to a functional unit that accepts it as another. If data channels are typed, this should result in either a type mismatch error or the automatic generation of a transformer to convert the two. A similar thing happens in most high level programming languages when real variables, for example, are combined with integers in an expression. The integers are automatically converted to real numbers.

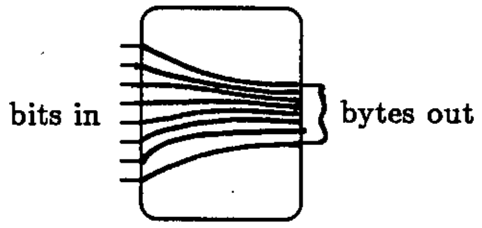


Figure 4–8: An eight bit bus to byte transformer

A common application for a transformer in simulations of digital circuits is to "bundle" a number of lines with bit values into a single more abstract value. For example, an eight bit bus may be viewed as an integer in the range 0 to 127, as an ASCII character, or any of numerous other choices. Such a transformer simply detects a change on any of the bit-lines and recomputes its conversion function with the new value(s). Defining a choice sum that detects any of the 2^n possible changes on an n-bit wide bus is tedious by when done by hand, but is precisely what the Any Actions operator discussed on page 36 was designed for. The similarity to the Generic Box concept is obvious, differing only in that the output function is calculated instantaneously, instead of having an unspecified delay. The following expression, describing an n-bit to integer conversion unit, should clarify matters:

part $\mathit{Bits\_to\_Int}(x_0, \ldots, x_n : \mathbf{bit})\ \{\mathrm{d}_{0\ldots n} : \mathbf{bit},\ \mathit{out} : \mathbf{int}\}\ \langle$

$$C(\tilde{x}) \Leftarrow \mathrm{ANY}(\{\mathrm{d}_0 \triangleright x_0\ \ \mathrm{out} \triangleleft f(\tilde{x})\} \ldots \{\mathrm{d}_n \triangleright x_n\ \ \mathrm{out} \triangleleft f(\tilde{x})\})\ C(\tilde{x})$$

$$f(\tilde{x}) = (2^0 \times x_0) + \cdots + (2^n \times x_n)$$

$\rangle$

The reason this produces the required behavior is that the scope of the variables bound by the input channels includes the guard itself. Thus if ANY generates the term $\{\mathrm{d}_0 \triangleright x_0\ \mathrm{d}_1 \triangleright x_1\ \mathrm{out} \triangleleft f(\tilde{x})\}\ C(\tilde{x})$, then the x_0 and x_1 in $\tilde{x}$ will be bound by the input channels, whilst $x_2, \ldots, x_n$ will be bound by the state variables.

Suppose we have a 4 bit to integer transformer that is attached to a bus with all lines low. Lines d_1 and d_2 then go high simultaneously, producing:

$$
\begin{aligned}
&\mathit{Bits\_to\_Int}(0,0,0,0) \bullet (\{\mathrm{d}_1{\triangleleft}1 \;\; \mathrm{d}_2{\triangleleft}1\}\, \Delta_D) \\
&\qquad = \{\mathrm{d}_1{\triangleleft}1 \;\; \mathrm{d}_2{\triangleleft}1 \;\; \mathtt{out}{\triangleleft}f(0,1,1,0)\}\, \big(\mathit{Bits\_to\_Int}(0,1,1,0) \bullet \Delta_D\big) \\
&\qquad = \{\mathrm{d}_1{\triangleleft}1 \;\; \mathrm{d}_2{\triangleleft}1 \;\; \mathtt{out}{\triangleleft}6\}\, \big(\mathit{Bits\_to\_Int}(0,1,1,0) \bullet \Delta_D\big)
\end{aligned}
$$

Where: $D = \{\mathrm{d}_0, \mathrm{d}_1, \mathrm{d}_2, \mathrm{d}_3\}$.

The correct conversion value (6) is instantaneously output. This lack of calculation time is not as unrealistic as it seems since we are not actually doing any calculations, but merely changing the way the value is viewed. Care must be taken to use transformers purely as filters. For example, it might be tempting to convert a serial bit stream into a sequence of integers with a transformer, but this would be incorrect since we are really specifying a circuit that can perform the conversion and not simply viewing the stream in a different way.

Transformers with state can be used to convert between more complicated abstractions. A signal ϕ_1, for instance, might be modeled at a lower level of abstraction as a sequence of voltages. A transformer would then be used to sample this stream and output a ϕ_1 event when the voltage crosses a predetermined threshold.

A similar application of transformers is particularly useful as a documentation mechanism. For example, if the occurrence of a particular value on a channel is used to control several processes, a transformer can be defined that assigns a descriptive name to the event. Events of this type are particularly common in clocked systems. A typical system clock might be defined as:

$$C \;\Leftarrow\; \{\varphi{\triangleleft}0\}\,\{\varphi{\triangleleft}1\}\,C$$

Now suppose that most of the flip-flops controlled by the clock change state on the negative going edge (*i.e.,* on a $\{\varphi{\triangleleft}0\}$ event). Normally the definitions of the flip-flops would have to include terms for absorbing the unwanted $\{\varphi{\triangleleft}1\}$ events. A transformer, however, can be used to assign a more descriptive name, say `clk`, to the negative edge event thus not only simplifying the definitions, but also makes them easier to understand. Here is a sample definition for such a transformer:

$$T \;\Leftarrow\; \{\varphi{\triangleleft}0 \;\; \mathtt{clk}\}\,T$$

Composing T with C and then with the system produces the required result.

4.4 Contributions of this Chapter

This chapter cleaned up and extended several simulation techniques developed by Milne. It was shown how the basic simulation technique could be used to conduct symbolic simulations with no extra effort. Then Milne's notion of constructive simulation was subdivided into two sub-classes and a small stumbling block in one of these (local constructive simulation) was removed. Synchronizing simulations were proposed as a way of comparing the results of simulating specifications and their implementations with little extra effort. Finally, several types of tools for examining the results of a simulation were developed and their application illustrated through examples.

Chapter 5

Verification

The introduction to Chapter 4 discussed the differences between the terms simulation and verification and then went on to examine simulation in detail. Simulation was defined as the procedure whereby a particular path through the synchronization tree describing a process's behavior is selected by the application of sequences of test stimuli. The resulting path is checked for validity by comparing it with some idea of what the process should do (a specification), either manually or automatically,

Suppose that instead of using a test pattern to select a path, we were to use the specification itself? The specification can, after all, be expressed as a synchronization tree and thereby compared directly with that of the implementation. A careful examination of the points where they fail to match will show whether the implementation does indeed satisfy the specification.

The following sections explore a variety of ways of performing the comparison between two synchronization trees. The first method, discussed only briefly, involves transforming an implementation tree into that of its specification by applying laws of the calculus. This is the traditional way of proving a system correct but, as we shall see, it is too restrictive to perform certain desirable comparisons. Based on these failings, the case is then made for another approach using *partial specifications.* Partial specifications, as the name implies, describe only a portion of the potential behavior of a process. The understanding is that the process will never be used in a way that exploits capabilities unmentioned in the specification. As can be imagined, it is difficult to avoid doing this unknowingly.

Two processes that function properly when their specifications are composed may cease to do so when their unspecified portions interact. A technique for preventing these undesirable interactions is evolved using *constraints* on the environments in which a specification may be placed. Methods for generating and combining such constraints form the central topic of this chapter. Nondeterminism causes problems in these methods and so additional operations are defined to handle it in the final sections.

5.1 Total Specifications and Transformations

The normal approach to verifying if a constructed system is correct is to write a specification of its intended function and then transform the construction using laws of the calculus into a syntactically equivalent form. Sometimes additional techniques, such as induction, are needed to complete the proof. This approach works only when the two processes have basically the same behaviors, modulo the abstraction of internal communications. In most applications, however, this is seldom the case. Knowledge about the environment in which the system is to be used will often permit a dramatic simplification of its specification. The presence of a global clock in a register transfer system, for example, allows one to largely ignore transient values ("glitches") produced by blocks of combinational logic (the clock samples the outputs of the blocks after they have stabilized). Proving a simplified specification equivalent to its implementation is usually impossible with a transformational approach because one will have capabilities (*e.g.*, to generate glitches) unavailable to the other.

Here is a simple example that illustrates a transformational proof and reveals its limitations:

Example: Suppose we have two "doubling" processes that input a value, output twice that value some time later and wait for a clock event before recursing. Each of the doublers is described by:

$$\begin{aligned} D &\Leftarrow \{\mathtt{b}{\triangleright}x\}\,\{\mathtt{c}{\triangleleft}(2\cdot x)\}\,\{\mathtt{clk}\}\,D \\ D' &\Leftarrow D\,[\mathtt{c}/\mathtt{b},\;\; \mathtt{d}/\mathtt{c}] \end{aligned}$$

Cascading two such processes should produce a "quadrupler" unit:

$$Q \Leftarrow (D \bullet D') - \mathtt{c}$$

which can be specified by:

$$S \Leftarrow \{\mathtt{b}\triangleright x\}\,\{\mathtt{d}\triangleleft(4\cdot x)\}\,\{\mathtt{clk}\}\,S$$

Proving that $S = Q$ is straightforward:

$$\begin{array}{rcll}
Q & = & (D \bullet D') - \mathtt{c} & \text{by defn. } Q \\
 & = & \big(\{\mathtt{b}\triangleright x\}\,\{\mathtt{c}\triangleleft(2\cdot x)\}\,\{\mathtt{clk}\}\,D & \text{by defn. } D,\ D' \\
 & & \bullet\,\{\mathtt{c}\triangleright y\}\,\{\mathtt{d}\triangleleft(2\cdot y)\}\,\{\mathtt{clk}\}\,D'\big) - \mathtt{c} & \\
 & = & \big(\{\mathtt{b}\triangleright x\}\,\{\mathtt{c}\triangleleft(2\cdot x)\}\,\{\mathtt{d}\triangleleft(2\cdot(2\cdot x))\}\,\{\mathtt{clk}\} & \\
 & & \quad (D \bullet D')\big) - \mathtt{c} & [\bullet\ +]\text{ four times} \\
 & = & \{\mathtt{b}\triangleright x\}\,\{\mathtt{d}\triangleleft(2\cdot(2\cdot x))\}\,\{\mathtt{clk}\}\,Q & [-\ +]\text{ four times} \\
 & = & \{\mathtt{b}\triangleright x\}\,\{\mathtt{d}\triangleleft(4\cdot x)\}\,\{\mathtt{clk}\}\,Q & \text{Arithmetic} \\
 & = & S & \text{Fixpoint Induction}
\end{array}$$

Notice that if the `clk` guard were not present, D could recurse and input a new value on b before D' has a chance to output on d. Applying law [• +] to such a case would result in all possible interleavings of b and d. No CIRCAL laws can be applied to transform the resulting expression into a form syntactically equivalent to S without the `clk` guard, since the latter does not contain a corresponding interleaving. Yet S without the guard is still a valid description of what the system will do provided that b events always wait until all pending d events have occurred.

For a transformational proof to succeed, S would have to be changed to reflect the possibility of b and d interleaving. But such a specification is not something that one would naturally write for a quadrupler unless knowledge of its eventual implementation were available. Requiring such advance knowledge makes a mockery of the whole concept of top-down design. If even such a simple example as this encounters problems in comparing implementations and specifications, what hope have we of refining truly complex designs? Some degree of freedom to write simplier but still valid specifications is needed and this is presented in the following sections.

5.2 Partial Specifications

The main problem with total specifications, as we have seen, is that they are seldom natural to formulate. Many systems, particularly digital ones, are designed around extremely simplified ideas about the behavior of primitive components. Thinking of a device as operating on binary data is far easier than analyzing it in terms of voltage and current laws, which in turn, are simpler than the physical laws governing electron motion. The goal in constructing a specification is to keep it as simple as possible yet still correctly reflect the behavior of an implementation at any lower level of abstraction. If the effects of these lower abstraction levels always has a direct influence on the higher levels, not only do specifications become difficult to write, but the whole notion of making behaviors manageable through hierarchical design is rendered ineffectual.

Adding to these problems is the common practice of using part of the functionality of an agent to implement a simpler function. The agent is placed in an environment that never attempts to use its extra capabilities. A specification crafted for this restricted environment will probably never match an implementation designed for more general uses. The usual specification for a Set/Reset flipflop, for example, says that it can accept a signal on either the set or the reset lines and alter the state accordingly. A perfectly acceptable implementation might have the added option of accepting signals on both lines simultaneously and generating an undefined output. Clearly, there is no way to transform the implementation into the spec in isolation since the former has more capabilities than the latter. Should the flipflop be used in an environment that never allows both signals to occur simultaneously, however, both should be considered equally acceptable.

What is needed is the concept of a *partial specification* that contains just enough information to fulfill the requirements of the environment. Provided that the points at which it disagrees with the implementation do not affect its functionality in that environment, we can say that it has been *satisfied* by the implementation. Such a satisfaction relation would provide the necessary freedom to produce sensible implementations of simple specifications. Care must be taken not to make the relation too free, since illegal or unacceptable implementations may then be declared valid. A compromise will usually have to be found between simplicity and realism.

So how can an implementation be checked to see if it satisfies a specification? In Section 4.3.2, we saw how the idempotency property of the Dot Operator guarantees that an agent P is identical to the composition of two copies of P. Now if one of the copies is replaced by another agent P', having the same sort as P and performing the same sequences of actions, the equality should be preserved (up to the presence of resultants of P' in the resultants of P):

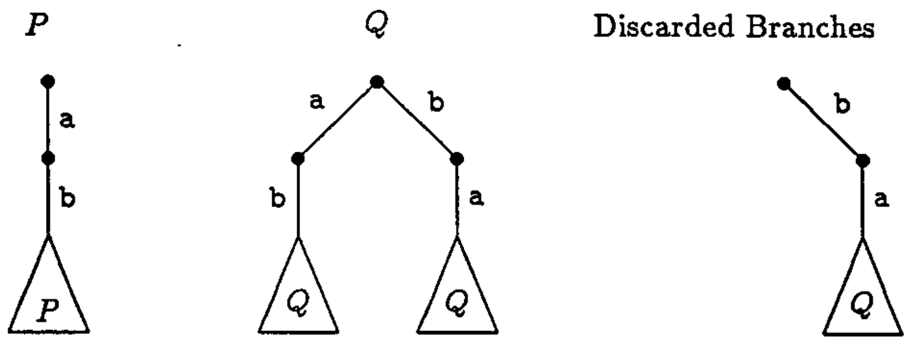
$$(P \bullet P') \quad = \quad P \tag{5.1}$$

P' must synchronize with every action of P.

Observe that Equation 5.1 remains true even if P' is capable of sequences in addition to those of P. For example:

$$\begin{aligned} P &\Leftarrow \{\mathtt{a}\}\,\{\mathtt{b}\}\,P \\ Q &\Leftarrow \{\mathtt{a}\}\,\{\mathtt{b}\}\,Q + \{\mathtt{b}\}\,\{\mathtt{a}\}\,Q \\ P \bullet Q &= \{\mathtt{a}\}\,\{\mathtt{b}\}\,(P \bullet Q) \end{aligned}$$

Since the composition has exactly the same structure as P, we can conclude that Q is capable of exactly the same actions as P. The "action matching" property of the Dot Operator was exploited to remove the extraneous actions available to Q. If at any point an action of P had not been matched by a corresponding action in Q, either the composition would have deadlocked or the resulting expression would not have been syntactically equivalent to P. Representing the processes by synchronization trees makes this technique easy to visualize. Here are the trees for the above example:



At each node there might be branches belonging to Q that do not have corresponding branches in P. These branches and their successor trees are discarded when the two are composed with the Dot Operator (in this case the branch labelled by b at the first node). Actions that label the first branch of a tree of discarded branches (the b label in this case) will be referred to as the *Distinguishing Actions*, since experiments on the two processes using these actions can distinguish them.

Based on this example, we might be tempted to define a *satisfaction* relation between two agents in terms of composition and pattern matching of the result with the simpler of the two. Unfortunately, this definition fails to cope with the nondeterministic choice operator since the idempotency property is valid only for *deterministic* choice sums (see Definition 2.5.1). A simple additional requirement rectifies this deficiency and we obtain the following definition of satisfaction due to Milne [Milne 84b]:

Definition 5.2.1 Strong Satisfaction

An implementation I is said to *satisfy* a deterministic partial specification given by

$$S \Leftarrow \sum_{\substack{i \in I \\ \alpha_j \neq \alpha_k}} \alpha_i S_i$$

written as

$$I \ \underline{\text{sat}} \ S$$

if the following three conditions hold:

(i) $\text{sortOf}(I) = \text{sortOf}(S)$.

(ii) $S \bullet I = \sum_{\substack{i \in I \\ \alpha_j \neq \alpha_k}} \alpha_i (S_i \bullet I_i)$ and $I_i \ \underline{\text{sat}} \ S_i$ for all $i \in I$.

(iii) $S \bullet I = \textcircled{\Sigma}_{i \in I} S \bullet I_i$ and $I_i \ \underline{\text{sat}} \ S$ for all $i \in I$.

□

The $\alpha_j \neq \alpha_k$ qualification ensures that the sums are deterministic.

The first and second requirements perform the composition and pattern matching operations discussed above. The third requirement says that *every* nondeterministic branch in the implementation must match the specification. The net effect is to make all the nondeterministic branches look the same. An alternative relation, the $\underline{\text{maysat}}$ relation, is defined by relaxing this requirement so that only one nondeterministic branch need match. Except for certain stylized cases, this *weak* relation is not very useful since the implementation may nondeterministically decide to act quite differently from the specification.

Definition 5.2.2 **Weak Satisfaction** [Milne 84b].

An implementation I *may satisfy* a deterministic partial specification given by

$$S \Leftarrow \sum_{\substack{i \in I \\ \alpha_j \neq \alpha_k}} \alpha_i S_i$$

written as

$$I \ \underline{\text{maysat}}\ S$$

if the following three conditions hold:

(i) $\text{sortOf}(I) = \text{sortOf}(S)$.

(ii) $S \bullet I = \sum_{\substack{i \in I \\ \alpha_j \neq \alpha_k}} \alpha_i (S_i \bullet I_i)$ and $I_i \ \underline{\text{maysat}}\ S_i$ for all $i \in I$.

(iii) $S \bullet I = \textstyle\bigcirc\!\!\!\!\sum_{i \in I} S \bullet I_i$ and $I_i \ \underline{\text{maysat}}\ S$ $\underline{\text{for at least one}}$ $i \in I$.

□

To see how the satisfaction relations can be applied, let us return to the example from the last section involving doubling agents. This time the clk guard will be removed so that the implementation can input new values on b before the previous calculation has been output on d. Here are the modified equations:

$$\begin{aligned} D &\Leftarrow \{\mathsf{b} \rhd x\}\{\mathsf{c} \lhd (2 \cdot x)\}\, D \\ D' &\Leftarrow \{\mathsf{c} \rhd x\}\{\mathsf{d} \lhd (2 \cdot x)\}\, D \\ Q &\Leftarrow (D \bullet D') - \mathsf{c} \\ S &\Leftarrow \{\mathsf{b} \rhd x\}\{\mathsf{d} \lhd (4 \cdot x)\}\, S \end{aligned}$$

We wish to show that

$$Q \quad \underline{\mathrm{sat}} \quad S$$

by applying Definition 5.2.1. The two have identical sorts, so the first requirement is met and there are no nondeterministic choices to worry about, so the third requirement can be ignored. This leaves requirement (*ii*). Composing S and Q produces:

$$\begin{aligned}
S \bullet Q &= \{\mathsf{b}\triangleright x\}\,(S_1 \bullet Q_1) \\
S_1 \bullet Q_1 &= \{\mathsf{d}\triangleleft(4\cdot x)\}\,S \bullet \left((D \bullet \{\mathsf{d}\triangleleft(4\cdot x)\}\,D') - \mathsf{c}\right) \\
&= \{\mathsf{d}\triangleleft(4\cdot x)\}\,S \bullet \\
&\qquad \left(\{\mathsf{b}\triangleright y\}\,(\ldots) + \{\mathsf{d}\triangleleft(4\cdot x)\}\,(\ldots) + \{\mathsf{b}\triangleright y \quad \mathsf{d}\triangleleft(4\cdot x)\}\,(\ldots)\right) \\
&= \{\mathsf{d}\triangleleft(4\cdot x)\}\,\left((\{\mathsf{b}\triangleright y\}\,(\ldots) \bullet D') - \mathsf{c} \;\bullet\; S]\right) \\
&= \{\mathsf{d}\triangleleft(4\cdot x)\}\,(S \bullet Q)
\end{aligned}$$

So Q does indeed satisfy S. Notice how the composition discarded the extraneous $\{\mathsf{a}\triangleright y\}$ and $\{\mathsf{a}\triangleright y \quad \mathsf{c}\triangleleft(4\cdot x)\}$ choice branches on the second line of the expansion of $S_1 \bullet Q_1$.

One might be tempted to always use S in place of Q in some larger system since it is syntactically much simpler. Consider what this means, however, if both are placed alternatively in parallel with a process defined like this:

$$R \;\Leftarrow\; \{\mathsf{b}\triangleleft 1\}\,\{\mathsf{b}\triangleleft 2\}\,\{\mathsf{d}\triangleright x\}\,R'$$

S would deadlock after accepting the first b communication, since it must output a d before accepting another. Q, on the other hand, can input new values on b while previous values are still percolating through to d. Although S and Q do behave identically in some contexts, they will not when placed parallel with R. Some method is needed indicate what contexts are safe.

The satisfaction relations due to Milne are a step in the right direction toward simplier verification proofs. As we have just seen, however, something more is needed if they are to be used without introducing the possibility of incorrect results.

5.3 Constraints

A time honoured approach to constructing a complex system is that of *modular decomposition*. First the intended behavior of the system is described by some form of specification. Then the specification is decomposed into a collection of "simpler" behaviors that together can be shown to give the required result. These smaller behaviors are in turn decomposed into still smaller units and so on, until some primitive level is reached. This method of continually moving toward a more concrete representation of the system is known as *step-wise refinement*. A high degree of confidence in the correctness of the end result can be obtained by proving that each refinement preserves the semantics of the previous levels of abstraction. Care must be taken, as we saw in the last section, that a refinement interacts properly with the context in which it is placed. It is all to easy to refine a specification for one block in the "safe" environment of the specifications of the other blocks and then forget that when they in turn are refined, unexpected interactions may develop. Figure 5–1 shows how the implementations of two interacting specifications may themselves have additional interactions.

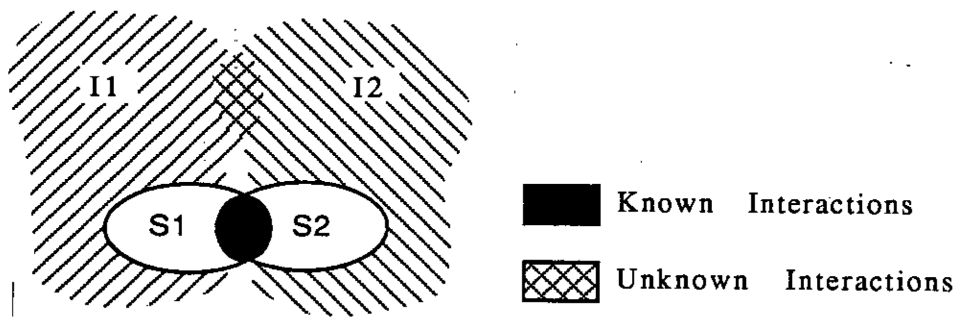


Figure 5–1: Unexpected interactions between two implementations.

Recently, some interesting work has been undertaken in this area. In particular, Larsen [Larsen 86] has developed a notion of *context dependent equivalence* that fits in naturally with the step-wise refinement approach to design. His idea is that information about the context[†] in which a sub-block is to be placed is already available and can perhaps be used to simplify the comparison between it and its refinement. Quite often the refinement will have more capabilities than the process it replaces, having been arrived at by juggling factors unrelated to behavior, such as package count and part availability. In this case, the process and its refinement will not be equivalent at all when used in isolation. When placed in a particular context, however, the differences may become irrelevant if the extra capabilities are never used. Larsen, therefore, parameterizes his equivalence with information about the context so that such a proof will succeed. In particular, for a one-hole context C and two processes P and Q, we have the following:

$$P \equiv_e Q \iff C[\![P]\!] \equiv C[\![Q]\!]$$

for some "environment" $e \in \mathrm{Inf}(C)$ where $\mathrm{Inf}(C)$ is a subset of a domain of information I containing information about the context C. He goes on to define this domain for CCS contexts and presents an ordering corresponding to how discriminating the environment is. The ordering has a minimal element representing the "least restrictive" environment that makes two processes act identically. This element is called the *weakest inner information* about an environment by analogy to Dijkstra's concept of *weakest preconditions* in program verification.

The design process used by most designers is not nearly as structured as advocates of the step-wise refinement methodology would desire. Often a system grows from both directions simultaneously, particularly as the design approaches the more primitive levels. High level design decisions must sometimes be changed when problems are encountered at lower levels (*e.g.*, a given module may not run fast enough when designed with one architecture, but will with another). In addition, experienced designers usually have a "bag of tricks" consisting of clever ways to implement certain common functional units. Ideally these tricks should verified once, then annotated with information about their interface requirements

[†]Remember, a context is a CIRCAL expression with a "hole" that must be filled by another expression to produce a legal behavior.

and stored away in a library for later use. Many VLSI design systems already maintain large databases of pre-designed functional units that are verified in the geometrical sense (for example [Lattice 82, Deas 84]). Individual functional units are handcrafted by experts to fulfill some requirement such as minimum area or maximum speed and stored in the database. Users or higher level tools connect together units of widely varying complexity, from simple pass gates to complete microprocessors, to produce a given design. Each will have restrictions on how other units can interact with it and still receive the expected results. These restrictions are called *constraints*.

Constraints have a direct connection with Larsen's weakest inner information concept. The latter corresponds to the smallest (in some sense) set of constraints on a unit that will allow it to function as specified. Larsen's notion of context dependent equivalence deals only with single hole contexts, which means that once a specification has been refined, its implementation becomes part of the context in which other blocks are refined in turn. Equivalence proofs thus become progressively more difficult as the system becomes less abstract. This section develops an alternative representation for constraints that exploits some of the properties of CIRCAL to limit this growth in complexity. The representation is also readily amenable to the bottom up approach required for producing standard part libraries. Verified parts can be stored in these libraries as triples consisting of a specification, an implementation and a constraint on when the specification may safely be used.

5.3.1 Constraints as Covering Trees

Earlier we saw how the behavior of a process can be represented by a tree. The branches at each node are labelled with the actions corresponding to the choices available to the process at that point. For the moment, let us consider trees that are fully deterministic; each node represents a single state of a process and all branches have distinct labels. Under what conditions, then, can one tree replace another?

Consider two deterministic tree that describe processes having the same sort. At any node, a set of actions (possibly empty) can be constructed from the labels of branches that are the same in both trees. These actions are called the ***common actions*** of the trees (*i.e.*, the actions common to both) and can be calculated for each corresponding pair of nodes. Another set can be constructed from the actions that do not match, and this will be called the ***distinguishing actions*** set. The union of the two sets contains all the actions available to both processes at that point in their evolution. A node pair that has an empty set of common actions corresponds to the two processes doing completely different things. The two are completely distinguishable there, so the branch leading up to this node pair should be added to the distinguishing action set of the parent node pair, since that path should be avoided. If the root nodes of the trees have no common actions, the two processes are not related and can never be used interchangeably.

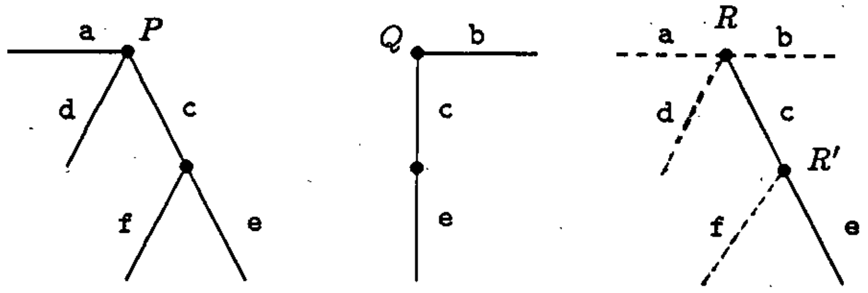


Figure 5–2: Sample covering tree

For any two trees, another tree can be formed by creating a node for each node pair and labelling the branches exiting from it with members of the common-action set. This tree is in some sense the "intersection" or "greatest common denominator" of the two original trees. Clearly, it can replace them in any context that does not attempt to interact with the distinguishing actions.

Figure 5–2 shows two trees P and Q, and their common denominator R. The distinguishing actions are indicated by dashed lines and actions rejected by both processes are not shown. The distinguishing actions of R are contained in the set of label-sets { {a}, {b}, {d} } and those of R' in { {f} }[†]. Both P and Q may be replaced by R provided that no a, b, or d communications are engaged in at the top level. R is said to be the *covering tree* of P and Q, or the *annotated* covering tree when the distinguishing actions are indicated. The idea is that in a system designed around the behavior of R, it will not matter which of P or Q is actually present.

The annotated covering tree (ACT) R is similar in nature to Larsen's weakest inner environment, in that it gives an indication of the most general environment which will interact in the same way with P and Q. An ACT, however, combines information about both the allowable environments and the process's behavior in one place. A weakest inner environment contains just the former type of information and requires a process to give the complete picture.

Now that we have seen what is required of an ACT, it is time to formalize some of the concepts.

Definition 5.3.1 Distinguishing Actions

The distinguishing actions of two processes P and Q are those actions belonging to the set (the da-set) defined by:

$$\text{da}(P,Q) \quad =_{def} \quad \{\lambda \mid P \xrightarrow{\lambda} \wedge Q \xrightarrow{\lambda} \star\} \quad \cup \quad \{\mu \mid P \xrightarrow{\mu} \star \wedge Q \xrightarrow{\mu}\}$$

□

The notation $P \xrightarrow{\lambda}$ is shorthand for $\exists P'. P \xrightarrow{\lambda} P'$ and $P \xrightarrow{\lambda} \star$ means that P rejects the λ communication. The definition can be summed up by saying that the da-set contains all of those actions that P accepts and Q rejects and those that P rejects and Q accepts.

[†]Singleton label-sets are written as plain labels to prevent proliferation of braces. Thus, R's distinguishing action set would be written {a, b, d}

Definition 5.3.2 Common Actions

The common actions of two processes P and Q are those actions belonging to the set (the ca-set) defined by:

$$\text{ca}(P,Q) =_{def} \{\lambda \mid P \xrightarrow{\lambda} \wedge Q \xrightarrow{\lambda}\} \cup \{\mu \mid P \xrightarrow{\mu} \star \wedge Q \xrightarrow{\mu} \star\}$$

The set can be further subdivided into an *initials* set, containing the λ's and a *refusals* set containing the μ's. These are respectively, the actions that the process is willing to accept and those that it rejects. □

Definition 5.3.3 Deterministic Annotated Covering Trees

The function ACT(P, Q) defined below constructs a process that can be expanded to generate an annotated covering tree for deterministic P and Q, with sortOf(P) = sortOf(Q) = L. This process is call ed an *Annotated Covering Process* (there can be more than one) and its synchronization tree is called an *Annotated Covering Tree.* The distinguishing actions are indicated by the dis operator.

$$\begin{aligned} P &\Leftarrow \sum_{\alpha_i \in A} \alpha_i P_i \\ Q &\Leftarrow \sum_{\beta_j \in B} \beta_j Q_j \\ \text{ACT}(P, Q) &= \sum_{\gamma_k \in A \cap B} \gamma_k \, \text{ACT}(P_k, Q_k) \ \ \underline{\text{dis}} \ \ A \uplus B \end{aligned}$$

□

Where $A \uplus B =_{def} (A \cup B) \setminus (A \cap B)$, *e.g.*, for $A = \{a, b, c\}$ and $B = \{a, b, d\}$, $A \uplus B = \{c, d\}$. Throughout the chapter, the da-set of an annotated covering tree P will usually be written as D_P and the ca-set as A_P.

The actions belonging to $A \cap B$ are those in the choice sum that are common to both processes and form the initial actions of ACT(P, Q). Combined with the refusals of the ACT—defined as pow$^-(L) \setminus (A \cup B)$)—they form the common actions set. The actions belonging to $A \uplus B$, on the other hand, are those that differ when the choice sums are overlayed, and hence belong to the da-set. All the actions understood by a process R (all subsets of the restricted powerset of its sort L) can be placed in one and only one of these sets. Thus initials$(R) \cup$ refusals$(R) \cup$ da$(R) = L$. Furthermore, any communication with an ACT will have one of four possible outcomes:

1. The communication is ignored since it does not belong to the sort of the ACT.
2. It is accepted and the ACT evolves.
3. It is rejected and the ACT evolves, perhaps to Δ_L.
4. The communication belongs to the set of distinguishing actions. It may be accepted or rejected depending on which of the original processes is actually present.

The first three outcomes are the usual possible results of an interaction as determined by the Dot Operator. The fourth is precisely the situation we wish to avoid, since what happens depends on which of the two original processes is actually present. We can therefore conclude that the two processes may be used interchangeably in any context that never attempts to communicate with their ACT's distinguishing actions. A predicate on contexts and ACT's can be defined that indicates if it is "safe" to place the ACT in the context. Safety is defined as the absence of communications with the distinguishing actions, so that the fourth outcome never arises. Here is an attempt at defining such a predicate:

Definition 5.3.4 Given an ACT generated by the annotated covering process U of sort S_U:

$$\begin{array}{rcl} U & \Leftarrow & U' \ \underline{\text{dis}} \ D_U \\ U' & \Leftarrow & \sum_{\alpha_i \in A_U} \alpha_i \, U_i \end{array}$$

It is safe to place U in a context formed from the core operators and another deterministic process V (with no distinguishing actions) iff:

$$
\begin{array}{lll}
(i) & \underline{\text{safe}}(\llbracket\,\rrbracket + V,\ U) \iff & \underline{\text{initials}}(V) \notin D_U \\
(ii) & \underline{\text{safe}}(\llbracket\,\rrbracket \oplus V,\ U) \iff & \underline{\text{initials}}(V) \notin D_U \\
(iii) & \underline{\text{safe}}(\llbracket\,\rrbracket\ [\rho],\ U) \iff & \rho \text{ preserves uniqueness of labels} \\
(iv) & \underline{\text{safe}}(\llbracket\,\rrbracket - \lambda,\ U) \iff & \text{Let } D'_U = \{x\backslash\lambda \mid x \in D_U\} \text{ in :} \\
 & & \quad (\not\exists \alpha \in A_U.\ \ \alpha\backslash\lambda \in D'_U) \\
 & & \quad \wedge\ \emptyset \notin D'_U \\
 & & \quad \wedge\ \underline{\text{safe}}\,(\llbracket\,\rrbracket - \lambda,\ U_i) \\
(v) & \underline{\text{safe}}(\llbracket\,\rrbracket \bullet V,\ U) \iff & \not\exists x \in D_U, y \in \underline{\text{initials}}(V).\ \ x \cap S_V = y \cap S_U \\
 & & \wedge \text{ The resultants are safe}
\end{array}
$$

Other contexts, notably action prefixing and recursion, are always safe. □

Discussion: The motivation for (i) and (ii) is straight forward. If any initial actions of V were to match any distinguishing actions of U, in the case of (i), unexpected nondeterminism could be introduced through law $[\gamma\ \oplus\ +]$. In case (ii), having one well defined possible outcome of a communication and another that is ill defined should be considered dangerous. Case (iii) is self explanatory—non-unique labels could lead to unexpected nondeterminism.

Abstraction is a little bit trickier. A member of a composite action may be removed making the result identical to one of the distinguishing actions, thus introducing potential nondeterminism. In the above definition, D'_U is the set of distinguishing actions with λ removed from each label-set in the set. If one of the initial actions α with λ removed belongs to this set, we know that the harmful nondeterminism may be introduced, so this eventuality is ruled out by the first clause. The second checks that no distinguishing actions are being hidden, which may or may not result in unknown nondeterministic moves. The last clause checks that resultants are also safe, because abstraction is a self-renewing operator.

As was discussed when the case was being made for the safety predicate, communications with distinguishing actions are undesirable because the outcome is unknown. The first clause in case (v) enforces this by making sure that none of the initials of V can synchronize with a distinguishing action of U. Composition, like abstraction, is a self renewing operator and so its resultants need checking for safety too.

Example: An ACP U defined by:

$$U \Leftarrow (\{\mathtt{a}\}\, U_1 + \{\mathtt{a}\ \ \mathtt{b}\}\, U_2)\ \ \underline{\text{dis}}\ \ \{\{\mathtt{b}\}, \{\mathtt{c}\}\}$$

can safely be placed in these contexts:

$$[\![\,]\!]\ [\mathtt{g/a}] \qquad \{\mathtt{a}\}\, P + [\![\,]\!] \qquad (\{\mathtt{a}\ \ \mathtt{b}\}\, Q + \{\mathtt{d}\}\, R) \bullet [\![\,]\!]$$

but not in these:

$[\![\,]\!]\ [\mathtt{c/a}]$	Distinguishing action same as an initial action.
$\{\mathtt{b}\}\, R + [\![\,]\!]$	b matches a d.a.
$\{\mathtt{c}\}\, Q \bullet [\![\,]\!]$	c communicates with a d.a.
$[\![\,]\!] - \mathtt{b}$	A d.a. is abstracted away
$[\![\,]\!] - \mathtt{a}$	Initial becomes same as a d.a.

□

In practice the safe predicate is not very useful. It can only be applied to a context having a single operator. Most useful contexts are combinations of several *e.g.*, composition and abstraction. Also, other processes in the context (the V's in the definition of safe) might be ACP's as well and not just deterministic processes. The rules given in the definition do not indicated under what conditions, for example, it is safe to place two ACP's in parallel. A more flexible notion of safety is needed.

5.3.2 Comparing Covering Trees

Clearly, to use ACT's effectively algebraic laws must be defined that allow them to be manipulated in the same fashion as normal processes. Central to the definition of such laws is a means of comparing ACT's. To do this, we notice that the method for generating an ACT from two processes as given in Definition 5.3.3 will yield only one of possibly many covering trees. Some will be more restrictive than others in that their distinguishing action sets will be larger. These will not fit safely into quite as many contexts, but are still perfectly valid. Such a notion of restrictivity suggests a pre-order on covering trees.

Definition 5.3.5 The Restrictiveness Pre-order $\sqsubseteq_R$

A covering tree U is said to be *more restrictive* (alternatively, *less general*) than another tree V with the same sort if $U \sqsubseteq_R V$. The preorder is defined as the intersection of ascending indexed relations $\cap_i \sqsubseteq_R^i$, where $U \sqsubseteq_R^0 V$ always holds and $U \sqsubseteq_R^{i+1} V$ iff:

$$\begin{array}{ll} i) & D_U \supseteq D_V \\ ii) & (\underline{\text{initials}}(V) \uplus \underline{\text{initials}}(U)) \subseteq D_U \\ iii) & U \xrightarrow{\lambda} U',\ V \xrightarrow{\lambda} V',\quad \lambda \notin D_U \ \text{ implies } \ U' \sqsubseteq_R^i V' \\ iv) & U \xrightarrow{\mu} \star,\quad \mu \notin D_U \ \text{ implies } \ V \xrightarrow{\mu} \star \end{array}$$

□

The indexed chain of relations allows recursive processes to be compared [Milner 80]. D_x represents the da-set of x. The resultants U' and V' in condition (*iii*) are unique since the trees are deterministic.

The first condition ensures that U has at least as many distinguishing actions as V. Condition (*ii*) makes sure that any differences in the common action sets belong to the da-set. This prevents the following from being true;

$$\{\mathtt{c}\}\, V' \ \underline{\text{dis}}\ \{\mathtt{a}\} \qquad \sqsubseteq_R \qquad (\{\mathtt{b}\}\, U' + \{\mathtt{c}\}\, U'')\ \underline{\text{dis}}\ \{\mathtt{a}\}$$

since the process on the right can accept a b communication, while the one on the left rejects it. A simple change makes the statement true:

$$\{\mathtt{c}\}\, V' \ \underline{\text{dis}}\ \{\mathtt{a},\, \mathtt{b}\} \qquad \sqsubseteq_R \qquad (\{\mathtt{b}\}\, U' + \{\mathtt{c}\}\, U'')\ \underline{\text{dis}}\ \{\mathtt{a}\}$$

Of particular interest are the maximal and minimal elements of this ordering. When the da-set of an ACT contains all communications possible with that process (*i.e.*, is equal to the powerset of the sort), any attempt at communication will have an unknown outcome. This is true because the only communications that will have a definite outcome are those that belong to the initial and refusal sets, which are empty. Such behavior is exactly that of the most non-deterministic process Ω, as discussed in Section 2.2.6. We can therefore infer that:

$$\Omega_L \ \sqsubseteq_R \ U \qquad [\sqsubseteq_R 1]$$

For all annotated covering trees U of sort L.

At the opposite extreme is the least restrictive covering tree for two processes P and Q, called the *Minimal Covering Tree* (MCT) The tree produced by the ACT function of Definition 5.3.3 is defined to be the MCT for P and Q. Thus,

$$U \sqsubseteq_R \text{MCT}(P, Q) \qquad [\sqsubseteq_R 2]$$

for all annotated covering trees U of P and Q.

The restrictiveness preorder can be used to define some laws about annotated covering trees:

Proposition 5.3.6

[$\underline{\text{dis}}$ 1]	$(\alpha P + Q)\ \underline{\text{dis}}\ \{\alpha\}$	$=_R$	$Q\ \underline{\text{dis}}\ \{\alpha\}$
[$\underline{\text{dis}}$ 2]	$Q\ \underline{\text{dis}}\ \{\alpha, \beta\}$	$\sqsubseteq_R$	$(\beta P + Q)\ \underline{\text{dis}}\ \{\alpha\}$
[$\underline{\text{dis}}$ $\emptyset$]	$P\ \underline{\text{dis}}\ \emptyset$	$=_R$	P
[$\underline{\text{dis}}$ $\underline{\text{dis}}$]	$(P\ \underline{\text{dis}}\ A)\ \underline{\text{dis}}\ B$	$=_R$	$P\ \underline{\text{dis}}\ A \cup B$

Proof:

[$\underline{\text{dis}}$ 1] Using Definition 5.3.5 and considering one direction of the equality: $(\alpha P + Q\ \underline{\text{dis}}\ \{\alpha\}) \sqsubseteq_R (Q\ \underline{\text{dis}}\ \{\alpha\})$. The distinguishing action sets are identical, so condition (i) is satisfied. The initial actions of the left side are $A = \{\alpha\} \cup \underline{\text{initials}}(Q)$ and those of the right side are $B = \underline{\text{initials}}(Q)$. Therefore, $A \uplus B = \{\alpha\} \subseteq \{\alpha\}$, fulfilling the requirements of condition (ii). The only actions common to both sides are $\underline{\text{initials}}(Q)$ and $Q \sqsubseteq_R Q$ from the reflexivity property of pre-orders, thus satisfy requirement (iii). The ordering holds in the other direction by a similar argument, so we can conclude that the two are equivalent. In practice, the left hand side is considered illegal since α belongs both to an initial action set and to a distinguishing actions set, violating the requirement that all actions belong to only one of the sets. This law shows that membership in the da-set is more influencial than membership in the refusal or initial sets.

[$\underline{\text{dis}}$ 2] Again, using Definition 5.3.5, the two distinguishing action sets are $A = \{\alpha, \beta\}$ and $B = \{\alpha\}$. Condition (i) requires that $A \supseteq B$, as is clearly true. To meet condition (ii), the initial action sets, $C = \underline{\text{initials}}(Q)$ and $D = \underline{\text{initials}}(Q) \cup \{\beta\}$ respectively, must satisfy $C \uplus D = \{\beta\} \subseteq A$, which they do. The last condition is met by the same reflexivity argument as used above.

[dis $\emptyset$] Clearly true. No distinguishing actions means that no restrictions are placed on the process's use.

[dis dis] Trivially true.

$\square$

Care must be taken when applying law [dis 2] from left to right that one does not move more actions from the da-set to the initial actions set than were present in the minimal covering tree. The result would be a meaningless tree that does not completely cover the two generating processes. For example, given two processes and their MCT:

$$\begin{aligned} P &\Leftarrow \{\mathtt{a}\}\, P_1 + \{\mathtt{b}\}\, P_2 + \{\mathtt{a}\ \ \mathtt{c}\}\, P_3 \\ Q &\Leftarrow \{\mathtt{b}\}\, Q_1 + \{\mathtt{c}\}\, Q_2 \\ \mathrm{MCT}(P,Q) &= [\{\mathtt{b}\}\ \mathrm{MCT}(P_2,Q_1)]\ \underline{\mathrm{dis}}\ \{\mathtt{a},\ \mathtt{c},\ \{\mathtt{a}\ \mathtt{c}\}\} \end{aligned}$$

Where a and c in the da-set stand for $\{\mathtt{a}\}$ and $\{\mathtt{c}\}$ respectively. Applying law [dis 2] produces:

$$\mathrm{MCT}(P,Q) \quad \sqsubseteq_R \quad (\{\mathtt{a}\}\, X + \{\mathtt{b}\}\, (\ldots))\ \underline{\mathrm{dis}}\ \{\mathtt{c},\ \{\mathtt{a}\ \mathtt{c}\}\}$$

Where X can be any process. This contradicts law [$\sqsubseteq_R$ 2], so we can conclude that the ACT on the right side of the ordering is not a legal covering tree of P and Q.

Having a means of comparing covering trees allows us to investigate their interactions with general contexts, as the next section shows.

5.3.3 Manipulating Covering Trees

Since the operators of the calculus can be expressed as operations on synchronization trees, one would expect to find similar operations on covering trees. Covering trees have a corresponding process representation in terms of the guarding, deterministic sum, recursion, and dis operators. It should therefore be possible to formulate laws along the lines of [• +] and [− +] that allow combinations of ACT's to be expressed as a single ACT. The laws will most likely take the form:

$$\text{op}(P_1 \ \underline{\text{dis}}\ D_1, \ldots , P_n \ \underline{\text{dis}}\ D_n) \quad =_R \quad \text{op}(Q_1, \ldots, Q_m) \ \underline{\text{dis}}\ D_Q$$

As you can see, propagating distinguishing action sets is a bottom-up procedure. Each argument to the operator must be in dis outermost form before the expression itself can be put in such a form. This is to be expected since the da-sets function as constraints on the environment and constraints are pushed outward as they interact with more and more of the environment.

Now we consider applying in turn each of the core operators to some representative covering trees. The laws that result should parallel the requirements first laid out in the definition of the safe predicate, since we are still trying to check for legal combinations of ACT's. Some sample ACT's are needed to construct the contexts from, so let:

$$\begin{array}{lllllll} U & \Leftarrow & U' \ \underline{\text{dis}} \ D_U & \qquad & U' & \Leftarrow & \sum_{\alpha_i \in A_U} \alpha_i U_i \\ V & \Leftarrow & V' \ \underline{\text{dis}} \ D_V & & V' & \Leftarrow & \sum_{\beta_j \in A_V} \beta_j V_j \end{array}$$

for some sets of actions A_s and D_s, $s = U, V$. The term $\sum_{\alpha_i \in A_s} \alpha_i P_i$ will often be abbreviated to $\sum_{A_s} \alpha_i P_i$ in the interests of saving space. Similarly, α_i will stand for $\forall \alpha_i \in A_s$, *i.e.*, all members of the set A_s. It will be obvious from the definitions above which set is begin referred to.

Law [dis +]

$$U + V \quad =_R \quad \Big(\sum_{\alpha_i \notin D_V} + \sum_{\beta_j \notin D_U} \Big) \ \underline{\text{dis}}\ D_U \cup D_V$$

Remember that dis is a special version of the choice operator that marks illegal branches. It is therefore commutative and associative with respect to +, so D_U and D_V can be grouped on the right and law [dis dis] applied to produce the union. Actions that have the possibility of introducing nondeterminism are relegated to the da-set by applying the absorption law [dis 1], hence the restrictions on the summations. If the restrictions cause the indexing set to become empty, an instance of the deadlock operator Δ with distinguishing actions results. Note that U and V must have the same sort by definition of the choice operator.

Law [$\underline{\text{dis}}\ \oplus$] Nondeterministic choice is similar to the deterministic version. There is no way of knowing whether U or V will actually be present, so we play it safe and distinguish actions that are the union of the two da-sets. Similarly, guards that have a counterpart in one of the da-sets must be removed, or the possibility of an unknown nondeterministic choice results.

$$\Big(\sum_{\alpha_i \notin D_V} \oplus \sum_{\beta_j \notin D_U} \Big)\ \underline{\text{dis}}\ D_U \cup D_V \quad \sqsubseteq_R \quad U \oplus V$$

Law [$\underline{\text{dis}}$ []] A renaming can be applied to a process as long as it preserves uniqueness of labels.

$$U\,[\rho] \quad = \quad U'\,[\rho]\ \underline{\text{dis}}\ D_U\,[\rho]$$

Law [$\underline{\text{dis}}\ \bullet$] The desire to prevent certain communications from taking place motivated the development of the $\underline{\text{dis}}$ operator in the first place. Since communication links are established by application of the Dot Operator, it is important to examine the interaction between the two.

Communicating with a distinguishing action does not automatically result in disaster. Other factors in the environment may prevent it from actually taking place *e.g.*, later composition with a deadlocked element. Distinguishing actions thus "absorb" attempts at communication, in much the same manner as matching actions in two composed deterministic sums unite to become one when law [$\bullet$ +] is applied to the expression. Here is the analogue to that law for the $\underline{\text{dis}}$ operator:

$$\begin{aligned} U \bullet V \quad &= \quad \sum_{\alpha_i \in A_U} \alpha_i U_i\ \underline{\text{dis}}\ D_U \quad \bullet \quad \sum_{\beta_j \in A_V} \beta_j V_j\ \underline{\text{dis}}\ D_V \\ &=_R \quad U' \bullet V' \quad \underline{\text{dis}} \quad \{ D_{U_i} \mid D_{U_i} \cap S_V = \emptyset \} \\ &\qquad\qquad \cup \{ D_{V_j} \mid D_{V_j} \cap S_U = \emptyset \} \\ &\qquad\qquad \cup \{ \alpha_i \cup D_{V_j} \mid D_{V_j} \cap S_U = \alpha_i \cap S_V \} \\ &\qquad\qquad \cup \{ \beta_j \cup D_{U_i} \mid D_{U_i} \cap S_V = \beta_j \cap S_U \} \\ &\qquad\qquad \cup \{ D_{U_i} \cup D_{V_j} \mid D_{U_i} \cap S_V = D_{V_j} \cap S_U \} \end{aligned}$$

Where $U' \bullet V'$ is a composition of deterministic choice sums expandable using [$\bullet$ +] (see the definitions of U and V above for an explanation of U' and V'). The rather large union that forms the new distinguishing actions set arises for much the same reasons as the three summations do in law [• +]. The first two include the independent distinguishing actions, the third and fourth handle possible interactions between initial actions and distinguishing actions and finally, the last generates interactions between D_U and D_V. Actions belonging to D_U, for example, would be excluded from the final set if they were not independent of S_V and did not synchronize with either a β_j or a $D_{V\,k}$. This is the way that the environment can elmininate the possibility of a communication with a distinguishing action and thus can remove it from the set.

Law [dis −] Abstraction is harder to deal with. The problem of communicating with distinguishing actions was alluded to above, but was put off by claiming that other parts of the context might block the communication. This argument no longer holds when the communications have been made internal by applying the abstraction operator, because if a distinguishing action is hidden, the process may nondeterministically decide to something completely unknown. The lack of knowledge is not so strange when we consider that the dis operator was developed to allow irrelevant portions of a behavior to be discarded. Following an internal distinguishing action, the process enters such a discarded portion and all knowledge of what it will do is lost. It can potentially engage in any sequence of actions drawn from the process's sort L. This type of behavior should seem familiar; it is exactly that of the most nondeterministic process Ω. The following law shows how Ω is introduced into an expression by abstracting away a distinguishing action.

$$\begin{aligned} U - \lambda &=_R \Big(\sum_{\alpha_i \in A'_U} \alpha_i U_i \Big) - \lambda \ \ \underline{\text{dis}} \ \ D'_U && \text{if } \lambda \notin D_U \\ &=_R \Omega \ \oplus \ \Big[\Big(\sum_{\alpha_i \in A'_U} \alpha_i U_i \Big) - \lambda \ \ \underline{\text{dis}} \ \ D'_U \backslash \emptyset \Big] && \text{if } \lambda \in D_U \end{aligned}$$

Where:

$$\begin{aligned} D'_U &= \{\gamma\backslash\lambda \mid \gamma \in D_U \text{ and } \lambda \neq \gamma\} \\ A'_U &= \{\alpha_i \mid \alpha_i \in A_U \text{ and } \alpha_i\backslash\lambda \notin D'_U\} \end{aligned}$$

The set D'_U is simply the da-set of U with λ removed from each element and any element that is equal to λ eliminated. A'_U is the subset of A_U whose elements when λ is removed do not belong to D'_U. Those elements that do belong to D'_U correspond to actions that could act nondeterministically with ones from the new da-set, so law [$\underline{\text{dis}}$ 1] is applied to absorb them. Note how Ω is introduced if a singleton label in D_U is hidden.

Examples:

$$\begin{aligned}
P &\Leftarrow (\{\mathsf{a}\}\, P1 + \{\mathsf{b}\}\, P)\ \underline{\text{dis}}\ \{\mathsf{c}\} \\
P1 &\Leftarrow (\{\mathsf{c}\}\, P + \{\mathsf{a}\}\, P1)\ \underline{\text{dis}}\ \{\mathsf{b}\}
\end{aligned}$$

$$\begin{aligned}
Q &\Leftarrow (\{\mathsf{c}\}\, Q1 + \{\mathsf{a}\}\, Q)\ \underline{\text{dis}}\ \{\mathsf{b}\} \\
Q1 &\Leftarrow (\{\mathsf{b}\}\, Q + \{\mathsf{a}\}\, Q1)\ \underline{\text{dis}}\ \{\mathsf{c}\}
\end{aligned}$$

$$\begin{aligned}
P \bullet Q &=_R \{\mathsf{a}\}\,(P1 \bullet Q)\ \underline{\text{dis}}\ \{\mathsf{b}, \mathsf{c}\} \\
P1 \bullet Q &=_R (\{\mathsf{a}\}\,(P1 \bullet Q) + \{\mathsf{c}\}\,(P \bullet Q1))\ \underline{\text{dis}}\ \{\mathsf{b}\} \\
P \bullet Q1 &=_R (\{\mathsf{a}\}\,(P1 \bullet Q1) + \{\mathsf{b}\}\,(P \bullet Q))\ \underline{\text{dis}}\ \{\mathsf{c}\} \\
P1 \bullet Q1 &=_R \{\mathsf{a}\}\,(P1 \bullet Q1)\ \underline{\text{dis}}\ \{\mathsf{b}, \mathsf{c}\}
\end{aligned}$$

$$\begin{aligned}
P - \mathsf{b} &=_R \big((P - \mathsf{b}) \oplus ((P - {}) + \{\mathsf{b}\}\,(P1 - \mathsf{b}))\big)\ \underline{\text{dis}}\ \{\mathsf{c}\} \\
&=_R \{\mathsf{a}\}\,(P1 - \mathsf{b})\ \underline{\text{dis}}\ \{\mathsf{c}\} \qquad \text{(by law [rec}_3\text{])} \\
P1 - \mathsf{b} &=_R \Omega \oplus (\{\mathsf{c}\}\,(P - \mathsf{b}) + \{\mathsf{a}\}\,(P1 - \mathsf{b}))
\end{aligned}$$

$$P + Q =_R (\{\mathsf{a}\}\, Q \oplus \{\mathsf{a}\}\, P1)\ \underline{\text{dis}}\ \{\mathsf{b}, \mathsf{c}\}$$

To sum up, covering trees are more generally useful than the partial specifications and satisfaction relations of the second section for the simple reason that they may be constructed for any two processes. A satisfaction relation, on the other hand, requires that the initial actions of one process must always be a subset of those of the other. This happens to be true of specification/implementation pairs, but might not be in general. In addition, a satisfaction relation provides no information on how two processes will interact with their environment, while a covering tree not only encapsulates this information, but also allows it to be manipulated algebraically.

Example: Here are two sample specification/implementation pairs:

$$\begin{aligned}
S_1 &\Leftarrow \{\texttt{in}\}\,\{\texttt{b}\}\,\{\texttt{clk}\}\,S_1\\
I_1 &\Leftarrow \{\texttt{in}\}\,\{\texttt{b}\}\,\{\texttt{clk}\}\,I_1 + \{\texttt{b}\}\,\{\texttt{in}\}\,\{\texttt{clk}\}\,I_1\\
S_2 &\Leftarrow \{\texttt{b}\}\,\{\texttt{out}\}\,\{\texttt{clk}\}\,S_2\\
I_2 &\Leftarrow \{\texttt{b}\}\,\{\texttt{out}\}\,\{\texttt{clk}\}\,I_2 + \{\texttt{out}\}\,\{\texttt{b}\}\,\{\texttt{clk}\}\,I_2
\end{aligned}$$

Both specifications say that the parts will perform two actions and wait for a clock event. The implementations allow the two events to happen in either order. Using Definition 5.2.1 we can say that:

$$I_1 \ \underline{\text{sat}} \ S_1 \qquad \text{and} \qquad I_2 \ \underline{\text{sat}} \ S_2$$

but it is clear that when S_1 is composed with S_2 the result will be quite different from the result of composing I_1 with I_2. The extra branches of the implementations interact to produce behavior that can not be deduced from their specifications.

Let us now see what happens when we compose the covering trees of the two pairs. The trees are calculated using the ACT function with these results:

$$\begin{aligned}
T_1 &= \text{ACT}(S_1, I_1) & T_2 &= \text{ACT}(S_2, I_2)\\
&= \{\texttt{in}\}\,R_1 \ \underline{\text{dis}}\ \{\texttt{b}\} & &= \{\texttt{b}\}\,R_2 \ \underline{\text{dis}}\ \{\texttt{out}\}\\
R_1 &= \{\texttt{b}\}\,\{\texttt{clk}\}\,T_1 \ \underline{\text{dis}}\ \{\texttt{in}\} & R_2 &= \{\texttt{out}\}\,\{\texttt{clk}\}\,T_2 \ \underline{\text{dis}}\ \{\texttt{b}\}
\end{aligned}$$

The composition is expanded using law [$\underline{\text{dis}}$ $\bullet$] with the following results:

$$\begin{aligned}
T_1 \bullet T_2 &= \{\texttt{in}\}\,(R_1 \bullet T_2)\ \underline{\text{dis}}\ \{\texttt{b, out}\}\\
R_1 \bullet T_2 &= \{\texttt{b}\}\,(\{\texttt{clk}\}\,T_1 \bullet R_2)\ \underline{\text{dis}}\ \{\texttt{in, out}\}\\
\{\texttt{clk}\}\,T_1 \bullet R_2 &= \{\texttt{out}\}\,\{\texttt{clk}\}\,(T_1 \bullet T_2)\ \underline{\text{dis}}\ \{\texttt{b}\}
\end{aligned}$$

Note how the da-sets of each resultant is the union of the sets of the two constituent processes. The distinguishing actions of the top level, for example, say that no communications may be made with `b` nor `out` until after the `in` has happened. Composing the two trees with a process that forces `in`, `b`, and `out` to occur in sequence eliminates the da-sets:

$$\begin{aligned}
C &\Leftarrow \{\texttt{in}\}\,\{\texttt{b}\}\,\{\texttt{out}\}\,C\\
C \bullet T_1 \bullet T_2 &= \{\texttt{in}\}\,\{\texttt{b}\}\,\{\texttt{out}\}\,\{\texttt{clk}\}\,(C \bullet T_1 \bullet T_2)
\end{aligned}$$

The $\{\texttt{in}\}$ guard of C causes the distinguishing actions of $T_1 \bullet T_2$ to be dropped, since they belong to the sort of C and are thus prevented from happening. Similarly, the da-sets of the succeeding resultants are also dropped. C has constrained the three events to act in sequence and in doing so eliminated the possible alternative orderings that necessitated the distinguishing actions. It is interesting to note that if C were composed with either element of the first specification/implementation pairs and another from the second, precisely the same sequence would result. The ACT need not be computed at all! This observation forms the basis for an alternative representation of constraints as discussed in the next section.

5.3.4 Constraints as Contexts

Covering trees are conceptually a bit messy, since the description of a process's behavior is intermingled with information on how it should interact with the environment. A cleaner approach is to separate the two into a purely behavioral description and a constraint upon its use. In other words, the annotated covering tree is replaced by an unannotated process and a context that captures the constraints on the use of the process. An parallel can then be drawn with Larsen's context dependent equivalence.

A system is typically constructed by fitting together various blocks in order to obtain the required behavior. The connections are determined by applications of the renaming and abstraction operators, but the actual composition is done with the Dot Operator. Since this type of context is so common, it is convenient to give it a special name:

Definition 5.3.7 Compositional Contexts

A context is said to be *compositional* if it is of the form:

$$P \bullet [\![\,]\!]$$

for some process P. □

Each block will usually be defined as a pair consisting of a specification—which indicates how the block looks to the rest of the system—and an implementation in terms of less abstract blocks. By applying Definition 5.3.3, a single covering tree can be defined for the pair that will govern the block's use. Other blocks in the system may also be represented by ACT's, so the overall behavior can be derived using law [$\underline{\text{dis}}$ •]. Since the various trees interact, one would expect there to be certain combinations that could prevent any communication with the distinguishing actions of a particular tree *i.e.*, when [$\underline{\text{dis}}$ •] is applied the da-set of the resulting expression is empty. This amounts to saying that in this particular (compositional) context the two generating processes are completely interchangeable.

To illustrate this, consider a particular block that has a specification[†] S and an implementation I. It's behavior as far as the outside world were concerned would then be described by the covering tree $U = \text{ACT}(S, I)$. U would be of the form $U = U' \ \underline{\text{dis}}\ D_U$ for some (possibly empty) distinguishing actions set D_U.

Now suppose that another tree $C \Leftarrow C' \ \underline{\text{dis}}\ D_C$ is constructed from U as follows. (1) A node is created for each node in U. (2) The initial actions of the node are the same as those of the corresponding node in U. (3) The refusal set, however is the union of both the refusal and the distinguishing actions sets of U. In other words, C will be a tree that refuses all of U's distinguishing actions and has none of its own.

Law [$\underline{\text{dis}}$ •] can be invoked to see what happens when C is placed in parallel with U. Two things need to be calculated: the body of the resulting process and its distinguishing actions set. First the da-set D_V is calculated using the

[†]Note that unlike partial specifications, these may have actions that unavailable to the implementation. Why this capability would be used is unclear, but it's available all the same.

following identities:

$$S_U = S_C$$
$$D_C = \emptyset$$
$$\underline{\text{initials}}(U) = \underline{\text{initials}}(C)$$
$$\alpha_i \in \underline{\text{initials}}(U)$$

Applying law [$\underline{\text{dis}}$ $\bullet$] to $U \bullet C$ leads to these equations for the new da-set:

$$\begin{array}{lcl}
\{D_{U_i} \mid D_{U_i} \cap S_C = \emptyset\} & = & \emptyset \\
\{D_{C_j} \mid D_{C_j} \cap S_U = \emptyset\} & = & \emptyset \\
\{\alpha_i \cup D_{C_j} \mid D_{C_j} \cap S_U = \alpha_i \cap S_C\} & = & \emptyset \\
\{\alpha_i \cup D_{U_j} \mid D_{U_j} \cap S_C = \alpha_i \cap S_U\} & = & \emptyset \\
\{D_{U_i} \cup D_{C_j} \mid D_{U_i} \cap S_C = D_{C_j} \cap S_U\} & = & \emptyset
\end{array}$$

by substitution and elementary algebra. D_V is the union of of these sets, which is simply the empty set. This is to be expected since C was defined so as to reject all of U's distinguishing actions. The bodies of the two trees (U' and C') are combined as usual using law [$\bullet$ +]. Thus the covering tree V for the complete system is given by:

$$V \quad = \quad U \bullet C \quad = \quad (U' \bullet C')\ \underline{\text{dis}}\ \emptyset \quad = \quad U' \bullet C'$$

A similar result is obtained for every resultant of U' and C'. We can conclude that C prevents all communications with the distinguishing actions and that:

$$C \bullet S = C \bullet I$$

C has constrained S and I so that they function in exactly the same way.

U was defined as the minimal (least restrictive) covering tree of S and I, so the context $C \bullet [\![\,]\!]$ is, to use Larsen's terminology, the weakest inner environment that makes S and I equivalent. The notion of environment has not been developed here, so from now on we will call this context the *weakest safe context*, and C the *constraint process.*

Definition 5.3.8 Weakest Safe Context.

The compositional context that produces the same behavior when filled by either of two deterministic processes P or Q:

$$C \bullet P \quad = \quad C \bullet Q$$

is called the *weakest safe context* if the *constraint process* C is defined as the process that results when the da-set of the annotated covering tree of P and Q is relegated to the refusal set. □

But what will happen if we wish to place the processes in another context? The part described by the processes will typically be connected to other parts in order to implement some system. The behaviors of the other parts unite to form a context in which the part will be placed. It is highly unlikely that this context will be identical to the weakest safe context, so how can we tell if it is also safe? If it is will it then be related in some way to the weakest safe context? The rest of the section develops a way of comparing contexts and shows how it can be used to check if two processes are interchangeable in a particular context.

Expanding on the previous example, suppose that we wish to use the S and I processes as part of a larger system. The rest of the system appears as a compositional context of the form $\mathcal{D} = R \bullet [\![\,]\!]$. R is a process (deterministic and with no distinguishing actions) that represents the aggregate behavior of the other blocks. We have already calculated the weakest safe context $\mathcal{C}[\![\,]\!] = C \bullet [\![\,]\!]$ of S and I. The question is, can the information contained in C be used to check if it is safe to place S and I interchangeably in $\mathcal{D}$? Let us see what happens if the equations are expanded:

$$\begin{array}{ll} & \mathcal{C}[\![S]\!] = \mathcal{C}[\![I]\!] \\ \text{so} & S \bullet C = I \bullet C \\ \text{and} & R \bullet [\![S \bullet C]\!] = R \bullet [\![I \bullet C]\!] \\ \text{thus} & (R \bullet C) \bullet S = (R \bullet C) \bullet I \\ \text{implies?} & R \bullet S = R \bullet I \end{array}$$

The only way for the final step to hold is if

$$R \bullet C = R \tag{5.2}$$

In other words, the context must be at least as restrictive as the weakest safe context. Equation 5.2 bears a striking resemblance to Milne's definition of satisfaction as discussed in Section 5.2. His relation was considered inadequate because it failed to take account of the effects of the environment on the processes being compared. This objection does not apply here, because the environments themselves are being compared. Thus $R \bullet C = R$ could be written as R sat C.

Another interesting observation is that Equation 5.2 corresponds to the following relation on (deterministic) C and R:

$$\begin{aligned} R \mathcal{R} C \iff & \\ & R \xrightarrow{\lambda} R' \text{ implies } \exists C'. C \xrightarrow{\lambda} C' \quad \wedge \quad R' \mathcal{R} C' \\ & C \xrightarrow{\lambda} \star \text{ implies } R \xrightarrow{\lambda} \star \end{aligned}$$

Any action that R can accept, must also be accepted by C and so on for their resultants. If C rejects an action, R must reject the same action or $R \bullet C$ will not behave like R.

$\mathcal{R}$ is just half of the well known *bisimulation* ordering [Milner 83], called the ***simulation ordering.*** It is this observation that gives a link to Larsen's work. His so-called Main Theorem states that if two processes are equivalent in an environment E, then they will also be equivalent in another environment F iff F simulates E. The environments E and F correspond to the contexts $\mathcal{C}$ and $\mathcal{D}$ respectively, and $R \mathcal{R} C$ is the simulation ordering.

Definition 5.3.9 The Simulation Ordering $\leq$.

A process R of sort S_R is said to *simulate* another process C of sort S_C iff these conditions hold:

$$\begin{aligned} R \leq C \iff & \\ 1) \quad & S_R \supseteq S_C \\ 2) \quad & R \xrightarrow{\lambda \in A} R' \text{ implies } \exists C'. C \xrightarrow{\lambda} C' \quad \wedge \quad R' \leq C' \\ 3) \quad & C \xrightarrow{\lambda \in A} \star \text{ implies } R \xrightarrow{\lambda} \star \\ 4) \quad & R \xrightarrow{\mu \notin A} R' \text{ implies } R' \leq C \end{aligned}$$

Where A is the set of actions that both processes have in common:

$$A = \{x \mid x \in \text{pow}^{-}(S_R) \quad \wedge \quad x \cap S_C \neq \emptyset\} \quad \cup \quad \text{pow}^{-}(S_U)$$

□

Requiring that the sort of C be a subset of the sort of R means that R is allowed to indulge in actions that might not be available to C. It must, however, always have the potential to accept every one that C can. Condition (2) takes this a step further by saying that the initials of R that intersect the sort of C must be matched by a corresponding action in the initials of C. Condition (3) ensures that all actions refused by C are also refused by R and the last condition handles independent actions by R. Note that the first condition makes the simulation ordering more general than the satisfaction relation since the latter requires that the two sorts be identical.

Example: P will simulate Q when they are defined as:

$$\begin{aligned} P &\Leftarrow \{\mathtt{t}\}\, P + \{\mathtt{a}\ \mathtt{t}\}\, \{\mathtt{b}\ \mathtt{t}\}\, P \\ Q &\Leftarrow \{\mathtt{a}\}\, \{\mathtt{b}\}\, Q + \{\mathtt{b}\}\, \{\mathtt{a}\}\, Q \end{aligned}$$

The first condition of the definition of the simulation ordering hold becasue $S_P = \{\mathtt{t},\ \mathtt{a},\ \mathtt{b}\}$ is a superset of $S_Q = \{\mathtt{a},\ \mathtt{b}\}$. The actions that the two have in common are

$$A = \{\mathtt{a},\ \mathtt{b},\ \{\mathtt{a}\ \mathtt{b}\},\ \{\mathtt{a}\ \mathtt{t}\},\ \{\mathtt{b}\ \mathtt{t}\},\ \{\mathtt{a}\ \mathtt{b}\ \mathtt{t}\}\}$$

elicit the same response from both, fulfilling the second condition. P rejects the same members of A as Q does and $P \leq Q$ after the independent $\mathtt{t}$ action happens. □

One would expect that if $P \leq Q$ and $Q \leq P$ then $P = Q$. This is the bisimulation equivalence and has the same properties as the intersection-of-ascending-chains equivalence used throughout the thesis (see Section 2.6).

Theorem 5.3.10 Bisimulation

$$P \leq Q \quad \wedge \quad Q \leq P \quad \text{implies} \quad P = Q$$

Proof: From Definition 5.3.9 condition (1), we know that $S_P \subseteq S_Q$ and $S_Q \subseteq S_P$, which implies that $S_P = S_Q$. Condition (4) can then be eliminated since neither side has any independent actions. □

The simulation ordering was defined in order to check if a context could replace a weakest safe context and still have the generating processes act identically. The following theorem shows exactly how the ordering is used to check a particular context.

Theorem 5.3.11 Compositional Context Replacement

Given two deterministic processes P and Q $(P \neq Q)$ that behave identically in a weakest safe context $\mathcal{C}[\![\,]\!] = C \bullet [\![\,]\!]$ for some deterministic process C with no distinguishing actions. Another context $\mathcal{D}[\![\,]\!] = R \bullet [\![\,]\!]$ for some deterministic, undistinguished process R, will preserve this equivalence iff R simulates C:

$$C \bullet P = C \bullet Q \text{ implies } R \bullet P = R \bullet Q \iff R \leq C$$

Proof: $(\Rightarrow)$ Assume that the left hand side is true, *i.e.,* that P and Q are equivalent in both contexts. Now if $R \not\leq C$, one or more of the conditions of Definition 5.3.9 will not be met. By case analysis on these conditions:

1. Assume that $S_R \not\supseteq S_C$. Then either $S_R \cap S_C = \emptyset$, or $S_R \subset S_C$. Since C is the weakest covering process of P and Q, its sort must be the same as S_P and S_Q. Therefore, if S_R is completely independent of S_C, it can interact with neither P nor Q. The only way for $R \bullet P = R \bullet Q$ to be true then is if $P = Q$, which leads to a contradiction.

   If $S_R \subset S_C$, then there must be a subset of S_C that is independent of S_R. Consequently, P and Q could be construct with branches labelled by actions from this set such that C prevents these actions from happening, but $\mathcal{D}$ cannot since they are independent as far as it is concerned. $R \bullet P$ would therefore not be equal to $R \bullet Q$, which contradicts the assumption.

2. Suppose that $R \xrightarrow{\lambda} R'$ but $C \xrightarrow{\lambda} \star$. A P and a Q could then be constructed such that $P \xrightarrow{\lambda} P'$, $Q \xrightarrow{\lambda} Q'$ and $P' \neq Q'$. The two processes would behave identically when composed with C, but would differ when composed with R. This contradicts the assumption that $R \bullet P = R \bullet Q$.

3. A similar argument is used to show that if $C \xrightarrow{\lambda} \star$ but $R \xrightarrow{\lambda} R'$ a contradiction is reached.

4. Actions independent of S_C are also independent of S_P and S_Q since C is the weakest covering process and has the same sort. Suppose $R \xrightarrow{\mu} R'$

for independent action μ and $R' \bullet P \neq R' \bullet Q$. But the assumption that $R \bullet P = R \bullet Q$ holds for all resultants, so a contradiction is reached.

($\Leftarrow$) For this case, assume that $C \bullet P = C \bullet Q$, $R \leq C$ and $R \bullet P \neq R \bullet Q$. If the inequality is true, there must be a λ such that:

$$\begin{array}{lcl} (R{\bullet}P) \xrightarrow{\lambda} & & (R{\bullet}P) \xrightarrow{\lambda} \star \\ & \text{or} & \\ (R{\bullet}Q) \xrightarrow{\lambda} \star & & (R{\bullet}Q) \xrightarrow{\lambda} \end{array}$$

If λ is independent of S_C, it cannot interact with P and Q and these assertions are false. On the other hand, if $\lambda \cap S_C \neq \emptyset$ then, by condition (2) of the definition of the simulation ordering, C must accept the same action, leading to:

$$\begin{array}{lcl} (C{\bullet}P) \xrightarrow{\lambda} & & (C{\bullet}P) \xrightarrow{\lambda} \star \\ & \text{or} & \\ (C{\bullet}Q) \xrightarrow{\lambda} \star & & (C{\bullet}Q) \xrightarrow{\lambda} \end{array}$$

This means that $C \bullet P \neq C \bullet Q$, which leads to a contradiction. □

Using this theorem and taking advantage of the broadcasting nature of guards in the calculus, a simple notion of many-hole contexts can be developed. Consider two interchangeable pairs and their constraint processes:

$$\begin{aligned} C_1 \bullet P_1 &= C_1 \bullet Q_1 \\ C_2 \bullet P_2 &= C_2 \bullet Q_2 \end{aligned}$$

Composing the two results in:

$$C_1 \bullet C_2 \bullet (P_1 \text{ or } Q_1) \bullet (P_2 \text{ or } Q_2) \quad = \quad \mathcal{C}[\![(P_1 \text{ or } Q_1), (P_2 \text{ or } Q_2)]\!]$$

Any mixture of P_1 or Q_1 with P_2 or Q_2 will be indistinguishable in the context $\mathcal{C}$. C_1 and C_2 are deterministic processes, so their composition may simplify considerably when expanded using law [$\bullet$ +]. In addition, if both $P_2 \leq C_1$ and $Q_2 \leq C_1$ then C_1 can be removed altogether, since (P_2 or Q_2) will correctly limit (P_1 or Q_1) due to the broadcast nature of guards. Similarly, C_2 can be removed if $P_1 \leq C_2$ and $Q_1 \leq C_2$.

This result illustrates a major difference between Larsen's work and the constraint method presented here. His context dependent equivalence, although applicable to contexts constructed from arbitrary operators, cannot cope with multiple hole contexts. We, on the other hand, get this ability "for free" from properties of the calculus at the expense of only being able to use it in compositional contexts. These are by far the most prevalent form in hardware applications, so it should not be regarded as a serious deficiency.

5.3.5 Discussion

So what are the advantages and disadvantages of the covering tree and compositional context approaches? Covering trees require the use of a special operator—the dis operator—for indicating dangerous actions, but are almost as compact as the simpler of the two generating processes. Moreover, they may be placed in any context provided that the laws governing the dis operator are obeyed.

Compositional safe contexts, on the other hand, can be constructed using the core operators of the calculus alone. In addition, the constraint process can be separated into smaller, more obvious, constraints that are combined with the Dot Operator in much the same way as one would formulate a conjunction of facts about safe contexts. On the minus side, weakest safe contexts tend to be larger than comparable covering trees because information about the generating processes must be included in the constraint process.

Covering trees may be more amenable to machine manipulation, whereas constraint processes are easier for a human to understand, particularly if presented in a "conjunction" of smaller constraints form. Since covering trees can be converted into safe contexts quite easily, there is a great deal of flexibility in the choice of which approach to use for an application.

5.4 An Example

Two techniques for specifying constraints on processes were discussed in the last section. By way of clarifying their use, we now apply each in turn to a simple example, our old favorite the latch.

Although the latch is a very simple circuit, the presence of feedback opens the door to some unexpected behavior, as we saw during the experiments conducted in Section 3.3. A specification for the required behavior is very simple, so it will be instructive to see under what conditions it really does produce the same behavior as the implementation. Here is the specification for the latch as it was first presented on page 85:

part *LATCH* {s, r, q, qb} ⟨

$$\begin{aligned} \mathit{Reset} &\Leftarrow \{\texttt{s}\triangleright 1\}\{\texttt{qb}\triangleleft 0\}\{\texttt{q}\triangleleft 1\}\,\mathit{Set} + \{\texttt{r}\triangleright x\}\,\mathit{Reset} + \{\texttt{s}\triangleright 0\}\,\mathit{Reset} \\ \mathit{Set} &\Leftarrow \{\texttt{r}\triangleright 1\}\{\texttt{q}\triangleleft 0\}\{\texttt{qb}\triangleleft 1\}\,\mathit{Reset} + \{\texttt{s}\triangleright x\}\,\mathit{Set} + \{\texttt{r}\triangleright 0\}\,\mathit{Set} \end{aligned}$$

⟩

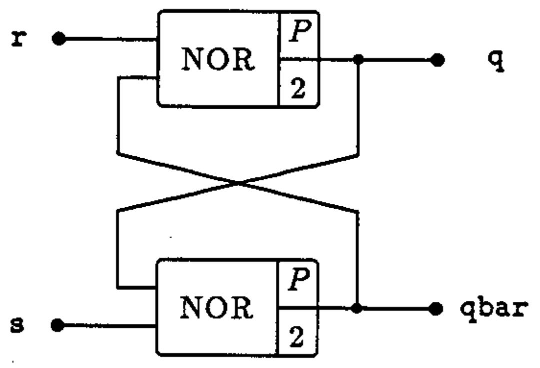


Figure 5–3: Latch implementation

The latch was constructed from two cross-coupled NOR gates as illustrated in Figure 5–3. Each NOR gates can be described by a generic box with an unspecified delay.

$$\begin{aligned} \mathit{NA}(r,qb,q) &\Leftarrow \mathit{Box2\_1}\ [\mathrm{NOR}/f, \texttt{r}/\texttt{in}_1,\ \texttt{qb}/\texttt{in}_2,\ \texttt{q}/\texttt{out}] \\ &= \mathrm{ANY}(\texttt{r}\triangleright,\ \texttt{q}\triangleright b) \\ &\qquad (\text{if } \mathrm{NOR}(r,qb) \neq q \text{ then } \{\texttt{q}\triangleleft\mathrm{NOR}(r,qb)\}) \\ &\qquad \mathit{NA}(r,qb,\mathrm{NOR}(r,qb)) \\ \mathit{NB}(q,s,qb) &\Leftarrow \mathit{Box2\_1}\ [\mathrm{NOR}/f, \texttt{q}/\texttt{in}_1,\ \texttt{s}/\texttt{in}_2,\ \texttt{qb}/\texttt{out}] \\ &= \mathrm{ANY}(\texttt{s}\triangleright,\ \texttt{q}\triangleright) \\ &\qquad (\text{if } \mathrm{NOR}(s,q) \neq qb \text{ then } \{\texttt{qb}\triangleleft\mathrm{NOR}(s,q)\}) \\ &\qquad \mathit{NA}(s,q,\mathrm{NOR}(s,q)) \end{aligned}$$

Both boxes have three state variables corresponding to the current states of the two input lines and the state of the output line. The composite system will therefore have six state variables, two of which (the duplicate q and qb) are redundant. The variable storing the value last output by *NA* on q should always be equal to the variable storing the value last input by *NB* on its q port and similarly for the qb channel. The redundent state variables are eliminated by defining an appropriate implementation name:

$$IMP(r,s,q,qb) \;\Leftarrow\; NA(r,qb,q) \bullet NB(q,s,qb)$$

There remain, however, four state variables in the implementation that must somehow be matched up with the two states of the specification (*Set* and *Reset*). This can be done by indicating that when q is high (and qb is low), the implementation is has been set and when it is low the implementation has been reset. This suggests the following mapping:

$$\begin{aligned} IMP(r,s,1,0) &\mapsto Set \\ IMP(r,s,0,1) &\mapsto Reset \end{aligned}$$

Now let us try applying Definition 5.3.3 to generate an Minimal Covering Process for the implementation and specification processes. The procedure is a follows: (1) Starting with the root nodes (initial states) of the two trees, each matching node is considered as a pair and is assigned a number in **bold face**. The pair is written $\langle \mathcal{L}, \mathcal{R} \rangle$, where $\mathcal{L}$ refers to the left member and $\mathcal{R}$ to the right, as inspired by the notation used by Prasad for bisimulation proofs [Prasad 84]. (2) Each half of the pair is written as a deterministic choice sum and placed side by side. The resultants, if they are not state names, are assigned unique ones by indexing the main state name. (3) The guards in the choice sums are compared. Any that match become part of the covering tree's choice sum, whilst those that do not are relegated to the da-set. The resulting equation describes a node in the covering tree and is highlighted with a box. Its resultants are state numbers corresponding to other state pairs in the derivation. Many state pairs will be eliminated from consideration, since actions leading up to them will belong to the da-set of the previous node.

We begin by assuming that the specification starts in the *Set* state and the implementation in state *IMP*(0, 0, 1, 0) (remember, the q line high implies that the latch is set).

(1) $\langle Set,\quad IMP(0,0,1,0)\rangle$

$$\mathcal{L} = \{\mathrm{r}\triangleright 1\}\, Set_1 + \{\mathrm{s}\triangleright x\}\, Set + \{\mathrm{r}\triangleright 0\}\, Set$$

$$\mathcal{R} = \{\mathrm{r}\triangleright\}\, IMP_1(r,0,0,1) + \{\mathrm{s}\triangleright\}\, IMP_2(0,s,1,0) + \{\mathrm{s}\triangleright\ \mathrm{r}\triangleright\}\, IMP_3(r,s,1,0)$$

$$\boxed{\{\mathrm{r}\triangleright 1\}\,(2) + \{\mathrm{s}\triangleright\}\,(5) + \{\mathrm{r}\triangleright 0\}\,(9)\quad \underline{\text{dis}}\quad \{\{\mathrm{r}\triangleright\ \mathrm{s}\triangleright\}\}}$$

(2) $\langle Set_1,\quad IMP_1(1,0,1,0)\rangle$

$$\mathcal{L} = \{\mathrm{q}\triangleleft 0\}\, Set_2$$

$$\begin{aligned}\mathcal{R} &= \left[(\text{if } \mathrm{NOR}(1,0) \neq 1 \text{ then } \{\mathrm{q}\triangleleft 0\})NA(1,0,0)\right]\\ &\quad \bullet (\{\mathrm{s}\triangleright\}\,(\ldots) + \{\mathrm{q}\triangleright\}\,(\ldots) + \{\mathrm{q}\triangleright\ \ \mathrm{s}\triangleright\}\,(\ldots))\\ &= \{\mathrm{q}\triangleleft 0\}\, IMP_4 + \{\mathrm{s}\triangleright\}\,(\ldots) + \{\mathrm{q}\triangleright 0\ \ \mathrm{s}\triangleright\}\,(\ldots)\end{aligned}$$

$$\boxed{\{\mathrm{q}\triangleleft 0\}\,(3)\quad \underline{\text{dis}}\quad \{\{\mathrm{s}\triangleright\}, \{\mathrm{s}\triangleright\ \ \mathrm{q}\triangleright 0\}\}}$$

(3) $\langle Set_2,\quad IMP_4(1,0,0,0)\rangle$

$$\mathcal{L} = \{\mathrm{qb}\triangleleft 1\}\, Reset$$

$$\begin{aligned}\mathcal{R} &= (\{\mathrm{r}\triangleright\}\,(\ldots) + \{\mathrm{qb}\triangleright\}\,(\ldots) + \{\mathrm{r}\triangleright\ \ \mathrm{qb}\triangleright\}\,(\ldots))\\ &\quad \bullet (\text{if } \mathrm{NOR}(0,0) \neq 0 \text{ then } \{\mathrm{qb}\triangleleft 1\})NB(0,0,1)\\ &= \{\mathrm{r}\triangleright\}\,(\ldots) + \{\mathrm{qb}\triangleright 0\}\, IMP_5 + \{\mathrm{r}\triangleright\ \mathrm{qb}\triangleright 0\}\,(\ldots)\end{aligned}$$

$$\boxed{\{\mathrm{qb}\triangleleft 1\}\,(4)\quad \underline{\text{dis}}\quad \{\{\mathrm{r}\triangleright\}, \{\mathrm{r}\triangleright\ \mathrm{qb}\triangleleft 1\}\}}$$

(4) $\langle Reset,\ IMP_5(1,0,0,1)\rangle$

$$\begin{aligned}\mathcal{R} &= NB(0,0,1) \bullet (\text{if } \mathrm{NOR}(1,1) \neq 0 \text{ then } \{\mathrm{q}\triangleleft 0\})NA(1,1,0)\\ &= NA(1,1,0) \bullet NB(0,0,1)\\ &= IMP(1,0,0,1)\end{aligned}$$

$$\boxed{\text{Goes to state (6) of the Reset comparison}}$$

(5) $\langle Set,\ IMP_2(0,s,1,0)\rangle$

$$\begin{aligned}\mathcal{R} &= NA(0,0,1) \bullet (\text{if } \mathrm{NOR}(s,1) \neq 0 \text{ then } \{\mathrm{qb}\triangleleft 0\})NB(s,1,0)\\ &= IMP(0,s,1,0)\end{aligned}$$

$$\boxed{\text{if } s = 0 \text{ then (1) else if } s = 1 \text{ then (6)}}$$

(6) $\langle Set,\ IMP(0,1,1,0)\rangle$

(Similar to State **(1)**)

$$\boxed{\{\mathtt{r}\triangleright 1\}\ (7) + \{\mathtt{s}\triangleright\}\ (5) + \{\mathtt{r}\triangleright 0\}\ (9)\quad \underline{\text{dis}}\quad \{\{\mathtt{r}\triangleright\ \mathtt{s}\triangleright\}\}}$$

(7) $\langle Set_1,\quad IMP_1(1,1,1,0)\rangle$

$$\begin{aligned}\mathcal{L} &= \{\mathtt{q}\triangleleft 0\}\ Set_2\\ \mathcal{R} &= \big[(\textbf{if } \textsc{nor}(1,0) \neq 1 \textbf{ then } \{\mathtt{q}\triangleleft 0\})NA(1,0,0)\big]\\ &\quad \bullet \big[\{\mathtt{s}\triangleright\}\ (\ldots) + \{\mathtt{q}\triangleright\}\ (\ldots) + \{\mathtt{q}\triangleright\quad \mathtt{s}\triangleright\}\ (\ldots)\big]\\ &= \{\mathtt{q}\triangleleft 0\}\ IMP_4(1,1,0,0) + \{\mathtt{s}\triangleright\}\ (\ldots) + \{\mathtt{q}\triangleright 0\ \mathtt{s}\triangleright\}\ (\ldots)\end{aligned}$$

$$\boxed{\{\mathtt{q}\triangleleft 0\}\ (8)\quad \underline{\text{dis}}\quad \{\{\mathtt{s}\triangleright\}, \{\mathtt{s}\triangleright\ \mathtt{q}\triangleright 0\}\}}$$

(8) $\langle Set_2,\ IMP_4(1,1,0,0)\rangle$

$$\begin{aligned}\mathcal{L} &= \{\mathtt{qb}\triangleleft 1\}\ Reset\\ \mathcal{R} &= NA(1,0,0) \bullet (\textbf{if } \textsc{nor}(0,1) \neq 0 \textbf{ then } \{\mathtt{qb}\triangleleft 0\}\ )NB(1,0,0)\\ &= NA(1,0,0) \bullet NB(1,0,0)\\ &= IMP(1,1,0,0)\end{aligned}$$

There is no match! *IMP* can not accept a qb◁1 action, so this branch is illegal. The guard {q◁0} leading to it in State **(7)** must be added to its da-set. But this leaves an empty initial actions set, so the {r▷1} action leading to **(7)** in State **(6)** must be added to the da-set.

(9) $\langle Set,\ IMP_6(0,s,1,0)\rangle$

$$\begin{aligned}\mathcal{R} &= NB(s,1,0) \bullet (\textbf{if } \textsc{nor}(0,0) \neq 1 \textbf{ then } \{\mathtt{q}\triangleleft 1\})NA(0,0,1)\\ &= NA(0,0,1) \bullet NB(s,1,0)\\ &= IMP(0,s,1,0)\end{aligned}$$

$$\boxed{\text{if } s = 1 \text{ then State (6), otherwise State (1)}}$$

The case for $\langle Reset,\ IMP(0,0,0,1)\rangle$ is completely symmetrical. Note that it is entered from State **(4)** in the above set of pairs. The entry state is the state corresponding to State **(6)** above, with $IMP(0,1,1,0)$ replaced by $IMP(1,0,1,0)$ since it will have been the r line that changed, not s.

Note how we had to backtrack in State (8) when we discovered that this branch lead to a node with no initial actions. While this is acceptable in theory (it corresponds to a dangerous deadlock element), in practice we will not want this to happen, so the branch in State (6) that lead up to the state is moved to the da-set. The branch is taken when the reset line goes high while the set line is high, a condition which the specification and the implementation treat in different ways. In any case, it is an illegal operation and should be avoided.

We can gather together the nodes derived above to form the covering process, which is defined as a new part:

```
part LATCH_CT {s, r, q, qb} ⟨
      Set'  ⇐  ( {r▷1}
                   {q◁0}  dis  {{s▷}, {s▷ q◁0}}
                     {qb◁1}  dis  {{r▷}, {r▷ qb◁1}}
                       Reset'  dis  {{s▷1}}
                 + {r▷0} Set'
                 + {s▷1} (Set'  dis  {{r▷1}})
                 + {s▷0} Set'
                 )  dis  {{s▷ r▷}}
    Reset'  ⇐  ( {s▷1}
                   {qb◁0}  dis  {{r▷}, {r▷ qb◁0}}
                     {q◁1}  dis  {{s▷}, {s▷ q◁1}}
                       Set'  dis  {{r▷1}}
                 + {s▷0} Reset'
                 + {r▷1} (Reset'  dis  {{s▷1}})
                 + {r▷0} Reset'
                 )  dis  {{s▷ r▷}}
⟩
```

Although not quite as simple as the original specification, the covering process has much the same structure and is a good deal clearer than the expansion of the implementation.

By way of comparison, let us see what the corresponding weakest safe context looks like. A new process is constructed from the covering node equations that includes each node's distinguishing actions in its refusal set, as discussed on page 151. Since the result falls naturally into several states, it is useful to define a mechanism similar to the part construct for grouping them together:

$$
\begin{array}{l}
\textbf{constraint}\ \mathit{LATCH\_CONS}\ \{\mathtt{r},\mathtt{s},\mathtt{qb},\mathtt{q}\}\ \langle \\
\quad \begin{array}{lcl}
\mathit{CR0} & \Leftarrow & \{\mathtt{s}\lhd 1\}\{\mathtt{qb}\rhd x\}\{\mathtt{q}\rhd y\}\ \mathit{CS1} \\
 & & +\{\mathtt{s}\lhd 0\}\ \mathit{CR0} \\
 & & +\{\mathtt{r}\lhd 0\}\ \mathit{CR0} \\
 & & +\{\mathtt{r}\lhd 1\}\ \mathit{CR1} \\
\mathit{CR1} & \Leftarrow & \{\mathtt{r}\lhd 0\}\ \mathit{CR0}+\{\mathtt{s}\lhd 0\}\ \mathit{CR1} \\
\mathit{CS0} & \Leftarrow & \{\mathtt{r}\lhd 1\}\{\mathtt{q}\rhd x\}\{\mathtt{qb}\rhd y\}\ \mathit{CR1} \\
 & & +\{\mathtt{r}\lhd 0\}\ \mathit{CS0} \\
 & & +\{\mathtt{s}\lhd 0\}\ \mathit{CS0} \\
 & & +\{\mathtt{s}\lhd 1\}\ \mathit{CS1} \\
\mathit{CS1} & \Leftarrow & \{\mathtt{s}\lhd 0\}\ \mathit{CS0}+\{\mathtt{r}\lhd 0\}\ \mathit{CS1}
\end{array} \\
\rangle
\end{array}
$$

This process is defined so that:

$$\mathit{LATCH\_CONS} \bullet \mathit{LATCH} \quad = \quad \mathit{LATCH\_CONS} \bullet \mathit{IMP}$$

The constraint places three important restrictions on the process(es) that may be composed with *LATCH*.

1. Simultaneous occurrences of s and r are not allowed.
2. The inputs are not allowed to change until the events resulting from a change of state are accepted by the environment. This disallows "spikes" on the input lines.
3. The s line is prevented from going high when the r line is already high, and vice-versa.

Notice that the context is allowed to signal the same value on the set and reset channels as was last signalled (for example, two $\{\mathtt{r}\lhd 0\}$ events in sequence). Normally our design style prevents such occurrences, so the above constraint could be somewhat simplified to prevent this possibility.

The original *LATCH* specification can be used freely in any compositional context that follows the rules imposed by *LATCH_CON*. Since latches are general purpose devices that are found in many circuits, the specification and its constraints may be placed in a library of standard parts for future use. Figure 5.4 shows what such an entry might look like. Notice that it is rather larger than the *LATCH_CT* covering tree described above.

Latch	`s, r, q, qb`

Specification:

$$\begin{aligned}
\mathit{Reset} &\Leftarrow \{\mathtt{s}\triangleright 1\}\,\{\mathtt{qb}\triangleleft 0\}\,\{\mathtt{q}\triangleleft 1\}\,\mathit{Set} \\
&\quad + \{\mathtt{r}\triangleright x\}\,\mathit{Reset} + \{\mathtt{s}\triangleright 0\}\,\mathit{Reset} \\
\mathit{Set} &\Leftarrow \{\mathtt{r}\triangleright 1\}\,\{\mathtt{q}\triangleleft 0\}\,\{\mathtt{qb}\triangleleft 1\}\,\mathit{Reset} \\
&\quad + \{\mathtt{s}\triangleright x\}\,\mathit{Set} + \{\mathtt{r}\triangleright 0\}\,\mathit{Set}
\end{aligned}$$

Constraints:

$$\begin{aligned}
\mathit{CR0} &\Leftarrow \{\mathtt{s}\triangleleft 1\}\,\{\mathtt{qb}\triangleright x\}\,\{\mathtt{q}\triangleright y\}\,\mathit{CS1} \\
&\quad + \{\mathtt{s}\triangleleft 0\}\,\mathit{CR0} \\
&\quad + \{\mathtt{r}\triangleleft 0\}\,\mathit{CR0} \\
&\quad + \{\mathtt{r}\triangleleft 1\}\,\mathit{CR1} \\
\mathit{CR1} &\Leftarrow \{\mathtt{r}\triangleleft 0\}\,\mathit{CR0} + \{\mathtt{s}\triangleleft 0\}\,\mathit{CR1} \\
\mathit{CS0} &\Leftarrow \{\mathtt{r}\triangleleft 1\}\,\{\mathtt{q}\triangleright x\}\,\{\mathtt{qb}\triangleright y\}\,\mathit{CR1} \\
&\quad + \{\mathtt{r}\triangleleft 0\}\,\mathit{CS0} \\
&\quad + \{\mathtt{s}\triangleleft 0\}\,\mathit{CS0} \\
&\quad + \{\mathtt{s}\triangleleft 1\}\,\mathit{CS1} \\
\mathit{CS1} &\Leftarrow \{\mathtt{s}\triangleleft 0\}\,\mathit{CS0} + \{\mathtt{r}\triangleleft 0\}\,\mathit{CS1}
\end{aligned}$$

Figure 5–4: A latch as a library component

We have derived two general descriptions of the latch that can be safely used in place of of its implementation in terms of cross coupled NOR gates. The

LATCH specification, however, is not as general as it should be. If the above derivation is studied carefully, it will soon become apparent that the order in which the q and qb events were specified to occur was carefully chosen. It just "happens" to match the order generated by the implementation. This is a dangerous course to adopt, since another (equally valid) implementation might cause these events to occur in the opposite order. There is no real reason to care which ordering is used, so the specification should be changed to indicate that either is acceptable. Since not enough information is available yet to know which implementation will be used, the choice of possible orderings must be nondeterministic. The output sequences of *LATCH* should then be changed to something of the form:

$$\{\mathtt{q}{\triangleleft}x\}\,\{\mathtt{qb}{\triangleleft}y\}\,(\ldots) \oplus \{\mathtt{qb}{\triangleleft}y\}\,\{\mathtt{q}{\triangleleft}x\}\,(\ldots) \quad = \quad \textcircled{\pi}_D(\{\mathtt{qb}{\triangleleft}x\}\,,\{\mathtt{q}{\triangleleft}y\})(\ldots)$$

This says that either q can happen before qb or vice versa—we don't care which. The distinct nondeterministic temporal permutation operator as defined on page 71 captures this concept cleanly.

If this enhancement is made, it will no longer be possible to construct an annotated covering process for the new specification and the previous implementation, since the method was defined only for deterministic choice sums. However, for this particular use of nondeterminism—as a way of indicating "don't care" orderings—a constraint can be constructed that will allow the new flexible specification to be used interchangeably with the implementation. To do this, we refer back to Theorem 3.1.7 which says that the deterministic temporal permutation operator is the identity for the nondeterministic version. From this we observe that:

$$\begin{aligned}
\textcircled{\pi}_D(\{\mathtt{q}{\triangleleft}x\},\{\mathtt{qb}{\triangleleft}y\}) \bullet \pi_D(\{\mathtt{q}{\triangleright}\},\{\mathtt{qb}{\triangleright}\}) &= \textcircled{\pi}_D(\{\mathtt{q}{\triangleleft}x\},\{\mathtt{qb}{\triangleleft}y\})\\
\{\mathtt{q}{\triangleleft}0\}\,\{\mathtt{qb}{\triangleleft}1\}\,(\ldots) \bullet \pi_D(\{\mathtt{q}{\triangleright}x\},\{\mathtt{qb}{\triangleright}y\}) &= \{\mathtt{q}{\triangleleft}0\}\,\{\mathtt{qb}{\triangleleft}1\}\,(\ldots)\\
\{\mathtt{qb}{\triangleleft}0\}\,\{\mathtt{q}{\triangleleft}1\}\,(\ldots) \bullet \pi_D(\{\mathtt{q}{\triangleright}x\},\{\mathtt{qb}{\triangleright}y\}) &= \{\mathtt{qb}{\triangleleft}0\}\,\{\mathtt{q}{\triangleleft}1\}\,(\ldots)
\end{aligned}$$

The process or environment that acts like the deterministic permutation operator can accept what ever output sequence is used. Therefore, any context that is willing to input all possible sequences of the output values is acceptable. The problem is that when the Simulation Ordering is used to check whether a particular context is safe, it will allow ones that accept only a single sequence. The single sequence is unable to cope with the nondeterminism in the specification and so cannot be considered safe. An *ad hoc* solution can be produced by not applying the simulation ordering to terms containing the permutation operator. Thus the context:

$$\{\mathsf{qb}{\triangleright}x\}\,\{\mathsf{q}{\triangleright}y\}\,(\ldots)\ \bullet\ [\![\,]\!]$$

is not an acceptable simulation of:

$$\pi_D(\{\mathsf{qb}{\triangleright}x\},\{\mathsf{q}{\triangleright}y\})(\ldots)\ \bullet\ [\![\,]\!]$$

only when the permutation is being used to absorb a nondeterministic choice.

While the *ad hoc* solution proposed above is adequate, it is not very satisfying and something more general is needed. The next section examines some such methods for coping with nondeterministic choices in a cleaner way.

5.5 Handling Nondeterminism

Nondeterministic choices are difficult to deal with because they can have many different interpretations. They are a sign that information has either been lost, is not yet available (*i.e.*, is unknown), or is not quite representable in the calculus. Each of these interpretations should be treated in a different way, since they symbolize quite different ideas. This chapter is about comparing processes, so it will be instructive to see what happens when they contain some form of nondeterminism. In this section, a look is taken at some different interpretations of nondeterminism and suggestions are offered for comparing nondeterministic processes.

Most uses of nondeterminism fall into the following categories:

1. As a means of capturing truly random behavior. For example, an interrupt might be modelled as a nondeterministic choice that is always present throughout the evolution of the process describing the behavior of a microprocessor. Similarly, a random number generator could be specified by a nondeterministic sum of all of its possible values.

2. To represent loss of information. The application of the abstraction operator produces this type of nondeterminism. Removing the knowledge of why an event happens makes it appear random.

3. To indicate multiple possibilities. This is the form that was discussed at the end of the last section. A number of events are allowed to happen, but it doesn't matter which one really does and/or in what order they all do. The choice will be made using information that is not available, so it cannot be represented by a deterministic choice.

The first interpretation refers to processes that are intentionally random. A process defined with this type of nondeterminism wants it to be preserved, so the only valid comparison between it and another process is through a transformational approach using the laws of the calculus. An implementation of a random number generator, for example, must show that it will generate the same range of numbers as promised by the specification. This range could be represented as a nondeterministic sum of all possible results. An example in the next chapter will go into more detail about this application.

The other two interpretations, however, have some interesting ramifications and will now be considered in more detail.

The second type of nondeterminism typically arises through the application of law [− +]. An action that is part of a deterministic choice can happen "spontaneously" as far as the environment is concerned once it has been made internal to a process. Again, this type of nondeterminism must be dealt with by applying laws of the calculus.

The last interpretation is more interesting, since ambiguity can be removed as more information is made available. Deciding if the information does resolve the ambiguity requires the definition of a new comparison operator, as we shall see below.

5.5.1 Ambiguity

The rationale behind top down design is that specifications for higher level blocks need not describe the detailed behavior of their eventual implementation. Typically, it is sufficient to specify what a sub-block does without worrying about how it does it. For example, at a high level of abstraction, a Boolean equation is sufficient to describe a combinational element. As delays make an appearance in the lower levels, however, more detailed information of the element's behavior through time is required. This information should only be necessary at the lower levels, and not have to be reflected in higher level descriptions.

Event based calculi frequently force one to be far more precise about temporal ordering than is strictly necessary or even desirable. Because of this, specifications must sometimes incorporate features of implementations simply to make them comparable using existing tools, as we saw in Section 4.3.4 when we first implemented a latch. This is highly undesirable since changing the implementation would necessitate changing the specification as well. A specification should remain fixed, yet have just enough ambiguity to allow the designer to investigate various avenues of implementation.

What, exactly, characterizes an ambiguous specification? To find out, let us consider two typical examples. The first is the specification for the latch ($LATCH$) as it was presented on page 85. Recall that the latch was required to output values on both the q and qb channels after a change on one of the input lines. For almost all uses of the latch, the order in which these channels output values is immaterial, so this fact should be reflected in its specification. The temporal permutation operators defined in Section 3.1.4 are ideal for this purpose, since they capture the notion of ambiguity in time. A portion of a revised $LATCH$ might look like this:

$$\{\mathtt{r}\triangleright\mathtt{1}\}\ \textcircled{\pi}_D(\{\mathtt{q}\triangleleft\mathtt{0}\}, \{\mathtt{qb}\triangleleft\mathtt{1}\})\ \ldots$$

A zero can be output on q and a one on qb in any order except simultaneously. An implementation can choose either one of the two sequences that are produced by expanding $\textcircled{\pi}_D$:

$$\textcircled{\pi}_D(\{\mathtt{q}\triangleleft\mathtt{0}\}, \{\mathtt{qb}\triangleleft\mathtt{1}\})\ldots \quad = \quad \{\mathtt{q}\triangleleft\mathtt{0}\}\,\{\mathtt{qb}\triangleleft\mathtt{1}\}\ldots \oplus \{\mathtt{qb}\triangleleft\mathtt{1}\}\,\{\mathtt{q}\triangleleft\mathtt{0}\}\ldots$$

The environment should be prepared to accept either of the sequences in order to cope sensibly with the spec, so passing it the same one every time is perfectly satisfactory. There is no notion of *fairness*† built into the nondeterministic choice operator; the environment cannot expect any particular one of the sequences and must be prepared to receive either. Producing the same sequence every time the $\{\mathtt{r}\triangleright\mathtt{l}\}$ event happens simply looks like we are being unfair to the other choice. This form of ambiguity is called *sequence nondeterminism* because there is a nondeterministic choice between possible sequences.

Another form arises when any one of several events, usually values output on a channel, can happen (hence is called *value nondeterminism*). A typical example, taken from Mitchell's thesis [Mitchell 85], is a change dispensing machine. It inputs a pound coin and outputs some equivalent amount of lower denomination coins. The choice of what particular coins are dispensed depends on what the machine was initially stocked with. The possibilities include 100 pence (*d*), 20 shillings (*s*) or 2 fifty-pence (*fifty-p*) pieces. This behavior is easily described by:

$$CD \;\Leftarrow\; \{\mathtt{pound}\}\left(\{\mathtt{slot}\triangleleft 100\,d\} \oplus \{\mathtt{slot}\triangleleft 20\,s\} \oplus \{\mathtt{slot}\triangleleft 2\,\mathit{fifty}\text{-}p\}\right) CD$$

It is perfectly reasonable for an implementation to dispense only one type of change, yet we have no way of formally showing that it is valid. No amount of equational manipulation will introduce the possibility of dispensing the other types, so an equivalence proof cannot succeed. A looser notion of what it means to implement a nondeterministic choice is needed. As Mitchell points out, however, such a notion cannot be too loose. An implementation that dispenses dollars and cents should not be allowed, even though it might perform correctly in all other respects.‡

Both forms of nondeterminism can, of course, be mixed to specify "don't care when" sequences of "don't care what" values. The categorizations used above

†Fairness means, loosely, that if a nondeterministic choice is presented to the environment infinitely often, every branch will eventually be taken

‡ Note that this form of nondeterminism should not be confused with the "Any Actions" operator defined in Section 2.3.5. The operator provides a number of possibilities any of which can be selected by the enivronment. With value nondeterminism, on the otherhand, the selection is made by unknown factors, hence the environment must be prepared to cope with all possibilities (perhaps using the ANY operator).

simply define the opposing ends of a spectrum. In both cases a single sequence of the implementation must be compared with several in the specification and shown to match only one. This is done by the following relation on processes:

Definition 5.5.1 Implementation

A process P defined by:

$$P \Leftarrow \sum_{i \in I}\!\!\!\!\!\!\bigcirc\;\; \sum_{j \in J} \alpha_{i,j}\, P_{i,j}$$

is *implemented* by another process Q (written Q $\underline{\text{imp}}$ P), where:

$$Q \Leftarrow \sum_{k \in K} \beta_k\, Q_k$$

if for all $j \in J$ and $k \in K$ there exists one and only one $i \in I$ such that $\alpha_{i,j} = \beta_k$ and Q_k $\underline{\text{imp}}$ $P_{i,j}$. □

This just says that the deterministic summation that defines Q must be present just once in the nondeterministic choices provided by P.

Here are some small examples to illustrate the operator's power:

Examples:

$$\begin{aligned}
P &\Leftarrow \{\mathtt{a}\}\, P \\
Q &\Leftarrow \{\mathtt{a}\}\, Q \oplus \{\mathtt{b}\}\, Q1 \\
Q1 &\Leftarrow \{\mathtt{c}\}\, Q \oplus \{\mathtt{d}\}\, Q1 \\
& P\ \underline{\text{imp}}\ Q \\
R &\Leftarrow \{\mathtt{b}\}\, \{\mathtt{c}\}\, R \\
& R\ \underline{\text{imp}}\ Q \\
S &\Leftarrow \textcircled{\pi}(\{\mathtt{q}\lhd\mathtt{1}\}, \{\mathtt{qb}\lhd\mathtt{0}\})\, \{\mathtt{clk}\}\, S \\
I &\Leftarrow \{\mathtt{qb}\lhd\mathtt{0}\}\, \{\mathtt{q}\lhd\mathtt{1}\}\, \{\mathtt{clk}\}\, I \\
& I\ \underline{\text{imp}}\ S
\end{aligned}$$

The last example demonstrates sequence nondeterminism. The environment in which S and I will used must be willing to accept all the possible sequences of $\{\mathtt{q}\triangleleft\mathtt{1}\}$ and $\{\mathtt{qb}\triangleleft\mathtt{0}\}$, since it rarely makes sense to specify a system that can nondeterministically deadlock. For this example, the environment might look something like:

$$E \Leftarrow \pi(\{\mathtt{q}\triangleleft\mathtt{1}\},\{\mathtt{qb}\triangleleft\mathtt{0}\})\,\{\mathtt{clk}\}E$$

Theorem 3.1.7 given on page 73 states that π is the identity for $\textcircled{\pi}$, so this environment will accept anything that S sends it. It is equally happy with the communications presented by I. To demonstrate this, the q and qb channels can be hidden in both processes with an identical result:

$$\begin{aligned} S - \{\mathtt{q},\ \mathtt{qb}\} &= \{\mathtt{clk}\}\,(S - \{\mathtt{q},\ \mathtt{qb}\}) \\ I - \{\mathtt{q},\ \mathtt{qb}\} &= \{\mathtt{clk}\}\,(I - \{\mathtt{q},\ \mathtt{qb}\}) \end{aligned}$$

The Dot Operator distributes over nondeterministic choice (Law $[\bullet\ \oplus]$), so the implementation relation can be applied to compositional contexts. The constraint technique discussed earlier can therefore be generalized to indicated in what contexts one process will implement another.

$$\begin{aligned} U \text{ of sort } \{\mathtt{a},\ \mathtt{b},\ \mathtt{c}\} &\Leftarrow (\{\mathtt{a}\}\,U + \{\mathtt{b}\}\,U) \oplus (\{\mathtt{a}\}\,U + \{\mathtt{c}\}\,U) \\ V \text{ of sort } \{\mathtt{a},\ \mathtt{b},\ \mathtt{c}\} &\Leftarrow \{\mathtt{b}\}\,V + \{\mathtt{c}\}\,V \\ C \text{ of sort } \{\mathtt{a},\ \mathtt{b},\ \mathtt{c}\} &\Leftarrow \{\mathtt{b}\}\,C \end{aligned}$$

$$C \bullet V \ \underline{\text{imp}}\ C \bullet U$$

because:

$$\begin{aligned} C \bullet V &= \{\mathtt{b}\}\,(V \bullet C) \\ C \bullet U &= \{\mathtt{b}\}\,(U \bullet C) \oplus \Delta_{\{\mathtt{a},\ \mathtt{b},\ \mathtt{c}\}} \end{aligned}$$

It is using the implementation ordering that compositional safe contexts come into their own. The <u>dis</u> operator as it stands cannot be extended to cope with these relations since it is so tightly tied to deterministic processes. Other, similar, operators would have to be contrived.

5.6 Summary

This chapter has covered a lot of ground, most of it fairly technical. It began by considering the normal approach to showing that two processes describe the same behavior by transforming them into an identical CIRCAL expression using laws of the calculus. This was demonstrated to be inadequate, because many applications specify only the desired portions of a behavior and ignore aspects of an implementation that are not relevant to the current use. Proving the specification of such an application equivalent to the implementation thus becomes impossible.

Milne addressed this problem by defining *partial specifications* that include only information relevant to the application at hand. He defines a satisfaction relation on processes that determines if one accurately reflects the partial specification imposed by the other. Again, this was shown to be inadequate because portions of the implementing process not covered by the specification might interact in unexpected ways with the context in which the part will be placed.

The satisfaction relations were abandoned and two related approaches to the problem of using partially specified processes safely were considered instead. The first involved looking at the specification and implementation processes as synchronization trees and deriving a greatest common tree called the *annotated covering tree*. The annotations on the tree indicate at each point in the evolution of the processes what actions will be able to differentiate them. The annotations can be manipulated algebraically using several laws that were then derived.

The second approach showed how the dangerous actions could be prevented from happening by defining an appropriate safe context. An ordering was defined that can be used to check if an arbitrary compositional context fulfills the requirements of a safe context. An example using our old friend the latch was then worked to contrast the two approaches.

Deterministic partial specifications alone are not general enough for representing most highlevel behaviors. These behaviors tend to incorporate ambiguity in the form of "don't care what order" sequences of "don't care what" values.

Such statements are readily captured by the nondeterministic choice operator, so a relation for comparing nondeterministic processes was developed. The relation indicates that one process *implements* some of the possibilities allowed by the other.

These three ways of comparing processes simplify considerably the difficult task of writing meaningful and concise specifications.

5.6.1 Contributions of this Chapter

This chapter presented the most important ideas contributed by this thesis. Foremost among these was the idea of constraints on the environment in which a specification may be used. The constraints ensure that there is no way of distinguishing the specification from its implementation, thus allowing them to be used interchangeably. Only through the careful use of constraints can problems of any complexity be subdivided into manageable pieces. Further flexibility is provided by allowing some carefully controlled ambiguity in a specification and then comparing it with its implementation using the *implements* relationship. This was adapted from work by Mitchell and others to fit the CIRCAL framework and completes the spectrum of analytic techniques that can be brought to bear on a problem.

Chapter 6

Some Examples

The presentation of operators and techniques in the previous chapters has been interspersed with some small examples to illustrate their uses. One of the goals of this thesis, however, is to apply these techniques to "real" problems. To this end we now tackle some more ambitious and perhaps more interesting examples. Some are not covered in full detail, but the aim is to investigate some of the pitfalls that are encountered in typical applications rather than to produce large collections of equations.

6.1 Gates

In this section, a surprisingly simple example will be considered that demonstrates some of the difficulties one encounters when performance as well as function is considered. Perhaps the most common operation in designing a digital circuit is connecting together logic gates to implement a particular function. The functional correctness of this circuit can easily be verified by applying the laws of Boolean logic. Verifying performance, or "what happens when," requirements is more difficult, as we shall now see.

In Section 2.8.2, the behavior of a generic n-input, one output functional unit with unspecified delay was described in CIRCAL by the equation:

$$\begin{aligned} \mathit{BoxN\_1}(\tilde{in}) \;\Leftarrow\;& \text{ANY}(\{\mathtt{in}_1\triangleright\},\ldots,\{\mathtt{in}_n\triangleright\}) \\ & \qquad \{\mathtt{out}\triangleleft f(in_1,\ldots,in_n)\}\ \mathit{BoxN\_1}(\tilde{in}) \end{aligned}$$

Where $\tilde{in} = \langle in_1, \ldots, in_n \rangle$. The original definition also had a conditional on the output guard that prevented duplicate values from being output. This conditional will be dropped in the following analysis since its presence complicates the equations needlessly and the results will be just as valid without it.

Later on, in Section 4.3.2, we encountered two cascaded two-input generic boxes in the guise of adders and saw that when all the inputs changed simultaneously, a spurious transient value was output before the correct one. This transient value is not just a feature of the addition function, but rather of the manner in which the box was defined to communicate with its environment. Cascading any generic boxes will produce exactly the same result. In fact, as more boxes are cascaded, more spurious outputs can occur before the correct one. Because of these transient values, one cannot blindly replace an n-input generic box by a cascade of $n-1$ two-input boxes and hope that the circuit will function correctly. The ability to do so depends largely on the behavior of the components that will be connected to the output. In particular, they must be able to absorb the spurious values without ill effect before finally taking action on the correct one. Some form of constraint must be placed on the environment to enforce this requirement before the cascade implementation may be used safely.

Will the ability to accept spurious values be the only constraint that must be placed on the environment, or will there be others? To investigate this, let us contrast the description of a three-input generic box with the partial expansion of the behavior of two cascaded two-input boxes:

$$\begin{aligned} B1 &\Leftarrow Box2\_1\ [\mathtt{b/out}\ g//f] \\ B2 &\Leftarrow Box2\_1\ [\mathtt{b/in_1\ in_3/in_2}] \end{aligned}$$

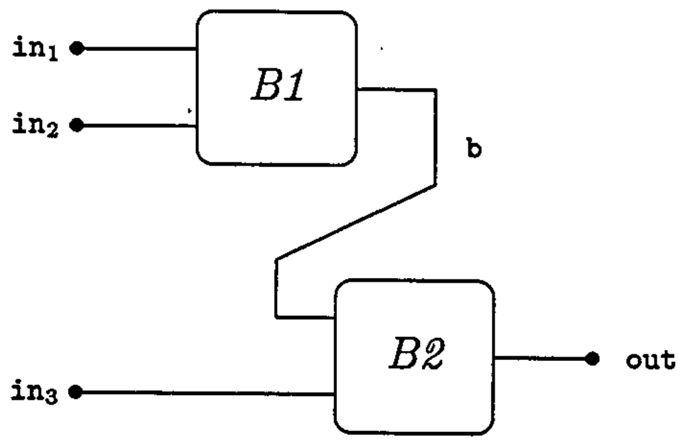


Figure 6–1: How *B1* and *B2* are connected

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\mathit{B1}(in_1, in_2) \bullet \mathit{B2}(b, in_3)) - \mathtt{b} \quad =} \\
(1) & \quad \{\mathtt{in}_1\triangleright\}\ (\mathit{B1}(\tilde{x}) \bullet \{\mathtt{out}\triangleleft f(g(in_1, in_2), in_3)\}\ \mathit{B2}(g(in_1, in_2), in_3)) \\
(2) & + \{\mathtt{in}_2\triangleright\}\ (\ldots) \\
(3) & + \{\mathtt{in}_3\triangleright\}\ (\mathit{B1}(\tilde{x}) \bullet \{\mathtt{out}\triangleleft f(b, in_3)\}\ \mathit{B2}(\tilde{y})) \\
(4) & + \{\mathtt{in}_1\triangleright\ \ \mathtt{in}_3\triangleright\} \\
 & \quad \{\mathtt{out}\triangleleft f(b, in_3)\} \\
 & \qquad (\mathit{B1}(\tilde{x}) \bullet \{\mathtt{out}\triangleleft f(g(in_1, in_2), in_3)\}\ \mathit{B2}(g(\ldots), in_3)) \\
(5) & + \{\mathtt{in}_2\triangleright\ \ \mathtt{in}_3\triangleright\}\ (\ldots) \\
(6) & + \{\mathtt{in}_1\triangleright\ \ \mathtt{in}_2\triangleright\}\ (\ldots) \\
(7) & + \{\mathtt{in}_1\triangleright\ \ \mathtt{in}_2\triangleright\ \ \mathtt{in}_3\triangleright\}\ (\ldots)
\end{array}
$$

Where $\tilde{x} = \langle in_1,\ in_2\rangle$ and $\tilde{y} = \langle b,\ in_3\rangle$.

The specification for the three-input box is much simpler:

$$
\begin{array}{lll}
\mathit{Box3\_1}(in_1, in_2, in_3) & \Leftarrow & \mathrm{ANY}(\{\mathtt{in}_1\triangleright\}, \{\mathtt{in}_2\triangleright\}, \{\mathtt{in}_3\triangleright\}) \\
 & & \qquad \{\mathtt{out}\triangleleft h(in_1, in_2, in_3)\} \\
 & & \qquad\qquad \mathit{Box3\_1}(in_1, in_2, in_3)
\end{array}
$$

The function $h(x, y, z)$ is just $f(g(x, y), z)$.

The most apparent difference between the specification and the constructed system is the number of state variables. The system has four (one for each of the inputs to the constituent boxes), while the specification has only three. The extra variable sets the starting state of the internal channel that connects the two boxes (the b line in Figure 6–1). Once either $\texttt{in}_1$ or $\texttt{in}_2$ or both change value, this variable will always have the value $g(in_1, in_2)$. Consequently, if we force the variable to start with this value, it becomes redundant and can be ignored. Limiting the possible initial conditions can be done by defining the system like this:

$$SYS(in_1, in_2, in_3) \;\Leftarrow\; [B1(in_1, in_2) \bullet B2(g(in_1, in_2), in_3)] - \mathtt{b}$$

Looking closer at the expansion for SYS, we can see that the top level consists of a choice sum of all possible combinations of the input channels—just as is produced by expanding the ANY in the specification. Moving down a level, however, reveals a host of discrepancies between the behavior of SYS and that of *Box3_1*. On line (1), *B1* can recurse and input a new value on $\texttt{in}_1$ or $\texttt{in}_2$ before the value associated with the last change of $\texttt{in}_1$ is output. On line (3), the previous value of the internal channel b—which has no counterpart in *Box3_1*—is used to calculate the new output value. Finally, notice the transient spike on the output line that occurs when $\texttt{in}_1$ and/or $\texttt{in}_2$ change simultaneously with $\texttt{in}_3$ as shown, for example, on line (4). This arises from the fact that inputs to *B1* must pass through two boxes worth of delay before affecting the output, while that of *B2* passes through only one.

The discrepancies fall into two classes. The first involves the inputs changing before the results of the previous input changes have percolated through the cascade. The second is the transient values that might be output if several inputs change simultaneously. The former is easily resolved by requiring that the environment never indulge in two sequential input events before an output event has occurred. The following constraint sums this up:

$$C1 \;\Leftarrow\; \{\mathtt{in}\triangleleft x\}\,\{\mathtt{out}\triangleright y\}\; C1$$

This constraint is fine so long as there is a single output event associated with each change at the inputs. As we have seen, this is not always true becasue of the transients that arise due to the delay added to the signal path by *B1*. How then can the constraint be extended to cope with an unknown number of out events? Some of the transient values may be the same as the correct one, so

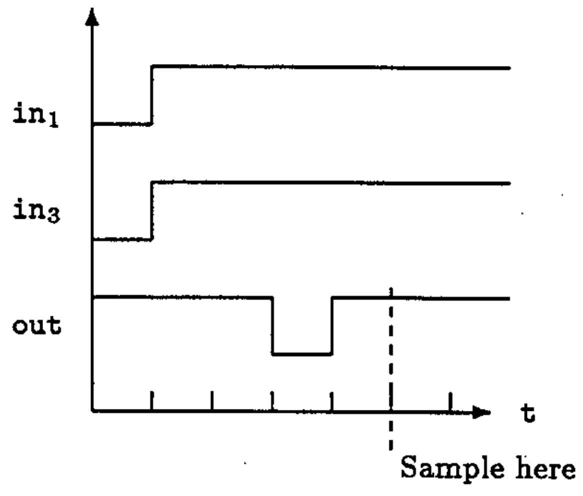


Figure 6–2: Example of a minimum safe sampling time.

we cannot just absorb output events until the right one comes along. Applying the techniques for creating and unfolding Covering Trees as discussed in Section 5.3.4 does not help either. The constraint that results prevents transients by not allowing simultaneous changes on the input lines. Intuitively, this seems unnecessarily restrictive, since there are many situations in which values changing simultaneously should have no harmful effects. A more general constraint on the environment is need.

Since the output will stabilize after a certain amount of time (provided that the inputs do not change during this interval), it seems reasonable to conclude that if we wait long enough, the correct value of the channel will always be observed. How long is long enough? For n cascaded boxes, the worst case is when a value change has to pass through all n of them. If the delay of each box is specified by an integral number of t ticks, clearly the output will be stable after an interval that is the sum of these delays. This interval is called the *minimum safe sampling time*, in units of t, and is pictured in Figure 6–2. It can be determined experimentally for cascaded generic boxes by applying a stimulus to each of the input channels in turn and observing which takes the longest to produce an output. The resulting time is the minimum safe sampling time.

By way of example, let us determine the minimum safe sampling time of the two cascaded two-input boxes used above. The descriptions for *B1* and *B2* are extended to include explicit delays of n and m ticks respectively:

$$\begin{aligned} B1(\widetilde{in}) &\Leftarrow \text{WAIT}_{\mathtt{t}}(\,\text{ANY}(\{\mathtt{in}_1\triangleright\ \mathtt{t}\},\{\mathtt{in}_2\triangleright\ \mathtt{t}\})\{\mathtt{t}\}^n\{\mathtt{b}\triangleleft g(in_1,in_2)\ \mathtt{t}\}\,B1(\widetilde{in})\,) \\ B2(\tilde{a}) &\Leftarrow \text{WAIT}_{\mathtt{t}}(\,\text{ANY}(\{\mathtt{b}\triangleright\ \mathtt{t}\},\{\mathtt{in}_3\triangleright\ \mathtt{t}\})\{\mathtt{t}\}^m\{\mathtt{out}\triangleleft f(a,in_3)\ \mathtt{t}\}\,B2(\tilde{a})\,) \end{aligned}$$

The two are composed and stimuli applied to $\mathtt{in}_1$, $\mathtt{in}_2$ and $\mathtt{in}_3$ in turn:

$$\begin{aligned} SYS(x,y,z) &\Leftarrow (B1(x,y)\ \bullet\ B2(g(x,y),z)) - \mathtt{b} \\ TICKS \text{ of sort } S_{TICKS} &\Leftarrow \{\mathtt{t}\}\,TICKS \\ STIM1 &\Leftarrow \{\mathtt{in}_1\triangleleft v\ \ \mathtt{t}\}\,TICKS \\ STIM2 &\Leftarrow \{\mathtt{in}_2\triangleleft v\ \ \mathtt{t}\}\,TICKS \\ STIM3 &\Leftarrow \{\mathtt{in}_3\triangleleft v\ \ \mathtt{t}\}\,TICKS \\ S_{TICKS} &= \{\mathtt{in}_1,\mathtt{in}_2,\mathtt{in}_3,\mathtt{t}\} \end{aligned}$$

Expanding each of the stimulus applications produces:

$$\begin{aligned} STIM1\bullet SYS(x,y,z) &= \{\mathtt{in}_1\triangleleft v\ \ \mathtt{t}\}\{\mathtt{t}\}^{(n+m+1)}\{\mathtt{out}\triangleleft f(g(v,y),z)\ \mathtt{t}\}\,(\ldots) \\ STIM2\bullet SYS(x,y,z) &= \{\mathtt{in}_2\triangleleft v\ \ \mathtt{t}\}\{\mathtt{t}\}^{(n+m+1)}\{\mathtt{out}\triangleleft f(g(x,v),z)\ \mathtt{t}\}\,(\ldots) \\ STIM3\bullet SYS(x,y,z) &= \{\mathtt{in}_3\triangleleft v\ \ \mathtt{t}\}\{\mathtt{t}\}^{m}\{\mathtt{out}\triangleleft f(g(x,y),v)\ \mathtt{t}\}\,(\ldots) \end{aligned}$$

The first two tests show that the maximum time for an output to appear is $(n+m+2)\,\mathtt{t}$. Any application that needs a stable output value must wait at least that many ticks from the time that the inputs change before taking action on the result. This can be expressed as:

$$\begin{aligned} C2 \Leftarrow\ & \text{WAIT}_{\mathtt{t}}(\,\text{ANY}(\{\mathtt{in}_i\triangleleft\ \mathtt{t}\},\ i=1,2,3) \\ & \quad (\{\mathtt{out}\triangleright\ \mathtt{t}\}\ +\{\mathtt{t}\})^{(n+m+2)}\,C2\,) \end{aligned}$$

The repeated term indicates that the environment must have the potential to accept an out event for $(n+m+2)$ ticks after the inputs change. The $\{\mathtt{out}\triangleright\ \mathtt{t}\}$ guard binds the variable *out* which will then contain the final output value of the channel after the transients have died out. *C2* supercedes *C1* because it handles transients properly and because it works for the case when no output is generated corresponding to an input event (the value didn't change), which *C1* does not. The delay between $\mathtt{in}_i$ events prevents them from interleaving with the output events.

In typical digital applications, registers clocked by a system clock separate blocks of combinational elements. These registers sample the outputs of the block once they have stabilized and hold them steady for other components to use. The above constraint ensures that the sampling period in such a system will be greater than the worst case delay.

6.1.1 Generalizing the Constraints

Now that we have seen how constraints can be formulated for three-input generic boxes, it is time to generalize them to apply to n-input boxes. A typical box—with $n > 2$ inputs—will be taken to have a delay of d ticks and is specified by:

$$\begin{aligned} \mathit{BoxN\_1}(\widetilde{in}) \;\Leftarrow\; & \mathrm{WAIT}_{\mathtt{t}}\big(\,\mathrm{ANY}(\{\mathtt{in}_i\triangleright\ \mathtt{t}\}\,,\ i = 1\ldots n) \\ & \quad \{\mathtt{t}\}^d \\ & \quad\quad \{\mathtt{out}\triangleleft f_n(f_{n-1}(\ldots), in_n)\ \ \mathtt{t}\} \\ & \quad\quad\ \ \mathit{BoxN\_1}(\widetilde{in})\,\big) \end{aligned} \tag{6.1}$$

It can be implemented by cascading $n-1$ two-input boxes with the array operator:

$$\mathit{IMP}(\widetilde{in}) \;\Leftarrow\; \underset{j=1}{\overset{n-1}{\boxed{A}}}\ \mathit{Box2\_1}_j$$

$$\begin{aligned} \mathit{Box2\_1}_j \;=_{def}\; & \mathit{Box2\_1}\ [\mathtt{in}_{j+1}/\mathtt{in}_2,\quad \mathtt{a}/\mathtt{in}_1, \\ & \qquad f_j(f_{j-1}(\ldots), in_j)/\!\!/in_1,\quad in_j/\!\!/in_2\quad m_j//m] \end{aligned}$$

Where $A =$ $[\mathtt{out}_j/\mathtt{a}_{j+1}]$. Each two-input box has a delay of m and is described by:

$$\begin{aligned} \mathit{Box2\_1}(in_1, in_2) \;\Leftarrow\; & \mathrm{WAIT}_{\mathtt{t}}\big(\,\mathrm{ANY}(\{\mathtt{in}_1\triangleright\ \mathtt{t}\},\ \{\mathtt{in}_2\triangleright\ \mathtt{t}\}) \\ & \qquad \{\mathtt{t}\}^m\,\{\mathtt{out}\triangleleft f(in_1, in_2)\ \mathtt{t}\}\ \mathit{Box2\_1}(in_1, in_2)\big) \end{aligned}$$

The delay of the spec is defined to be $d = m_1 + \cdots + m_n + (n-1)$.

Theorem 6.1.1 Cascading Generic Boxes

The implementation and the specification for an n-input generic box as defined above are interchangeable in any context that satisfies:

$$\mathit{CON} \;\Leftarrow\; \mathrm{WAIT}_{\mathtt{t}}\big(\,\mathrm{ANY}(\{\mathtt{in}_i\triangleleft\ \mathtt{t}\},\ i = 1\ldots n)\,(\{\mathtt{t}\}\ + \{\mathtt{out}\triangleright\ \mathtt{t}\})^M\,\mathit{CON}\,\big)$$

Where $M = m_1 + \cdots + m_n + (n-1)$. *The context should not take action on any values received until* M *ticks have elapsed.*

Proof: By induction on n.

Base Case The base case of $n = 3$ was considered briefly above and will now be considered in detail. The implementation is defined by:

$$\begin{aligned}
\mathit{B1}(\tilde{in}) &\Leftarrow \text{WAIT}_\texttt{t}\big(\text{ANY}(\{\texttt{in}_1\triangleright\ \texttt{t}\},\ \{\texttt{in}_2\triangleright\ \texttt{t}\}) \\
&\qquad \{\texttt{t}\}^{m_1}\ \{\texttt{a}\triangleleft f_1(in_1, in_2)\ \texttt{t}\}\ \mathit{B1}(\tilde{in})\big) \\
\mathit{B2}(\tilde{a}) &\Leftarrow \text{WAIT}_\texttt{t}\big(\text{ANY}(\{\texttt{a}\triangleright\ \texttt{t}\},\ \{\texttt{in}_3\triangleright\ \texttt{t}\}) \\
&\qquad \{\texttt{t}\}^{m_2}\ \{\texttt{out}\triangleleft f_2(a, in_3)\ \texttt{t}\}\ \mathit{B2}(\tilde{a})\big) \\
\mathit{IMP}(in_1, in_2, in_3) &\Leftarrow \big(\mathit{B1}(in_1, in_2) \bullet \mathit{B2}(f_1(in_1, in_2), in_3)\big) - \texttt{a}
\end{aligned}$$

A context is now defined that accepts a value on the out channel and re-outputs it on the b channel $M = m_1 + m_2 + 2$ ticks after the input channels had changed. It is the simplest context to satisfy both *CON* and the requirement that the environment not make use of any of the transient values. Here is its definition:

$$\begin{aligned}
\mathcal{E}[\![\ ]\!] &\Leftarrow (E \bullet [\![\ ]\!]) - \texttt{out} \\
E &\Leftarrow \text{WAIT}_\texttt{t}\big(\text{ANY}(\{\texttt{in}_i\triangleleft\ \texttt{t}\},\ i = 1, 2, 3) \\
&\qquad (\{\texttt{t}\} + \{\texttt{out}\triangleright v\ \texttt{t}\})^{M-1} \\
&\qquad\quad (\{\texttt{b}\triangleleft v\ \texttt{t}\}\ + \{\texttt{out}\triangleright v\ \texttt{b}\triangleleft v\ \texttt{t}\})E\big)
\end{aligned}$$

Any occurrence of an out event will bind v, which gets output on channel b on the first tick that it is safe to do so. Now we have to show that:

$$\mathcal{E}[\![\ \mathit{IMP}\ ]\!] \quad =_? \quad \mathcal{E}[\![\ \mathit{Box3\_1}\ ]\!]$$

The behavior of the three-input box is described by Equation 6.1, where the delay d is $m_1 + m_2 + 1$.

When *Box3_1* is placed in $\mathcal{E}$, its output event always synchronizes with the $\{\texttt{out}\triangleright v\ \texttt{b}\triangleleft v\ t\}$ guard, so:

$$\begin{aligned}
\mathcal{E}[\![\ \mathit{Box3\_1}(\tilde{in})\ ]\!] &= [E \bullet \mathit{Box3\_1}(\tilde{in})] - \texttt{out} \\
&= \text{WAIT}_\texttt{t}\big(\text{ANY}(\{\texttt{in}_i\triangleleft\ \texttt{t}\},\ i = 1, 2, 3) \\
&\qquad \{\texttt{t}\}^{m_1+m_2+1} \\
&\qquad\quad \{\texttt{b}\triangleleft f_2(f_1(\ldots), in_3)\ \texttt{t}\}\ \mathcal{E}[\![\ \mathit{Box3\_1}(\tilde{in})\ ]\!]\big)
\end{aligned} \tag{6.2}$$

But this is identical in structure to *Box3_1* with out relabelled to b, which shows that the context has no effect on the specification. Placing *IMP* in $\mathcal{E}$ should result in an identical expression. Here goes:

$$\begin{aligned}
\mathcal{E}[\![\mathit{IMP}]\!] &= (E \bullet \mathit{IMP}) - \{\,\mathtt{out}\,\} \\
&= \{\mathtt{t}\}\, \mathcal{E}[\![\mathit{IMP}]\!] \\
&\quad + \{\mathtt{in}_1 \triangleleft\ \mathtt{t}\}\, E' \\
&\quad + \{\mathtt{in}_2 \triangleleft\ \mathtt{t}\}\, E' \\
&\quad + \{\mathtt{in}_3 \triangleleft\ \mathtt{t}\}\, E' \\
&\quad + \{\mathtt{in}_1\ \mathtt{in}_2 \triangleleft\ \mathtt{t}\}\, E' \\
&\quad + \{\mathtt{in}_1\ \mathtt{in}_3 \triangleleft\ \mathtt{t}\}\, E' \\
&\quad + \{\mathtt{in}_2\ \mathtt{in}_3 \triangleleft\ \mathtt{t}\}\, E' \\
&\quad + \{\mathtt{in}_1\ \mathtt{in}_2 \triangleleft\ \mathtt{in}_3 \triangleleft\ \mathtt{t}\}\, E' \\
&= \mathrm{WAIT}_{\mathtt{t}}\big(\,\mathrm{ANY}(\{\mathtt{in}_i \triangleleft\ \mathtt{t}\},\ i = 1,2,3)\, E'\,\big) \\
E' &\Leftarrow \{\mathtt{t}\}^{m_1+m_2+1}\, \{\mathtt{b} \triangleleft f_2(f_1(in_1, in_2), in_3)\ \mathtt{t}\}\, \mathcal{E}[\![\mathit{IMP}]\!]
\end{aligned}$$

All the terms have the same resultant because E not only makes the b event happen after a fixed delay, but also prevents input events from interleaving with events that occur during this interval. Thus if *B2* inputs a value on the $\mathtt{in}_3$ channel, *B1* must pass the time in its $\mathrm{WAIT}_{\mathtt{t}}$ loop until the b event happens.

Substituting the definition of E', an expression is obtained that is identical to Equation 6.2. Both the specification and the implementation behave identically in $\mathcal{E}$, as required. We can conclude that the theorem is valid for the base case of $n = 3$.

Induction Step For the induction step, we assume that the equality is true for a box with $n-1$ inputs and show that it then holds for n inputs. Precisely the same reasoning is used as in the base case, with *B1* replaced by the $n-1$ input box. This box is described by:

$$\begin{aligned}
\mathit{BN\text{-}1}(\widetilde{in}) &\Leftarrow \mathrm{WAIT}_{\mathtt{t}}\big(\,\mathrm{ANY}(\{\mathtt{in}_i \triangleright\ \mathtt{t}\},\ i = 1 \ldots n-1) \\
&\qquad \{\mathtt{t}\}^{M_{n-1}} \\
&\qquad\quad \{\mathtt{a} \triangleleft f_{n-1}(f_{n-2}(\ldots), in_{n-1})\ \mathtt{t}\}\, \mathit{BN\text{-}1}(\widetilde{in})\,\big) \\
M_{n-1} &= m_1 + \cdots + m_{n-1} + (n-2)
\end{aligned}$$

B2 then becomes the n'th box, so it will have a delay of m_n and compute the function f_n. The context $\mathcal{E}$ is changed to have a sampling period of $m_1 + \cdots + m_n + n$. By using exactly the same analysis as in the base case, it can be shown that the modified *B2* composed with *BN-1* behaves identically to *BoxN_1* when placed in $\mathcal{E}$. □

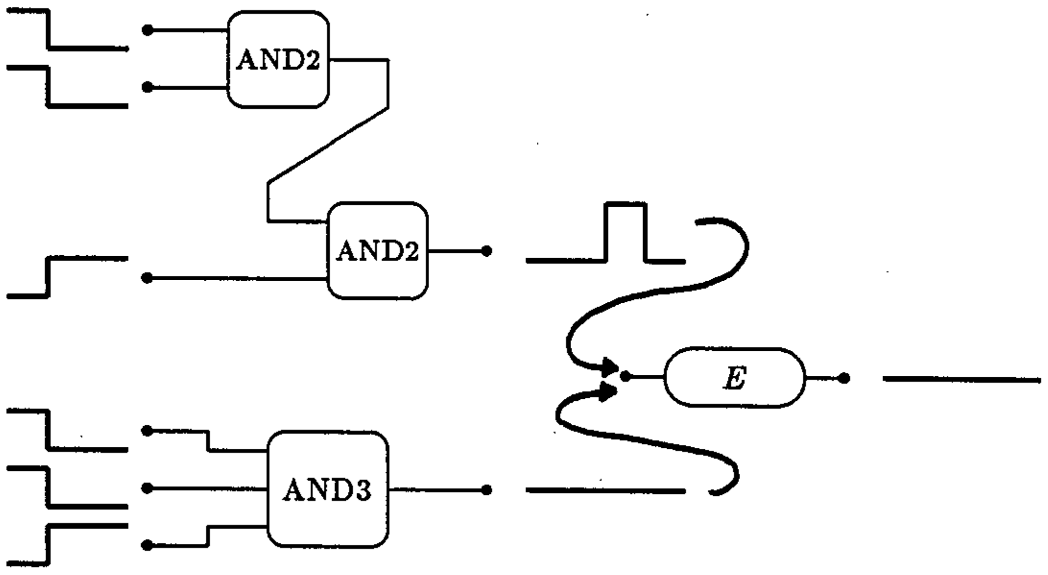


Figure 6–3: The wave forms input to the two circuits produce corresponding output waveforms that are in turn fed to the environment process E. This process absorbs transients to produce a stable waveform.

Figure 6–3 summarizes pictorially the results of using this theorem to construct a three input AND gate.

As we saw earlier, the length M of the danger period can be determined empirically from the implementation by applying stimuli to each of the inputs in turn. The passage of time is marked until an output event occurs for each stimulus; the largest of these timings is the length of the danger period. If this method is used, the individual gate delays of a particular implementation need not be known and the constraint on its use can be written directly in terms of the measured value. Unfortunately, the method works only for cascaded boxes and not for general networks of boxes, because it relies on the fact that only one output event will happen for a given input event.[†] In a general network a particular input may effect several gates in parallel, leading to transient outputs. These invalidate the condition for stopping the marking of time, making the empirical approach unusable. The only solution is to trace all possible paths through the block; adding up the delays of the boxes pas$^{\text{sed}}_{\wedge}$ through. Using the largest such value, one can still formulate a constraint similar to *CON* that makes the block appear to be an n input, m output generic box.

6.1.2 Summary

This section has shown that simple circuits often develop unexpected complexities when performance as well as functional criteria are considered. Although only generic boxes were considered, similar techniques can be applied to any transient producing circuit. If a safe sampling time can be determined for the circuit's outputs, a constraint can be setup that requires that the environment be able to handle their transients. Transient values in the form of spikes and glitches are the bane of digital designers, particularly in high performance applications. Being able to verify their harmlessness may well reduce the problems that they cause considerably. The extremely precise nature of the calculus makes the analysis more laborious than might be expected for such a simple problem, but having been done once, the results can be applied to any similar situation.

6.2 A Hardware Card Deck

Although difficult to deal with, nondeterminism can be integral to a particular application and therefore unavoidable. In this section we shall examine such an application in the form of a hardware implementation of a deck of playing cards. The example has the following objectives:

[†]Although this might not be true if the boxes do not generate an event if the value to be output is the same as was last output.

- To demonstrate how a truely nondeterministic specification (as opposed to one containing ambiguity) can be implemented.
- To show some of the problems encountered when processes are connected in a loop.
- To show how such a loop may be dynamically shrunk.

The card deck is represented by a sequence of "card events" that correspond to the drawing of a random card. The events are communications on the card channel of values drawn from the set:

$$CV = \{A, 2, 3, 4, 5, 6, 7, 8, 9, J, Q, K\} \times \{\spadesuit, \clubsuit, \heartsuit, \diamondsuit\}$$

Any sequence is possible, so we use the di$\overset{\text{s}}{\wedge}$tinct nondeterministic temporal permutation operator to represent this:

$$SHUFFLED\_DECK \;\Leftarrow\; \textcircled{T}_D(\{\mathtt{card}\triangleleft x\},\quad x \in CV)\, SHUFFLED\_DECK$$

The distinct version must be used since we don't want events of the form $\{\mathtt{card}\triangleleft A\spadesuit \quad \mathtt{card}\triangleleft J\diamondsuit\}$ to occur.

In essence, this specification says that a shuffled deck is the nondeterministic sum of all possible sequences of card events. The unpredictable nature of the nondeterministic choice operator is used to represent random chance. So how can randomness be implemented in terms of hardware devices? Is the calculus expressive enough to deal with it? To answer these questions, consideration of the card deck will temporarily be put off until the problem of generating random values has been solved. This is done in the next section.

6.2.1 Generating Random Numbers

Many schemes exist for producing random numbers in hardware. Most rely on a random and non-uniform clock to continuously increment some form of counter. Sampling the output of the counter yields the required number. Two forces work together to produce the randomness of the result. One is the unpredictable nature of the clock that drives the counter—its period constantly changes. The other is the unknown point at which the output of the counter is sampled with respect to the time it takes to cycle through all its values. Hardware circuits that are concerned with generating truly random numbers use some physically random source such as thermal noise across the junction of a diode as the clock. Further randomness can be introduced by having the user press a button that samples the output of the counter to produce a random number. The first form will not be considered here, because it is a physical embodiment of the nondeterministic choice operator. The second is more interesting, since it illustrates some uses of the abstraction operator.

Implementing the button sampling scheme is easy. A normal, uniform, clock is controlled by a pushbutton switch in such a way that the clock cycles only while the button is held down. The clock drives a counter that is incremented modulo the maximum number to be generated with each cycle. Provided that the clock rate divided by the period of the counter is much higher than the rate at which the button can be pushed, a large number of cycles will be run through even for a "quick" push. When the button is released, the counter halts with a random number on its outputs. The randomness derives from such things as switch bounce and inability of the person controlling the button to hold it down for anywhere near identical periods relative to the cycle time of the counter.

Here is the description of such a part in CIRCAL. It inputs a range on channel `range`, waits for the button to be pressed and returns the result via the channel `number`.

```
part RAND_GEN {range, number : int} ⟨
    Get_Val     ⇐ {range▷n} Wait_for_But
    Wait_for_But ⇐ {ButDwn▷} Count(0)
    Count(x)    ⇐ {clk} Count(x + 1 mod n)
                    + {ButUp number◁x} Get_Val
⟩
```

The net effect should be equivalent to something of the form:

$$RAND \Leftarrow \{\mathtt{range}\triangleright n\} \sum_{i=0}^{n-1} \{\mathtt{number}\triangleleft i\}\ RAND$$

How can we prove this equivalence?

The first thing to notice is that *RAND_GEN* has a declared sort of {`range`, `number`}, whereas the equations inside the part's body refer to other channels, namely {`clk`, `ButUp`, `ButDwn`}. In Section 2.7, we adopted the convention that channels referenced in the body of the part but not in the sort declaration are implicitly abstracted away. Thus *RAND_GEN* should more properly be written:

$$
\begin{array}{l}
\textbf{part}\ \mathit{RAND\_GEN}\ \{\texttt{range},\ \texttt{number} : \mathrm{int}\}\ \langle \\
\quad (\,\mathit{Get\_Val}\ \Leftarrow\ \ldots) - \{\texttt{clk},\ \texttt{ButUp},\ \texttt{ButDwn}\} \\
\quad (\,\mathit{Wait\_for\_But}\ \Leftarrow\ \ldots) - \{\texttt{clk},\ \texttt{ButUp},\ \texttt{ButDwn}\} \\
\quad (\,\mathit{Count}\ \Leftarrow\ \ldots) - \{\texttt{clk},\ \texttt{ButUp},\ \texttt{ButDwn}\} \\
\rangle
\end{array}
$$

Applying Law $[-\ +]$ to each of the equations produces:

$$
\begin{aligned}
\mathit{Get\_Val}' &\Leftarrow \{\mathrm{range}\triangleright n\}\ \mathit{Wait\_for\_But}' \\
\mathit{Wait\_for\_But}' &\Leftarrow \mathit{Count}'(0) \\
\mathit{Count}'(x) &\Leftarrow \mathit{Count}(x+1 \bmod n)\ \oplus\ \{\mathrm{number}\triangleleft x\}\ \mathit{Get\_Val}'
\end{aligned}
$$

The state *Wait_for_But′* can be eliminated entirely since it silently becomes *Count′*(0). *Get_Val′* then matches the first portion of *RAND* up to the nondeterministic sum. All we have to do now is show that the recursive equation for *Count′* produces this sum.

Substituting for x all the possible values that it can have (in this case $0 \ldots n-1$), we obtain n interdependent equations:

$$
\begin{aligned}
\mathit{Count}'(0) &\Leftarrow \mathit{Count}'(1) \oplus \{\mathrm{number}\triangleleft 0\}\ \mathit{Get\_Val}' \\
\mathit{Count}'(1) &\Leftarrow \mathit{Count}'(2) \oplus \{\mathrm{number}\triangleleft 1\}\ \mathit{Get\_Val}' \\
&\ \vdots \\
\mathit{Count}'(n-1) &\Leftarrow \mathit{Count}'(0) \oplus \{\mathrm{number}\triangleleft n-1\}\ \mathit{Get\_Val}'
\end{aligned}
$$

The states can be combined and the guards collected into a summation to produce:

$$
\mathit{Count}'(0) \Leftarrow \mathit{Count}'(0) \oplus \sum_{i=0}^{n-1} \{\mathrm{number}\triangleleft i\}\ \mathit{Get\_Val}'
$$

The unguarded recursion comes about because the `clk` channel was hidden, so we can use law [rec$_4$] (derived in Section 3.1) to simplify it:

$$Count'(0) \quad = \quad \sum_{i=0}^{n-1} \{\texttt{number}\triangleleft i\}\ Get\_Val'$$

Inserting this back into the equation for Get_Val' results in:

$$Get\_Val' \quad \Leftarrow \quad \{\texttt{range}\triangleright n\} \sum_{i=0}^{n-1} \{\texttt{number}\triangleleft i\}\ Get\_Val'$$

The result is identical in structure to the behavior of *RAND*. We can therefore conclude that the button scheme does implement randomness properly and that:

$$RAND\_GEN \quad = \quad RAND$$

6.2.2 Picking a Card

Selecting a random card from the card deck is not quite as simple as generating a random number. Each card must be removed from the deck as it is picked so that it cannot be picked again. One way to look at it is that as long as the button is held down, the cards are "riffled" through very quickly. When the button is released, the card that is being touched is displayed and removed from the deck. This continues until only one card remains.

The easiest way to implement the riffling action is to use something like a shift register. Instead of flip-flops, the shifter will be made up of "card slices." Each slice inputs the value of its neighbor to the left and passes it to the right with every clock tick. The output of the last slice is fed back around to the first, producing a loop. If one of the slices is initially marked as containing a "token" and the rest are cleared, the token will cycle around the loop as the clock ticks away time. When the button is released, the shifting stops and the slice containing the token is considered to be picked. When the button is next pushed, this element removes itself from the loop after first passing the token on to its neighbor on the right. Eventually, as the button is pushed multiple times, the loop shrinks until only one element remains. A clear signal can then be generated which re-initializes the slices, effectively reshuffling the deck.

Each card slice is conceptually quite simple. It can be in one of two states: (1) either shifting a value from left to right (unpicked), or (2) passing the value through transparently (picked). The first state is described by:

$$AVAIL \;\Leftarrow\; \{\mathtt{in}\triangleright v\}\,\{\mathtt{clk}\}\,\{\mathtt{out}\triangleleft v\}\,AVAIL$$

The second by:

$$PICKED \;\Leftarrow\; \{\mathtt{in}\triangleright v\}\,\{\mathtt{out}\triangleleft v\}\,PICKED$$

Both the in and out channels have type bit.

AVAIL acts just like a D-type flipflop. It samples the input line and reproduces the value on the output line after the next clock edge. As with D flip-flops, multiple copies of *AVAIL* can be cascaded to achieve a shifting effect; values input by the left-most stage are passed right one stage with each tick. The way *AVAIL* is defined, each stage must wait for the previous one to output a value before it can accept the clock event and output in turn to the next stage. So what happens when the output of the last slice is connected to the input of the first? Each will be waiting on the output of its neighbor on the left, resulting in the classic "snake eating its tail" form of deadlock. Indeed, Δ would result if several instances of *AVAIL* are formed into a loop using the Dot and relabelling operators and the expression expanded using law [$\bullet$ +].

To avoid this nasty state of affairs, each slice is "decoupled" from its predecessor to the left. The decoupling agent corresponds to a time delay between a value being output and its reception by the next stage. In an actual circuit, there will probably be combinational elements between the stages which would provide this delay. Here is the description of the decoupling agent:

$$DECOUPLE(v) \;\Leftarrow\; \{\mathtt{out}\triangleleft v\}\,\{\mathtt{in}\triangleright x\}\,DECOUPLE(x)$$

Notice that the state value is output before a new one is read. This is necessary to "prime" the loop. After the first event, the agent acts exactly like an unspecified delay.

To get a feel for how the complete system will work, let us connect two card slices and their decoupling agents in a loop and expand the expression's behavior.

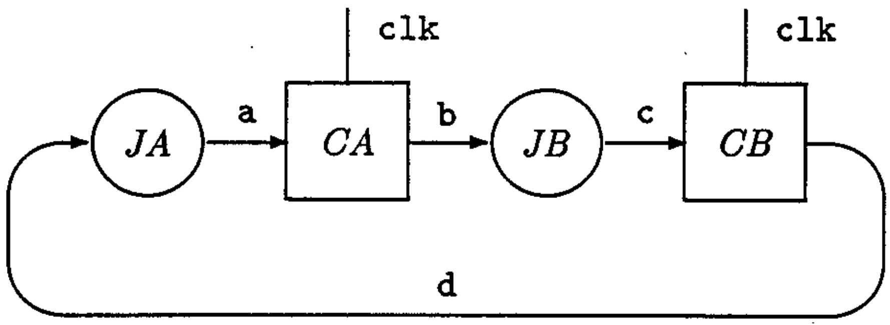


Figure 6–4: Cascaded card slices and their joins.

The system is pictured in Figure 6–4 and is described by:

$$\begin{aligned}
CA &\Leftarrow AVAIL\,[\texttt{a/in},\ \ \texttt{b/out}]\\
CB &\Leftarrow AVAIL\,[\texttt{c/in},\ \ \texttt{d/out}]\\
JA &\Leftarrow DECOUPLE\,[\texttt{d/in},\ \ \texttt{a/out}]\\
JB &\Leftarrow DECOUPLE\,[\texttt{b/in},\ \ \texttt{c/out}]\\
SYS &\Leftarrow CA \bullet JA(1) \bullet CB \bullet JB(0)
\end{aligned}$$

Since expanding *SYS* produces a bit of a mess, we expand half of it at a time:

$$\begin{aligned}
CA \bullet JA(v) \quad = \quad & \{\texttt{a}\triangleleft v\}\,(\{\texttt{d}\triangleright x\}\,(JA(x) \bullet \{\texttt{clk}\}\,\{\texttt{b}\triangleleft v\}\,CA)\\
&+ \{\texttt{clk}\}\,(\{\texttt{b}\triangleleft v\}\,CA \bullet \{\texttt{d}\triangleright x\}\,JA(x))\\
&+ \{\texttt{clk}\ \ \texttt{d}\triangleright x\}\,(\{\texttt{b}\triangleleft v\}\,CA \bullet JA(x)))
\end{aligned}$$

CB will always produce a d event after a clk event, so the first and third branches of the choice sum will never be taken when the two halves are composed. Expanding the second branch further produces:

$$\begin{aligned}
\{\texttt{b}\triangleleft v\}\,CA \bullet \{\texttt{d}\triangleright x\}\,JA(x) \quad &= \quad \{\texttt{b}\triangleleft v\}\,\{\texttt{d}\triangleright x\}\,(JA(x) \bullet CA)\\
&\quad + \{\texttt{d}\triangleright x\}\,\{\texttt{b}\triangleleft v\}\,(JA(x) \bullet CA)\\
&\quad + \{\texttt{d}\triangleright x\ \ \texttt{b}\triangleleft v\}\,(JA(x) \bullet CA)\\
&= \quad \pi(\{\texttt{b}\triangleleft v\}, \{\texttt{d}\triangleright x\})\,(JA(x) \bullet CA)
\end{aligned}$$

The second half can be analyzed in precisely the same manner. Combining the two at the top level results in:

$$\begin{aligned}(JA(v)\bullet CA)\bullet(JB(w)\bullet CB) &= \{\mathtt{a}\lhd v\}\,\{\mathtt{c}\lhd w\}\,(R1\bullet R2)\\ &\quad+\{\mathtt{c}\lhd w\}\,\{\mathtt{a}\lhd v\}\,(R1\bullet R2)\\ &\quad+\{\mathtt{c}\lhd w\quad \mathtt{a}\lhd v\}\,(R1\bullet R2)\\ &= \pi(\{\mathtt{a}\lhd v\},\{\mathtt{c}\rhd w\})\,(R1\bullet R2)\end{aligned}$$

$$\begin{aligned}R1(v) &\Leftarrow \{\mathtt{clk}\}\,\pi(\{\mathtt{b}\lhd v\},\{\mathtt{d}\rhd x\})\,(JA(x)\bullet CA)\\ R2(w) &\Leftarrow \{\mathtt{clk}\}\,\pi(\{\mathtt{d}\lhd w\},\{\mathtt{b}\rhd y\})\,(JB(y)\bullet CB)\end{aligned}$$

To produce the expansion for SYS, the above equations are collected, v is set to 1 and w to 0. Two auxiliary states, *S1* and *S2*, are defined implicitly to simplify the notation:

$$\begin{aligned}SYS &\Leftarrow S1(1)\bullet S2(0)\\ &= (JA(1)\bullet CA)\bullet(JB(0)\bullet CB)\\ &= \pi(\{\mathtt{a}\lhd 1\},\{\mathtt{c}\lhd 0\})\,\{\mathtt{clk}\}\,.\\ &\qquad(\pi(\{\mathtt{b}\lhd 1\},\{\mathtt{d}\rhd x\})\,S1(x)\;\bullet\;\pi(\{\mathtt{d}\lhd 0\},\{\mathtt{b}\rhd y\})\,S2(y))\\ &= \pi(\{\mathtt{a}\lhd 1\},\{\mathtt{c}\lhd 0\})\,\{\mathtt{clk}\}\,\pi(\{\mathtt{b}\lhd 1\},\{\mathtt{d}\lhd 0\})\,(S1(0)\bullet S2(1))\end{aligned}$$

Applying the temporal permutation operator as early as possible has reduced the composition to a manageable form. The result shows that the outputs of the join elements and those of the card elements are allowed to happen in any order (concurrently). Notice how the state variables have swapped values in the resultant. This demonstrates the shifting effect of the slices and also that the value of the last slice will circle back to the first.

Fifty-two slices are needed to model a complete card deck, so it is worth generalizing the above result.

Proposition 6.2.1 *A stage is defined to be a decoupling agent connected to a card slice. Cascading n stages that are in the shift state so that the rightmost stage is connected to the leftmost produces a system with n state variables that is described by:*

$$SYS(v_1,\ldots,v_n) \Leftarrow \pi(\{\mathtt{in}_i\lhd v_i\})\,\{\mathtt{clk}\}\,\pi(\{\mathtt{out}_i\lhd v_i\})\,SYS(v_n,v_1,\ldots,v_{n-1})$$

For $i=1,\ldots,n$.

Proof: By induction on n. The base case is the expansion of:

$$SYS(v) \Leftarrow AVAIL \bullet (DECOUPLE(v)\ [\texttt{out/in}\ \ \texttt{in/out}])$$

Which is just:

$$\begin{aligned} SYS(v) &= \{\texttt{in}\triangleleft v\}\,\{\texttt{clk}\}\,\{\texttt{out}\triangleleft v\}\,SYS(v) \\ &= \{\texttt{in}\triangleleft v\}\,\{\texttt{clk}\}\,\{\texttt{out}\triangleleft v\}\,\{\texttt{in}\triangleleft v\}\,\{\texttt{clk}\}\,\{\texttt{in}\triangleleft v\}\ \ldots \end{aligned}$$

as required. No shifting takes place because there is only one element.

Induction Step The $n = 2$ case was considered in detail above, so we must now use a similar analysis to show that given a loop of $n-1$ stages,

$$\begin{aligned} S(v_1,\ldots,v_{n-1}) &\Leftarrow \pi(\{\texttt{in}_i\triangleleft v_i\})\ \{\texttt{clk}\}\ \pi(\{\texttt{out}_i\triangleleft v_i\}) \\ &\qquad\qquad S(v_{n-1},v_1,\ldots,v_{n-2}) \\ i &= 1,\ldots,n-1 \end{aligned}$$

the loop can be broken, another stage added, and the loop closed again to produce an identical equation with the $i = 1,\ldots,n-1$ qualification replaced by $i = 1,\ldots,n$.

The loop is broken by renaming the input channel of the leftmost decoupling agent to some independent label, say b. Channel b is completely independent of {clk, in, out}, so it can interleave not only with the out events, but also with clk. This makes the expansion of the broken loop rather messy. However, we know that when the loop is reconnected, b will be attached to $\texttt{out}_{n-1}$, which always occurs after clk, so a constraint C can be defined that exploits this knowledge to prevent b from happening simultaneously with the clock event. It will then interleave only with the output events. Here is the constraint and the broken loop:

$$\begin{aligned}
C &\Leftarrow \{\mathtt{clk}\}\,\{\mathtt{b}\triangleleft\}\,C \\
STAGE_1(v_1) &\Leftarrow AVAIL\,[\mathtt{in}_1/\mathtt{in},\quad \mathtt{out}_1/\mathtt{out}] \\
&\quad \bullet(DECOUPLE(v_1)\,[\mathtt{b}/\mathtt{in},\quad \mathtt{in}_1/\mathtt{out}]) \\
STAGE_i(v_i) &\Leftarrow AVAIL\,[\mathtt{in}_i/\mathtt{in},\quad \mathtt{out}_i/\mathtt{out}] \\
&\quad \bullet(DECOUPLE(v_i)\,[\mathtt{out}_{i-1}/\mathtt{in},\quad \mathtt{in}_i/\mathtt{out}]) \\
S'(v_1,\ldots,v_{n-1}) &\Leftarrow \prod_{i=1}^{n-1} STAGE_i(v_i) \\
C\bullet S' &= \pi(\{\mathtt{in}_i\triangleleft v_i\})\ \{\mathtt{clk}\}\ \pi(\{\mathtt{b}\triangleleft x\},\{\mathtt{out}_i\triangleleft v_i\}) \\
&\qquad C\bullet S'(x,v_1,\ldots,v_{n-2})
\end{aligned}$$

S' is just S with the loop broken.

To perform the induction step, $STAGE_n$ must be composed with S'. The stage's behavior is described by:

$$\begin{aligned}
STAGE_n(v_n) &\Leftarrow AVAIL\,[\mathtt{in}_n/\mathtt{in},\quad \mathtt{out}_n/\mathtt{out}] \\
&\quad \bullet\,(DECOUPLE(v_n)\,[\mathtt{out}_{n-1}/\mathtt{in},\quad \mathtt{in}_n/\mathtt{out}]) \\
&= \{\mathtt{in}_n\triangleright x\}\,\{\mathtt{clk}\}\,\{\mathtt{out}_n\triangleleft x\}\,(\ldots) \\
&\quad \bullet\,\{\mathtt{in}_n\triangleleft v_n\}\,\{\mathtt{out}_{n-1}\triangleright v_n\}\,(\ldots) \\
&= \{\mathtt{in}_n\triangleleft v_n\}\,\{\mathtt{clk}\}\,\pi(\{\mathtt{out}_n\triangleleft v_n\},\{\mathtt{out}_{n-1}\triangleright x\})\,STAGE_n(x)
\end{aligned}$$

Composing this with S' and the constraint produces:

$$\begin{aligned}
C\bullet S'\bullet STAGE_n &= \pi(\{\mathtt{in}_i\triangleleft v_i\})\quad \{\mathtt{clk}\}\quad \pi(\{\mathtt{b}\triangleright x\},\{\mathtt{out}_i\triangleleft v_i\}) \\
&\qquad C\bullet S'(x,v_1,\ldots,v_{n-2})\bullet STAGE_n(v_{n-1}) \\
i &= 1,\ldots,n
\end{aligned}$$

The loop is closed again by relabelling b to $\mathtt{out}_n$. The value stored by v_n no longer drops off the end, but instead is bound to x. $STAGE_n$ clearly satisfies C (that b and hence $\mathtt{out}_n$ occur after clk), so the constraint can be removed:

$$\begin{aligned}
SYS(v_1,\ldots,v_n) &= S'(v_1,\ldots,v_{n-1})\bullet STAGE_n(v_n) \\
&= \pi(\{\mathtt{in}_i\triangleleft v_i\})\ \{\mathtt{clk}\}\ \pi(\{\mathtt{out}_i\triangleleft v_i\})SYS(v_n,v_1,\ldots,v_{n-1}) \\
i &= 1,\ldots,n
\end{aligned}$$

This completes the inductive step and concludes the proof.[†] □

To pick a random card in the range 1 to n, we use the same trick as was discussed earlier of clocking the shifter for as long as a button is held down. When the button is raised—signalled by the `ButUp` event—the clock is stopped and the outputs are allowed to settle. Provided that only one of the decoupling agents starts out with a state value of one (the rest being zero), only one of the n output lines will be high. The button's actions are specified by:

$$\textbf{part}\ \mathit{BUTTON}\,\{\texttt{ButUp, ButDwn, clk}\}\ \langle$$
$$\begin{aligned} \mathit{WAIT} &\Leftarrow \{\texttt{ButDwn}\}\{\texttt{clk}\}\,\mathit{CLK} \\ \mathit{CLK} &\Leftarrow \{\texttt{clk}\}\,\mathit{CLK} + \{\texttt{ButUp}\}\,\mathit{WAIT} \end{aligned}$$
$$\rangle$$

At least one clock edge always separates button events.

The card deck as we have implemented it so far sets one of the fifty-two output lines high if that card is being "touched." The card being touched changes with each clock event for as long as the clock is running. Only when the clock stops can the outside world consider this card picked. Our specification for the complete deck as given at the beginning of the section (page 181) represents a card being picked as a communication on the `card` channel of a value draw from the set CV. A transformer must therefore be defined so that the two differing approaches can be compared. The transformer remembers which slice last indicated that it was being touched and uses this value to generate the proper `card` event when the clock stops. This is signalled by `ButUp` which can be used to do the sampling provided that it occurs after all the output lines have settled:

$$\mathit{ST\_TO\_CRD}(\tilde{v}) \Leftarrow [\mathrm{ANY}(\{\texttt{out}_i \triangleright v_i\}) + \{\texttt{ButUp}\quad \texttt{card} \triangleleft c(\tilde{v})\}]\ \mathit{ST\_TO\_CRD}(\tilde{v})$$

The function c maps unary information onto card values. For example, if v_{25} is one and the rest are zero, $c(\tilde{v})$ would select the twenty-fifth value from the set of

[†]Inspiration for the trick of breaking the cyclic dependency with a constraint so that induction can be used came from a conversation with Kim Larsen.

card values *e.g.*, the result might be Q♠. If this system were implemented as an integrated circuit, a grid decoding scheme could be used to generate signals on two sets of pins: thirteen value pins and four suit pins. Figure 6–5 shows how the card slices might be arranged to do this in CMOS logic. Any active slice pulls its row and column lines low, generating a negative-true logic signal on the corresponding pins.

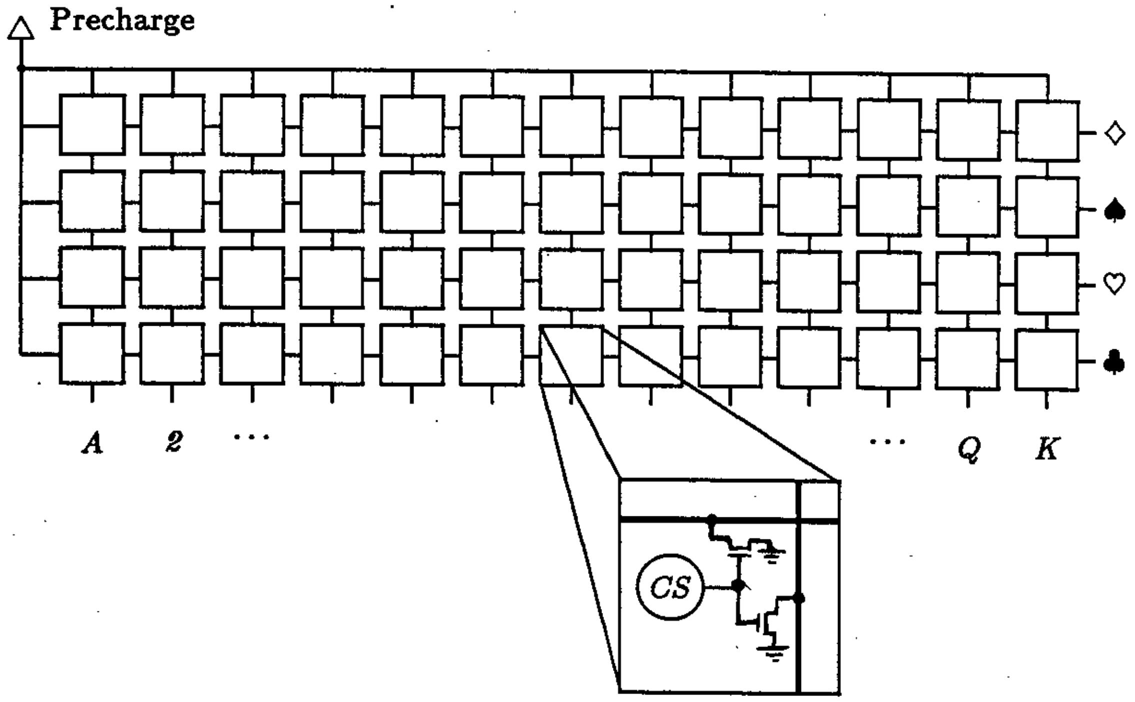


Figure 6–5: Possible grid decoding floorplan for a card deck chip

The composition of 52 slices, the *BUTTON* part, and the state-to-card-event transformer can be analyzed in precisely the same fashion as the random number generator discussed earlier. Abstracting away the button and clock events results in a system that produces a sequence of random {card◁x} events *without* removing the chosen card from the deck. To make it a true card deck, each slice must pass into the *PICKED* state if the clock stops while it is active (has a high value on its output channel). Since this happens when the button is released, the ButUp event can be used to make the state transition. Here is the description of a card slice updated to reflect these requirements and also changed to combine the function of the decoupling agent with that of the shifter:

$$
\begin{array}{l}
\textbf{part } \mathit{SLICE}(v)\ \{\texttt{ButUp, in, out, clk}\}\ \langle \\
\quad \begin{array}{rcl}
\mathit{AVAIL}(v) & \Leftarrow & \{\texttt{clk}\}\,\pi(\{\texttt{in}{\triangleright}x\}, \{\texttt{out}{\triangleleft}v\})\,\mathit{AVAIL}(x) \\
 & & +\{\texttt{ButUp}\}\,(\textbf{if } v=1 \textbf{ then } \mathit{PICKED} \textbf{ else } \mathit{AVAIL}(v)) \\
\mathit{PICKED} & \Leftarrow & (\{\texttt{clk}\} + \{\texttt{ButUp}\})\,\mathit{PICKED} \\
 & & +\{\texttt{in}{\triangleright}x\}\,\{\texttt{out}{\triangleleft}x\}\,\mathit{PICKED}
\end{array} \\
\rangle
\end{array}
$$

The card deck itself is defined to be:

$$
\begin{array}{rcl}
\mathit{DECK}(\tilde{v}) & \Leftarrow & (\mathit{BUTTON} \bullet \mathit{SLICES}(\tilde{v}) \\
 & & \bullet\, \mathit{ST\_TO\_CRD}(\tilde{v})) - \{\texttt{in, out, clk, ButUp, ButDwn}\} \\
\mathit{SLICES}(\tilde{v}) & \Leftarrow & \prod_{i=1}^{52} \mathit{SLICE}_i(v_i) \\
\mathit{SLICE}_i(v_i) & = & \mathit{SLICE}\ [\texttt{out}_{g(\texttt{i})}/\texttt{in},\ \texttt{out}_\texttt{i}/\texttt{out}] \\
g(i) & = & \textbf{if } i=1 \textbf{ then } 52 \textbf{ else } i-1
\end{array}
$$

The next section shows in detail what must be done to remove a card slice from the loop.

6.2.3 Removing a Card from the Deck

To illustrate what happens when a card is removed from the deck, we now compose a slice in the *AVAIL* state (*S1*) with another in the PICKED state (*S2*). The composition should function identically to a single available slice (with *S2* acting like an unspecified delay);

$$
\begin{array}{rcl}
\mathit{S1}(v) & \Leftarrow & \mathit{SLICE}\ [\texttt{b/out}] \\
\mathit{S2} & \Leftarrow & \mathit{SLICE.PICKED}\ [\texttt{b/in}] \\
\mathit{SA}(v) & = & (\mathit{S1}(v) \bullet \mathit{S2}) - \texttt{b} \\
 & = & \{\texttt{clk}\}\ \pi(\{\texttt{in}{\triangleright}x\}, \{\texttt{out}{\triangleleft}v\})\,\mathit{SA}(x) \\
 & & +\{\texttt{ButUp}\}\,(\textbf{if } v=1 \textbf{ then } \mathit{SP} \textbf{ else } \mathit{SA}(v)) \\
\mathit{SP} & = & \mathit{S2} \bullet (\mathit{S1.PICKED}\ [\texttt{b/out}]) \\
 & = & (\{\texttt{clk}\} + \{\texttt{ButUp}\})\,\mathit{SP} \\
 & & +\{\texttt{in}{\triangleright}x\}\,((\mathit{S1.PICKED}\ [\texttt{b/out}] \bullet \{\texttt{out}{\triangleleft}x\}\,\mathit{S2}) - \texttt{b})
\end{array}
$$

By multiple applications of laws $[\bullet\ +]$, $[-\ +]$ and $[\oplus_I]$. Notice that until a ButUp event occurs, *SA* behaves exactly like *AVAIL*. This means that as long as values are being shifted around the loop, already picked slices will be completely transparent. The problem comes when a slice that is connected to a picked slice is itself picked. As shown in the *SP* state above, the $\texttt{out}\triangleleft x$ event will interleave with the next $\texttt{in}\triangleright y$ event from *S1*'s *PICKED* state. Two picked slices in sequence should appear as an unspecified delay (*i.e.*, as something that does an $\texttt{in}\triangleright x$ followed by an $\texttt{out}\triangleleft x$), which these do not because of the interleaving. The extraneous possibilities introduced by the interleaving can be eliminated by specifying that the input event occur only after a clock event. This forces the $\texttt{out}\triangleleft x$ event to happen before any of the events in *S1.PICKED* [b/out]:

$$\begin{aligned} C1 &\Leftarrow \{\texttt{clk}\}.\{\texttt{in}\triangleleft x\}\ C1 \\ C1 \bullet SP &= (\{\texttt{clk}\} + \{\texttt{ButUp}\})(C2 \bullet SP) \\ &\quad + \{\texttt{in}\triangleright x\}\ \{\texttt{out}\triangleleft x\}\ (C1 \bullet SP) \end{aligned}$$

We can conclude that a picked slice in series with an available one acts completely transparently provided that the input events to the pair are separated by clock ticks. As long as there is at least one unpicked slice, this constraint will be satisfied. The output channel of the unpicked slice is connected to the first input in the chain of picked slices and events happening on that channel always do so after a clock event. Care must be taken not to let the last available slice enter the picked state itself as deadlock would ensue. The slice would be waiting for input from the previous slice before outputting a value of its own. But the previous slice is the end of a chain of picked slices which inputs this output value before generating any output of its own. Both portions are waiting for each other's output, hence the deadlock.

In addition to preventing the last picked slice from entering the *PICKED* state, a mechanism is need for resetting the entire deck so that the slices are made available again. Moreover, one and only one of the slices must have have its state variable initialized to one. Both these requirements are easily met having each slice generate a `picked` signal as it is chosen. The deck is considered empty when all the slices have generated such a signal. Each slice is then reset to the *AVAIL*(0) state, except for the last one picked, which will already be in the *AVAIL*(1) state. Here is the final description for a card slice, incorporating all of these additions:

$$
\begin{aligned}
&\textbf{part } \mathit{SLICE}(v)\ \{\texttt{in},\ \texttt{out},\ \texttt{next},\ \texttt{picked}\!:\!\textbf{bit},\ \texttt{clr},\ \texttt{clk},\ \texttt{ButUp},\ \texttt{ButDwn}\}\ \langle \\
&\quad \mathit{AVAIL}(v) \Leftarrow \{\texttt{clk}\}\,\pi(\{\texttt{in}\triangleright x\},\ \{\texttt{out}\triangleleft v\},\ \{\texttt{next}\triangleleft v\})\ \mathit{AVAIL}(x) \\
&\qquad + \{\texttt{ButDwn}\}\ \mathit{AVAIL}(v) \\
&\qquad + \{\texttt{ButUp}\}\ (\textbf{if}\ \ v = 1\ \ \textbf{then}\ \{\texttt{picked}\triangleleft 1\}\ \mathit{BP} \\
&\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ \mathit{AVAIL}(v)) \\
&\quad \mathit{BP} \Leftarrow \{\texttt{clr}\}\,\{\texttt{ButDwn}\}\,\{\texttt{picked}\triangleleft 0\}\ \mathit{AVAIL}(1) \\
&\qquad + \{\texttt{ButDwn}\}\ \mathit{PICKED} \\
&\quad \mathit{PICKED} \Leftarrow (\{\texttt{clk}\} + \{\texttt{ButDwn}\} + \{\texttt{ButUp}\})\ \mathit{PICKED} \\
&\qquad + \{\texttt{clr}\}\,\{\texttt{picked}\triangleleft 0\}\ \mathit{AVAIL}(0) \\
&\qquad + \pi(\{\texttt{out}\triangleleft 0\},\ (\{\texttt{in}\triangleright x\}\,\{\texttt{next}\triangleleft x\}))\ \mathit{PICKED} \\
&\rangle
\end{aligned}
$$

While the structure similar to that of the previous definition, several important changes have been made. The output channel of the slice has been split in two: **out** and **next**. The **next** channel passes values to the neighboring slice, while the **out** channel signals the value of the slice itself. This prevents transient values from being signalled to the outside world when the slice is in the *PICKED* state and values are being passed through transparently. To reach this state, the **out** line must have been high when the button was released, so it must go low when the button is next pressed, hence the interleaving of the $\{\texttt{out}\triangleleft 0\}$ event with the $\{\texttt{in}\triangleright x\}\,\{\texttt{next}\triangleleft x\}$ sequence. The ButUp event still indicates that the card is picked if it is in the $\mathit{AVAIL}(1)$ state at the time, but instead of going directly to the *PICKED* state, an intermediate state is passed through first. This state checks if a clear signal is generated before the next ButDwn event, as would happen when the slice is the last card in the deck. If so, the slice resets itself to the $\mathit{AVAIL}(1)$ immediately without ever entering the picked state. All the other slices will have been picked and will therefore be reset to $\mathit{AVAIL}(0)$.

Generating the clear signal is easy since all the slices indicate when they have been chosen on their **picked** channel. The signal is just the logical AND of these channels:

$$
\begin{aligned}
\mathit{CLEAR}(\tilde{v}) \Leftarrow\ & \text{ANY}(\texttt{picked}_i\triangleright v_i,\quad i = 1,\ldots,52) \\
& \qquad (\textbf{if}\ \wedge\tilde{v}\ \textbf{then}\ \{\texttt{clr}\}\,)\,\mathit{CLEAR}(\tilde{v})
\end{aligned}
$$

The complete deck in its final form (*i.e.*, outputting only **card** events) is implemented by

$$\begin{aligned}
DECK(\tilde{v}) &\Leftarrow [CLEAR(\tilde{v}) \bullet BUTTON \bullet ST\_TO\_CRD(\tilde{v}) \\
&\quad \bullet \prod_{i=1}^{52} SLICE_i(v_i)] \\
&\quad - \{\mathtt{in_i},\ \mathtt{out_i},\ \mathtt{clr},\ \mathtt{next_i},\ \mathtt{ButUp},\ \mathtt{ButDwn},\ \mathtt{picked_i},\ \mathtt{clk}\} \\
SLICE_i(v_i) &= SLICE\ [\rho] \\
\rho &= \langle \mathtt{next}_{g(\mathtt{i})}/\mathtt{in},\ \mathtt{out_i}/\mathtt{out},\ \mathtt{next_i}/\mathtt{next},\ \mathtt{picked_i}/\mathtt{picked}\rangle \\
g(i) &= \textbf{if } i = 1 \textbf{ then } 52 \textbf{ else } i - 1
\end{aligned}$$

In the last section, we discussed how the loop of card slices will "spontaneously" produce a random card value when all the channels except the output channel are abstracted away. In this section, we took the analysis a step further by considering the removal of chosen cards from the deck. It was demonstrated that the loop does indeed shrink to produce a smaller card deck which is in turn amenable to the same analysis as the larger case. The structure of the resulting behavior mimics the tree-like form generated by the nondeterministic temporal permutation operator as used in the top level specification:

$$SHUFFLED\_DECK \Leftarrow ⓉD(\{\mathtt{card}\triangleleft x\},\ x \in CV)\, SHUFFLED\_DECK$$

Finally, a clear signal resets the slices when they have all been picked, paralleling the recursion in *SHUFFLED_DECK*.

An implementation of the card slice in terms of gate level components is described in Appendix A. The analysis is made possible only by extensive use of constraints to partition the problem into manageable pieces. The final, unresolved constraints provide operating requirements (*e.g.*, maximum clock rate) for that particular implementation.

6.2.4 Summary

This example was interesting because nondeterminism due to abstraction (as discussed in Section 5.5) played a key rôle. It was demonstrated that random values can be produced by cycling through a number of possibilities and abstracting away from the action that stops the cycling. To the environment, it appears as though the part spontaneously produces a sequence of values.

Constructing the card deck involved connecting a series of card slices in a loop so that the shifting action would cycle through each in turn. The slices were initially defined to input a value, wait for the clock event and then output it. This proved inadequate because when connected in a loop, each slice would be waiting on its neighbor to produce output, thus leading to deadlock. To break the deadlock, decoupling agents were introduced that acted like unspecified delays between the slices. These allowed the slices time enough to output their value before inputting new ones from their neighbors.

Finally the problem of removing picked card slices from the deck was considered. As each card was picked, it transformed itself into a state that transparently passed values through, effectively shrinking the loop. Care had to be taken that the last available slice did not enter the picked state since that would introduce the same type of deadlock as discussed above. This was done by re-initializing the deck before the slice could be removed.

The example has achieved its goals of demonstrating abstraction nondetermism, problems with cyclically connected process and how such cycles can be dynamically reduced. Some use of constraints was also made in order to simplify a few of the derivations.

6.3 A CRT Controller

Most computer terminals contain a microchip that generates the signals required for displaying character data on a video monitor. The chip fetches character codes from a bank of RAM that stores a representation of the screen (the screen memory), converts them to serial bit data, and uses this data to modulate the intensity of the electron beam in the monitor. In addition to painting the picture, signals must be generated for blanking the beam when it retraces to the beginning of a new scan line and when it returns to the top of the screen.

These signals must happen at precisely the right time for a meaningful picture to result.

In this section, we will examine a highly idealized CRT controller (loosely based on the National Semiconductor DP8350) together with some of the associated parts that go into making a display controller. The purpose is twofold. First of all, the plethora of timing related signals make this an excellent showcase for demonstrating the techniques for representing time that were discussed in Chapter 3. Multiple clocking levels should challenge our ability to decompose the design into temporally as well as spatially self-contained blocks.

Secondly, the basic simulation method presented in Chapter 4 will be illustrated by using it to explore the behavior of the system. The idea is not to produce a fully verified implementation, but rather examine some design decisions and see what effect they have on performance.

The section begins by describing the requirements imposed by the video monitor and goes on to show how they determine the timing discipline. Video monitor fundamentals are covered only cursorily as are the numerous functions provided by most CRT controller chips. The interested reader is referred to [Kane 80] for a far more complete introduction.

6.3.1 The Video Monitor

A typical video monitor consists of a Cathode Ray Tube (CRT) and circuitry for decoding the control signals that drive it. A CRT is an evacuated glass tube with a phosphor coated screen at one end and an electron gun at the other. The gun emits a beam of electrons that, if the intensity is high enough, will cause the phosphor to glow and so display a picture. The beam is swept across the screen from left to right with its intensity modulated by the picture information in what are called *scan lines*. At the end of each line, the beam's intensity is reduced and it is quickly brought back to the left and down one line (the *horizontal retrace* operation). The time at which this occurs is controlled by the HSYNC control pulse. Similarly, when the beam reaches the bottom of the screen, the VSYNC pulse is used to initiate the vertical retrace to the top left corner. In most European monitors, the VSYNC pulse is generated at the rate of 50 Hertz.

This can be expressed as a clock relationship (see page 65):

$$\begin{aligned} \mathit{REFRESH} &= \mathrm{Clk}(\,\mathtt{t}_{sec}, 50, \mathtt{vsync}\,) \\ &= \mathrm{rec}\ X.(\,\{\mathtt{t}_{sec}\ \mathtt{vsync}\}\,\{\mathtt{vsync}\}^{49}\,X\,) \end{aligned}$$

For computer applications, the screen is usually divided up into a grid of virtual dots called *pixels*. These dots are produced by turning the electron beam fully at precisely the right points in its path across the screen. The beam is controlled by a binary VIDEO signal, so a particular pixel on a scan line can be turned on by bringing this signal high at the correct time after an HSYNC. For this example, we will consider a very small grid of four scan lines per screen with four pixels per scan line. Furthermore, it is assumed that the vertical refresh takes the same amount of time as it does to display a scan line. This means that there are five HSYNC's per VSYNC (one per line plus one for the refresh time):

$$\begin{aligned} \mathit{VSYNC} &= \mathrm{Clk}(\,\mathtt{vsync}, 5, \mathtt{hsync}\,) \\ &= \mathrm{rec}\ X.(\,\{\mathtt{vsync}\ \mathtt{hsync}\}\,\{\mathtt{hsync}\}^{4}\,X\,) \end{aligned}$$

If *VSYNC* is composed with *REFRESH*, we can see that the horizontal synchronization pulse will happen with a frequency of $50 \times 5 = 250$ Hertz.

Characters are displayed as blocks of pixels, some dark and some light, so that when viewed at a distance the correct shape is seen. Our toy monitor will display two characters per line and two lines per screen. Each character is two pixels wide and two scan lines high. Figure 6–6 shows the resulting grid with four different characters being displayed to form the shape of a cross.

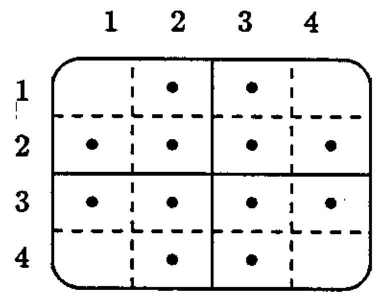


Figure 6–6: Screen representation

The fastest clock in the system is the one that modulates the video signal to the monitor. In the worst case, it must make a transition for every pixel on the screen (if they are all on). Since there are two characters per line and two pixels per character, there must be at least four dot-clock transitions before an HSYNC. The horizontal retrace time is arbitrarily chosen to be the same as the time it takes to display one character (two dot clocks), so there will actually be six transitions:

$$\begin{aligned} HSYNC &= \mathrm{Clk}(\mathtt{hsync}, 6, \mathtt{dotclk}) \\ &= \mathrm{rec}\ X.(\{\mathtt{hsync\ dotclk}\}\,\{\mathtt{dotclk}\}^5\, X) \end{aligned}$$

Composing this relation with *VSYNC* and *REFRESH* shows that the maximum clock rate in the system will be $50 \times 5 \times 6 = 1.5$ kiloHertz.

Two other clock rates are also defined that, although not need by the monitor, are needed by the control circuitry. These are the line clock and the character clock. Each character is two scan lines high, so the rate at which a line of text is displayed is half the HSYNC rate. A new line is only started during the display portion and not during the vertical retrace, so the definition of the line rate is not a simple clock relationship:

$$\begin{aligned} LINE\_RATE \Leftarrow\ & \{\mathtt{vsync\ hsync}\}\,\{\mathtt{hsync\ line}\}\,\{\mathtt{hsync}\} \\ & \{\mathtt{hsync\ line}\}\,\{\mathtt{hsync}\}\ LINE\_RATE \end{aligned}$$

Note that this parallels the *VSYNC* definition in that there are five `hsync`'s per `vsync`. The `line` signal is asserted at the same time as `hsync`, so whatever circuitry that uses it to access data has the horizontal retrace time to do so (two cycles of `dotclk`).

Since there are two pixels per character in the horizontal direction, the character clock goes at half the rate of the dot clock:

$$CHAR\_RATE = \mathrm{Clk}(\mathtt{char}, 2, \mathtt{dotclk})$$

6.3.2 Generating Characters

Text is stored in a *screen memory* as a sequence of character codes. These codes must be converted into pixel information and sent to the monitor at the correct time to produce a meaningful image. The decoding process is accomplished by a *character generator* that takes as inputs the character code, the number of the current scan line and the dot clock. High and low bit values are shifted out on the `video` channel synchronized to the dot clock. The bit patterns for the characters are usually stored in a Read-Only-Memory inside the character generator. For the purposes of this simulation, the ROM will be represented by a function that takes a scan line and a character number and returns a two bit quantity that is the bit pattern of that scan line of the character. Furthermore, the beam must be blanked during the horizontal and vertical retraces, so a bit pattern of zeros is loaded when these are initiated. Here is the description of the complete part, based on these requirements:

$$
\begin{array}{l}
\textbf{part } \mathit{CHAR\_GEN}(\tilde{u})\ \{\texttt{data}:0\ldots3,\ \texttt{video}:\textbf{bit},\ \texttt{char},\ \texttt{hsync},\ \texttt{dotclk}\}\ \langle \\
\quad \begin{array}{rcl}
\mathit{CG}(\mathit{data},l,v) & \Leftarrow & \{\texttt{data}\triangleright\}\ \mathit{CG}(\tilde{u}) \\
 & & +\{\texttt{dotclk}\}\ \{\texttt{video}\triangleleft(v \wedge 1)\}\ \mathit{CG}(\mathit{data},l,(v \gg 1)) \\
 & & +\{\texttt{char}\ \ \texttt{dotclk}\}\ \{\texttt{video}\triangleleft(\mathrm{rom}(\mathit{data},l) \wedge 1)\} \\
 & & \qquad \mathit{CG}(\mathit{data},l,\underline{\mathrm{rom}}(\mathit{data},l) \gg 1) \\
 & & +\{\texttt{hsync}\ \ \texttt{char}\ \ \texttt{dotclk}\}\ \{\texttt{video}\triangleleft 0\} \\
 & & \qquad \mathit{CG}(\mathit{data},(l+1 \bmod 2),0) \\
 & & +\{\texttt{vsync}\ \ \texttt{hsync}\ \ \texttt{char}\ \ \texttt{dotclk}\}\ \{\texttt{video}\triangleleft 0\} \\
 & & \qquad \mathit{BLANK}(\mathit{data},0,0) \\
\mathit{BLANK}(\mathit{data},l,v) & \Leftarrow & \{\texttt{hsync}\ \ \texttt{char}\ \ \texttt{dotclk}\}\ \mathit{CG}(\mathit{data},l,v) \\
 & & +\{\texttt{char}\ \ \texttt{dotclk}\}\ \mathit{BLANK}(\mathit{data},l,v) \\
 & & +\{\texttt{dotclk}\}\ \mathit{BLANK}(\mathit{data},l,v) \\
 & & +\{\texttt{data}\triangleright\}\ \mathit{BLANK}(\mathit{data},l,v)
\end{array} \\
\rangle
\end{array}
$$

The definition is separated into two major states, each performing a different function. The first (*CG*) describes what happens during the visible portion of the display cycle, while the second (*BLANK*) is responsible for the interval during vertical retrace. Both have the same state variables, although only *data* is modified during *BLANK*. The state variable v stores the bit string currently being shifted out. The expression $(v \wedge 1)$ selects the low order bit, which controls the electron beam's intensity, and $(v \gg 1)$ shifts the string one bit position right and places a zero on the left *e.g.,* $11_2 \gg 1 = 01_2$. The variable l is incremented modulo 2 with each `hsync`, so it alternates between 0 and 1 and indexes the scan lines that make up a text line. It is initialized to 0 at the beginning of the vertical retrace in preparation for the first scan line of the first character. Finally, the variable *data* always contains a number in the range 0 to 3 which is the code of the character that is to be displayed. Using this and *l*, the function <u>rom</u> calculates a two-bit quantity according to Table 6–1 which is the bit representation of that character at that scan line.

data	*l*	*rom*	
0	0	0	1
	1	1	1
1	0	1	0
	1	1	1
2	0	1	1
	1	0	1
3	0	1	1
	1	1	0

Table 6–1: Table of data returned by the rom function.

6.3.3 The Screen Memory

A character code to be displayed must be made available before the char event that causes it to be decoded happens. Since the screen memory that stores the code will have an access delay, steps must be taken to fetch the code before its needed. This is done while the previous character is being displayed:

$$\begin{aligned} \mathit{FETCH} \Leftarrow\ & \{\texttt{vsync hsync char}\}\{\texttt{char}\}\{\texttt{char}\}\,\mathit{FETCH} \\ & + \{\texttt{hsync char}\}\{\texttt{get}\}\{\texttt{char}\}\{\texttt{get}\}\{\texttt{char}\}\,\mathit{FETCH} \end{aligned}$$

No characters are fetched during the vertical retrace time as shown by the first line. The second line shows that the first character is fetched during the horizontal retrace time and the next one character time later.

The screen memory returns a character code for the current screen location on the data channel (which will usually be the system bus) when requested to do so by a **get** signal. In practice, address generation circuitry is needed to access the memory, but for this example it will be considered as part of the memory module. Thus to the rest of the system, the memory will look like a sequence of data values corresponding to the code of the first character on the screen, followed by that of the second, followed by the first code again (used to generate the second scan line), and so forth. For the screen representation given in Figure 6–6, the first character will have a code of 0 (referring to Table 6–1), the second a code of 1, the third 2, and the fourth 3:

$$\begin{aligned} MEM \Leftarrow\ & (\{\texttt{get}\}\,\{\texttt{data}\triangleleft\texttt{0}\}\,\{\texttt{get}\}\,\{\texttt{data}\triangleleft\texttt{1}\}\,)^2 \\ & (\{\texttt{get}\}\,\{\texttt{data}\triangleleft\texttt{2}\}\,\{\texttt{get}\}\,\{\texttt{data}\triangleleft\texttt{3}\}\,)^2\ MEM \end{aligned}$$

The description could be further refined to input **line** and **vsync** events and in that way generate the addresses, but this expression is clearer and has the same effect.

6.3.4 Simulating the Controller

The controller itself just generates all the timing signals that were defined above. It can therefore be specified by the composition of these definitions:

$$\begin{aligned} CONTROLLER \Leftarrow\ & VSYNC \\ & \bullet\ HSYNC \\ & \bullet\ CHAR\_RATE \\ & \bullet\ LINE\_RATE \\ & \bullet\ FETCH \end{aligned}$$

The *REFRESH* definition was not included because a real controller will input the fastest clock (`dotclk`) and divide it to produce the others. *REFRESH* simply allows us to derive the correct clock rate from the refresh requirements of the video monitor.

A complete display can be produced by connecting the controller to the screen memory and the character generator:

$$DISPLAY \Leftarrow CONTROLLER \bullet CHAR\_GEN \bullet MEM$$

Since the screen memory is not being changed by any outside source, *DISPLAY* forms a self-contained system and can be simulated without the need to supply input stimuli. Expanding *DISPLAY* using law [• +] produces a sequence that looks like this:

{`vsync hsync char dotclk`} {`video`◁0} {`dotclk`} {`video`◁0}
{`char dotclk`} {`video`◁0} {`dotclk`} {`video`◁0} ...

Only the behavior of the `video` channel is suitable for display as a waveform since it outputs bit data. The others are either pure events (`hsync`, `dotclk`, and so forth) or output non-bit values (`data`). An alternative representation called a ***sequence diagram*** is used to display the simulation results so that they can be easily understood. Time runs from the top of the page downward with each channel having its own time line. Instead of transitions of waveforms, events on the channel (including the passing of data values) are indicated by solid circles (●). Events that may occur at any time during a particular interval (as, for example, might happen if the π operator is used) are represented by an oval (▭) that covers the interval. Note that the oval implies that one event happens during that interval, and not many as it may appear. Using these conventions, the results of simulating *DISPLAY* are depicted in Figure 6–7.

Some points to notice are that (1) Video is blanked (low values are on the right of the waveform) during horizontal and vertical retrace as required. (2) The points at which `get` and `data` occur are constrained only by the `char` events and can therefore interleave with the `dotclk`events. That is why they are portrayed by ovals. Notice that the data arrives at least one dot time before it is displayed, even though there is a delay in fetching it from memory.

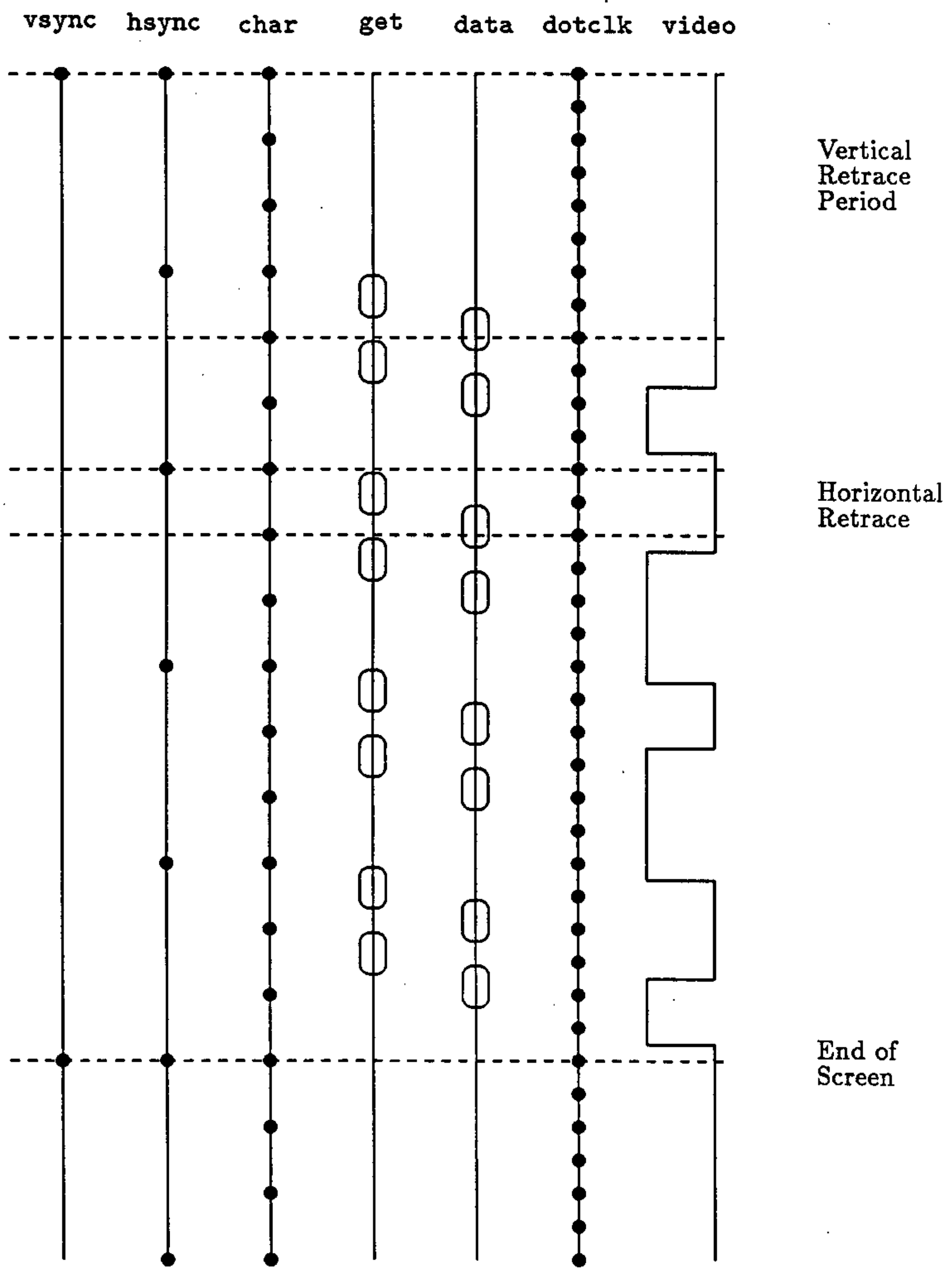


Figure 6–7: Results of simulating *DISPLAY*.

6.4 Adding a Line Buffer

Simple CRT controllers, such as the one presented here are quite greedy in their use of the system bus. They need to have data delivered frequently and at precisely the right time. Examining the behaviors of the data channel (which corresponds to the system data bus) and the get channel (loosely related to the address bus) in Figure 6–7, we can see that there is almost continuous activity, except during the vertical retrace time. A microprocessor sharing the bus would have only this interval for long uninterrupted memory access, such as might be needed to scroll the screen memory. In our small example, the percentage of time that the bus is free is quite high, but this would lower dramatically for real controllers which must typically display eighty characters per line.

The reason that bus utilization is so high is that every character must be fetched, not just once, but once for each scan line. If a buffer is used to retain one line of text, bus accesses would be cut down by the number of scan lines per character (two in this case). The buffer must be circular to present each character in the line in turn at one scan line and then again in the same order at the next. Here is the description for such a loadable circular buffer:

$$\begin{aligned}
BUF(c_1,c_2) &\Leftarrow \{\mathtt{hsync\ char}\}\,(\{\mathtt{data}\triangleleft c_2\}\,BUF1(c_2,c_1) \\
&\qquad + \{\mathtt{buf}\triangleright x\}\,\{\mathtt{data}\triangleleft x\}\,BUF1(x,c_1)) \\
BUF1(c_1,c_2) &\Leftarrow \{\mathtt{char}\}\,(\{\mathtt{data}\triangleleft c_2\}\,\{\mathtt{char}\}\,BUF(c_2,c_1) \\
&\qquad + \{\mathtt{buf}\triangleright x\}\,\{\mathtt{data}\triangleleft x\}\,\{\mathtt{char}\}\,BUF(x,c_1))
\end{aligned}$$

The buffer loads values received on the buf channel and passes them through to the data channel at the same time. This allows a line to be displayed while it's being loaded, thus simplifying the timing requirements by not requiring earlier fetch times. The buffer is not shifted during the character time before an hsync, in order that the first stored character code will be output during the horizontal retrace, hence the two states.

With the buffer inserted between the screen memory and the character generator, the memory fetch times must be changed:

$$\begin{aligned} \mathit{FETCH} \Leftarrow\ & \{\texttt{vsync char}\}\,\{\texttt{char}\}\,\{\texttt{char}\}\,\mathit{FETCH} \\ & + \{\texttt{line char}\}\,\{\texttt{get}\}\,\{\texttt{char}\}\,\{\texttt{get}\}\,\{\texttt{char}\}\,\mathit{FETCH} \\ & + \{\texttt{char}\}\,\mathit{FETCH} \end{aligned}$$

The only difference is that characters are fetched at the beginning of a line, rather than at every scan line. With this change, *MEM* must also be modified, both to output the correct values and to do so on buf. The buf channel will therefore correspond to the system bus.

$$\begin{aligned} \mathit{MEM} \Leftarrow\ & \{\texttt{get}\}\,\{\texttt{buf}\triangleleft 0\}\,\{\texttt{get}\}\,\{\texttt{buf}\triangleleft 1\} \\ & \{\texttt{get}\}\,\{\texttt{buf}\triangleleft 2\}\,\{\texttt{get}\}\,\{\texttt{buf}\triangleleft 3\}\,\mathit{MEM} \end{aligned}$$

The system is expanded to include the buffer:

$$\mathit{DISPLAY}' \Leftarrow \mathit{CONTROLLER} \bullet \mathit{CHAR\_GEN} \bullet \mathit{BUFFER} \bullet \mathit{MEM}$$

and the same simulation is run again with the results shown in Figure 6–8.

Comparing the output of the data channel in Figure 6–7 with that of the buf channel in Figure 6–8 reveals that the bus access time has been cut in half by the addition of the buffer. The saving becomes even more dramatic as the number of scan lines per character increases.

This example has shown that experimenting with a design—even in a crude form—can reveal valuable information about performance. Now that the experiments have indicated that a line buffer might be a desirable feature, the next step would be to implement the various parts making up *DISPLAY* and verify that they meet the requirements outlined above. Implementing and verifying the initial version of the controller would not have revealed the improvements brought about by the buffer, thus illustrating the value of high level experimentation.

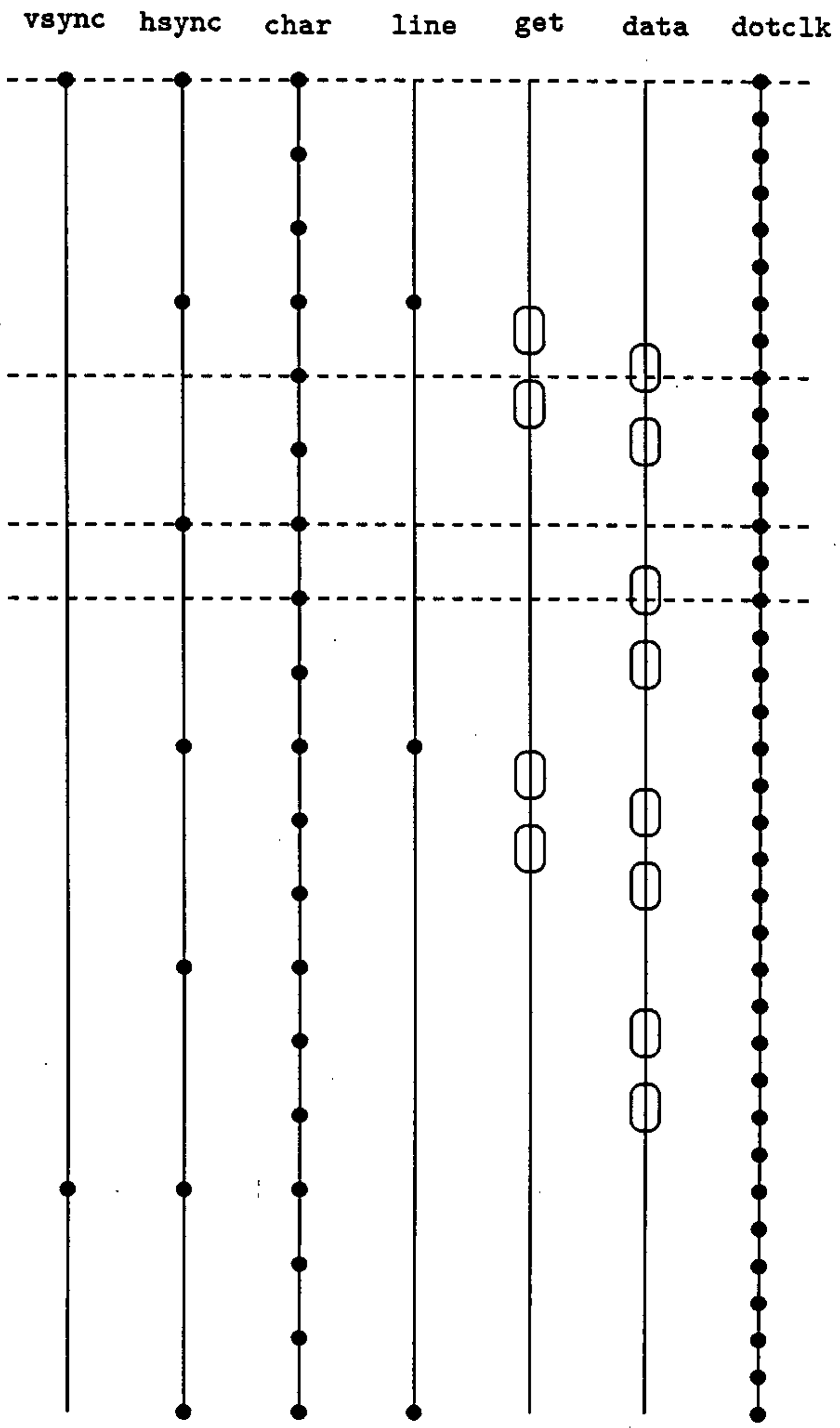


Figure 6–8: Adding a line buffer to *DISPLAY*.

Chapter 7

Discussion

Three desirable components of a hardware verification system were identified in the introduction. These were:

1. A Description Language

2. A Method for Verifying Designs

3. A Method for Experimenting on Designs

The rest of the thesis went on to consider each in detail using a formal calculus as a base.

Chapter 2 introduced the calculus—the CIRCAL calculus of Milne—and described its core operators. Grey areas in the definitions of a few operators were identified and solutions proposed. These included the passing of values along channels between processes for which new notation and composition rules were given. Also considered were interpretations of diverging processes (those that engage in infinite internal actions), which were equated with the deadlock operator. This interpretation became particularly important when the concept of explicit time was introduced in a later chapter, since it gives the behavior of a system that passes time without contact with the environment.

Once the core operators were presented, several others were derived from them for capturing certain common hardware constructions. This was followed by a discussion on packaging collections of equations that describe a single object for ease of manipulation. The result was the part construct. It was noticed that for many combinational elements, function can be separated from the structure of communications with the environment. Based on this observation, the idea of generic boxes that can be personalized with a particular function was introduced. The chapter concluded by considering various ways of representing physical phenomena in the calculus and then evolving a design style. Altogether, the material covered forms the Description Language component of a hardware verification system.

Chapter 3 started off by describing a way of using the calculus to model discrete time due to Milne and then went on to extend it in several directions. It was shown that hierarchies of system clocks could be easily represented and a special notation was introduced to define their interrelationships. Similar notation was used to model the hardware phenomena of clock skew and clock drift. Other operators were defined to convey the notion of temporal ambiguity of events *i.e.*, events that can occur either simultaneously or in any sequential order. Using these operators, the idea of clock intervals was developed to indicate that a set of events may happen at any time during the interval delimited by two others. This concept is particularly important in high level specifications where implementation dependent details of sequence are meaningless. To make the interval concept more useful, additional equational laws must be derived so that they can be manipulated without continual recourse to their definition. In addition, it would be advantageous to establish a link with work on Interval Temporal Logic [Moszkowski 83] so that some of its powerful temporal concepts may be exploited.

Delay was identified as an important factor in the temporal behavior of digital systems. Accordingly, several types of delay were characterized using the discrete model of time and their physical causes discussed. The chapter closed by considering the effects of delay on feedback circuits, as represented by a latch constructed from two cross-coupled NOR gates. Simple experiments on the implementation revealed that it suffers from metastability problems under certain input conditions. It might be desirable to categorize what types of feedback are prone to these problems and see if they can be recognized from their CIRCAL expressions.

In Chapter 4, the first analytic techniques were introduced. A notion of simulation/experimentation, originally due to Milne, was derived from basic properties of the Dot composition operator. We saw how the same technique yields symbolic simulations without extra effort, whereby the outputs of a module can be determined as a function of its input values. Another idea of Milne's called ***constructive simulation*** also derives from properties of the Dot Operator. With this approach, the result of a simulation is constructed using the same hierarchy as the target design, potentially increasing the efficiency of an execution immensely. Problems with the mechanics of conducting such simulations were revealed and simple solutions proposed. Further work needs to be done on implementing these techniques on parallel processors in order to exploit the gains in simulation efficiency fully.

After discussing the constructive approach, attention was turned to a way of simulating a specification and an implementation in parallel. It was shown that such a simulation would reveal any difference between the two through the action of the Dot Operator alone, thus taking a step toward true verification. Finally, several constructs were developed for analyzing the output of simulations. They were categorized by the manner in which they could be attached to the system under test and the effect they have on output values. A simple, but effective, example was a spike detector which signalled an error when an illegal transient value appeared on a line. This completed support for the third area required for a verification system: a Method of Experimentation.

Chapter 5 considered the second and last component of a verification system: a Method of Verification. Actually several methods were presented, each with its own strengths and weaknesses. Proof of equivalence using algebraic and transformation methods was shown to be inadequate because a specification will often include only a subset of the behavior of a valid implementation. To get around this limitation, the idea of a *partial specification* was introduced along with a means of showing that it is *satisfied* by an implementation. Although a step in the right direction, the idea of partial specifications turned out to be flawed when it was noticed that implementations could interact illegally in ways not covered by their specifications. To rectify this, constraints were placed on their usage. Two ways of generating and propagating these constraints were given, namely Annotated Covering Trees and Safe Contexts. Several laws were given for manipulating the former and others should be derived. Safety was defined only for *compositional* contexts and work remains to be done on generalizing it to all contexts. The limitation did not prove too obstructive since most designs are built by composing smaller blocks. Additional equational laws would ease the burden of checking if constraints are satisfied, as would the verification of common contexts. An example of such a frequently encountered context is the hole between two registers in a register transfer system. The clock period of the registers would eliminate most transient values produced by any process that fills the hole, as long as the period is greater than the maximum delay introduced by the process. This is much simpler to verify than proving the process correct in the complete context.

The last chapter dealt with several larger examples. These demonstrated the techniques presented previously and showed how various types of behavior could be expressed in CIRCAL. A wide variety of problems were covered, ranging from ways of dealing with transient values, through generating random numbers, to modelling hierarchical clocks. They demonstrate that the calculus has a surprisingly wide range of application. Further examples will reveal additional techniques and derived operators.

7.1 Further Ideas

Perhaps the most important requirement that came across when working the larger examples in Chapter 6 was the need for mechanized assistance in manipulating equations. Clever notation can go only just so far in alleviating the growth of equational complexity. Many of the expansions done during the course of the examples were checked using an equational rewrite system written in PROLOG [Clocksin 81]. The system is a successor to an earlier LISP based one [Traub 83] and uses some ideas from a functional extension to PROLOG described in [Newton 85]. It proved essential for exploring the feasibility of various communication structures and, because simulation is available "for free" by the application of law [• +], made the CRT controller example in Chapter 6 quite easy to conduct. The system works by rewriting equations as much as possible into a "standard form" using expansion laws such as [• +], [− +] and so forth. The standard form is a nondeterministic sum of deterministic summations:

$$P \Leftarrow \mathop{\textstyle\sum\!\!\!\!\bigcirc}_{i} \sum_{j} \gamma_{ij} P_{ij}$$

which is just the syntactic representation of a synchronization tree (Section 2.5). Further refinements could be made to sort guards with a lexical ordering so that two expressions can be textually compared.

The system is not nearly as adept as those of Gordon [Gordon 83a] and Barrow [Barrow 83] in simplifying functions, but it works quite well as a proof aid and as a means of investigating the ramifications of design decisions. It seems possible to extend the system so that it can derive covering contexts as discussed in Chapter 5, which would be a step toward a more comprehensive verification system. Another approach would be to formulate a simulation language around CIRCAL, as Moskowski has done with his Tempura language [Moszkowski 84] and Sheeran with her implementation of μFP [Sheeran 83]. With this method, the constructive and hierarchical techniques developed in Chapter 4 could be exploited to yield more efficient simulations.

Other areas merit further research as well. Some work was done on applying the calculus to low level circuit elements without much success. The elements were modelled as black-boxes that input and output current and voltage values as determined by their function (capacitor, resistor, etc.). At the electrical level, however, components cannot be considered in isolation since their behavior depends heavily on the dynamic context in which they find themselves. Simple composition operations, such as those done with the Dot operator, give way to complex relaxation processes that are difficult to characterize with acceptance semantics. Digital systems seek to limit most of this flexibility, so they are far more suitable for modelling in the calculus than are analog devices. Perhaps an approach similar to Cardelli's Analog Processes [Cardelli 82] can be derived which incorporates a notion of continuous values.

Even with this limitation, the surprisingly wide applicability of CIRCAL to many problems in hardware design shows that formal techniques hold a lot of potential. The primary impediment to large scale applications at the moment is the rapid growth in complexity of behavioral expressions. As machine assistance matures and new operators are defined to capture common concepts, the size of problems that can be considered will also grow. We hope that this growth will soon reach the level where real designs can be verified in detail, thus yielding safer and less costly products.

Appendix A

An Implementation of a Card Slice

The ~~third~~ second example of Chapter 6 (on page 180) concerned an implementation of a hardware card deck. The deck was implemented as a loop of "card slices," one for each card in the deck. Random choice was produced by shifting a single high value quickly through the loop until a button was released, which happens nondeterministically as far as the system is concerned. After several iterations, a suitable description of the behavior of a slice was determined and this reproduced here:

```
part SLICE(v) {in, out, next, picked: bit, clr, clk, ButUp, ButDwn} ⟨
    AVAIL(v)  ⇐ {clk} π({in▷x}, {out◁v}, {next◁v}) AVAIL(x)
                + {ButDwn} AVAIL(v)
                + {ButUp} (if  v = 1  then {picked◁1} BP
                                      else  AVAIL(v))
          BP  ⇐ {clr} {ButDwn} {picked◁0} AVAIL(1)
                + {ButDwn} PICKED
     PICKED   ⇐ ({clk} + {ButDwn} + {ButUp}) PICKED
                + {clr} {picked◁0} AVAIL(0)
                + π({out◁0}, ({in▷x} {next◁x})) PICKED
⟩
```

The part has two major states: one corresponding to the slice being available and shifting values from left to right, the other to the slice being picked and passing values through transparently. The intermediate state *BP* was introduced to allow the last available card in the loop to be reset without entering the *PICKED* state.

An auxiliary part was also specified that generated the correct signal for resetting the deck to an unpicked state:

$$CLEAR(\tilde{v}) \Leftarrow \text{ANY}(\texttt{picked}_i \triangleright v_i, \quad i = 1, \ldots, 52)$$
$$(\text{if } \wedge\tilde{v} \text{ then } \{\texttt{clr}\})\, CLEAR(\tilde{v})$$

The output is just the logical AND of the picked lines.

A final part was defined that determined the behavior of the button with respect to the system clock.

part $BUTTON$ {ButUp, ButDwn, clk} $\langle$

$$WAIT \Leftarrow \{\texttt{ButDwn}\}\{\texttt{clk}\}\, CLK$$
$$CLK \Leftarrow \{\texttt{clk}\}\, CLK + \{\texttt{ButUp}\}\, WAIT$$

$\rangle$

Figure A–1 shows one possible implementation of a card slice in terms of gate level elements. The information about whether or not the slice has been picked is stored on a JK flip-flop, while the D-type flip-flop implements the shifting action of the slice. Both are positive edge-triggered, meaning that information is transfered from the inputs to the outputs by a low to high transition on the clock line.

Informally, the implementation works like this. G1 passes the input values to FF1 when the slice has not yet been picked (the pb line is high). These are clocked through to G3 which passes them on to the next slice. When a save signal is generated (as a result of the button being released), FF2 samples the value being output by the slice. If the value is one then the flip-flop is set, indicating that the card has been picked. This disables G1 (pb is low) and forces FF1 to continually input a zero. G2 is enabled (p is high) and passes values directly from in to next. On the next clock tick, FF1 passes its high value to the next slice through G3 (the last slice's value being passed through G2 is always zero since there is only one active slice at a time). It then reads a zero from the disabled G1 and continues to signal this on out until the slice is reset. All the slices have been picked when their respective p lines go high.

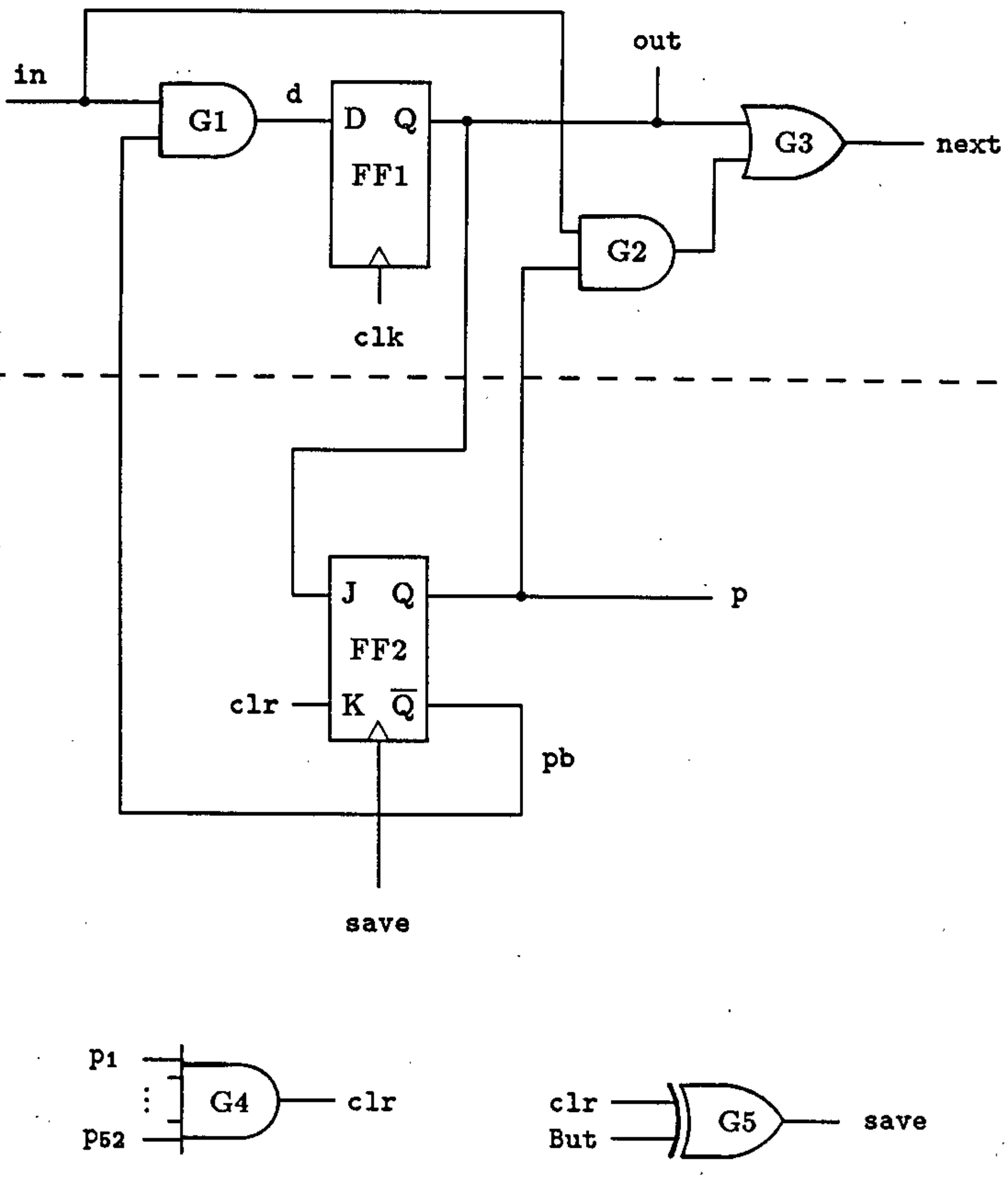


Figure A–1: Implementation of a card slice.

These are ANDed together to produce a clear signal (`clr`), which is fed to the K inputs of all the JK flip-flops, causing them to be cleared on the next **save** event. The last slice picked will still have its J input high, so K going high in turn means that the flip-flop will toggle when next clocked, resetting it to zero as well. The **save** event that is used to clock FF2 is normally produced by the button signal going low (*i.e.*, the button being released). However when all the cards have been picked, as indicated by the `clr` being high, **save** is generated by the button being <u>pressed</u>. The reason the controlling action changes is that a new deck must be made available <u>before</u> the riffling action commences again. If this were not done, all the slices would enter the picked state, resulting in deadlock. The `ButDwn` event always precedes the clock that causes the riffling, so it can be used to clear the FF2's. This should only happen when the deck is full, hence the button line is exclusive OR-ed with the `clr` line.

To show that this implementation meets the requirements of *SLICE*, we must first have some descriptions of the constituent parts. Since delays will effect the speed at which the circuit may be run, the behavior of each part will be given in terms of explicit time. A Universal Clock which emits $\mathtt{t_g}$ ticks a unit gate-delay apart is used to measure the passage of time. G1 therefore has a delay of one $\mathtt{t_g}$ tick:

$$
\begin{aligned}
&\textbf{part}\ \mathit{G1}(\tilde{w})\ \{\mathtt{in},\ \mathtt{pb},\ \mathtt{d},\ \mathtt{t_g}\}\ \langle \\
&\qquad \mathit{G1}(\tilde{w}) \ \Leftarrow\ \mathrm{WAIT}_{\mathtt{t_g}}\big(\mathrm{ANY}(\{\mathtt{in}\triangleright\ \mathtt{t_g}\},\ \{\mathtt{pb}\triangleright\ \mathtt{t_g}\}) \\
&\qquad\qquad\qquad\qquad \{\mathtt{d}\triangleleft(in \wedge pb)\quad \mathtt{t_g}\}\ \mathit{G1}(\tilde{w})\big) \\
&\qquad\qquad \tilde{w} \ = \ \langle in,\ pb\rangle \\
&\rangle
\end{aligned}
$$

Flip-flops are made up out of several gates internally, so they can be expected to have greater than unit delay. The D-type flip-flop is arbitrarily chosen to have an output delay of three ticks:

$$
\begin{aligned}
&\textbf{part}\ \mathit{FF1}(i)\ \{\mathtt{d},\ \mathtt{out}:\textbf{bit},\ \mathtt{clk},\ \mathtt{t_g}\}\ \langle \\
&\qquad \mathit{FF1}(i) \ \Leftarrow\ \mathrm{WAIT}_{\mathtt{t_g}}\big(\{\mathtt{d}\triangleright x\quad \mathtt{t_g}\}\ \mathit{FF1}(x) \\
&\qquad\qquad\qquad\qquad +\{\mathtt{clk}\quad \mathtt{t_g}\}\ \{\mathtt{t_g}\}^2\{\mathtt{out}\triangleleft i\quad \mathtt{t_g}\}\ \mathit{FF1}(i)\big) \\
&\rangle
\end{aligned}
$$

The two output gates, G2 and G3, are grouped together and modelled by a three-input generic box:

$$
\begin{array}{l}
\textbf{part}\ \mathit{G2G3}(\tilde{u})\ \{\texttt{in},\ \texttt{out},\ \texttt{p},\ \texttt{next}:\textbf{bit},\ \texttt{t}_\texttt{g}\}\ \langle \\
\qquad \begin{array}{rcl}
\mathit{G2G3}(\tilde u) & \Leftarrow & \text{WAIT}_{\texttt{t}_\texttt{g}}(\text{ANY}(\{\texttt{in}\triangleright\ \texttt{t}_\texttt{g}\}, \{\texttt{out}\triangleright\ \texttt{t}_\texttt{g}\}, \{\texttt{p}\triangleright\ \texttt{t}_\texttt{g}\}) \\
 & & \qquad\qquad \{\texttt{t}_\texttt{g}\} \\
 & & \qquad\qquad\quad \{\texttt{next}\triangleleft f(\tilde u)\ \ \texttt{t}_\texttt{g}\} \\
 & & \qquad\qquad\qquad\quad \mathit{G2G3}(\tilde u)) \\
\tilde u & = & \langle in, out, p\rangle \\
f(p, in, out) & = & (p\wedge in)\vee out
\end{array} \\
\rangle
\end{array}
$$

The part could be further refined into two cascaded two-input boxes—an OR gate and an AND gate.

The results of Section 6.1 say that parts attached to blocks of combinational elements must be prepared able to accept transient values before the correct one. If the next slice has not been picked, *G2G3* will be connected to a G1 gate forming a two element block which is in turn connected to a D-type flip-flop. The flip-flop can handle any input changes up until the clk event occurs and then must have stable inputs for a further three ticks. Thus, at the minimum, there will be three gates between the output of one flip-flop and the input of the next (two from *G2G3* and one from *G1*). So what is the maximum (worst case) delay? If all the slices have been picked except for one, the output of that slice's flip-flop must pass through $52 - 1 = 51$ *G2G3* parts and a G1 part before reaching its own D input. This is equivalent to a combinational block whose longest path is $51 \times 2 + 1 = 103$ gates long. The minimum possible delay D between clock events is then $103\,\texttt{t}_\texttt{g}$ plus the output delay of the flip-flop:

$$D = (103 + 3)\,\texttt{t}_\texttt{g} = 106\,\texttt{t}_\texttt{g}$$

which prompts the following constraint on the clock signal:

$$\mathit{C2} \ \Leftarrow\ \text{WAIT}_{\texttt{t}_\texttt{g}}(\{\texttt{clk}\ \ \texttt{t}_\texttt{g}\}\,\{\texttt{t}_\texttt{g}\}^{106}\,\mathit{C2})$$

This says is that when eventually a clk happens, there must be at least 106 ticks before the next one.

Returning to the analysis of the implementation, note that there are three cases to consider. The first is when the slice is available and shifting values through itself. The second is when the slice has been picked and acts simply as a delay. The third is the transition between these two steady states. Each case will be considered separately.

A.1 Case I: The Slice is Available

We must show that the portion above the dotted line in Figure A–1 functions in the same manner as the *AVAIL* state in the specification *SLICE*. In other words

$$T(v) \Leftarrow (\mathit{G1} \bullet \mathit{FF1} \bullet \mathit{G2G3}) - \{\mathtt{d},\ \mathtt{t_g}\}$$

is equivalent somehow to

$$\begin{aligned} \mathit{AVAIL}(v) \Leftarrow\ & \{\mathtt{clk}\}\,\pi(\{\mathtt{in}\triangleright x\}, \{\mathtt{out}\triangleleft v\}, \{\mathtt{next}\triangleleft v\})\,\mathit{AVAIL}(x) \\ & + \{\mathtt{ButUp}\}\,(\textbf{if } v = 1 \textbf{ then } \{\mathtt{picked}\triangleleft 1\}\,\mathit{BP} \\ & \qquad\qquad\qquad \textbf{else } \mathit{AVAIL}(v)) \end{aligned}$$

We do this by constraining T so that it reflects the steady-state condition of the slice being available. Such a condition means that the ButUp event can be ignored since it causes a change in state.

The first thing to notice is that if the slice is available, pb will be high and *G1* will act just like a unit delay. This is the steady-state condition alluded to above and is produced by placing the gate in a context that prevents any pb events from occurring:

$$\begin{aligned} \mathit{PB} \textbf{ of sort } \{\mathtt{pb}, \mathtt{t_g}\} &\Leftarrow \{\mathtt{t_g}\}\,\mathit{PB} \\ \mathit{PB} \bullet \mathit{G1}(in, 1) &= \mathrm{WAIT}_{\mathtt{t_g}}(\{\mathtt{in}\triangleright\ \mathtt{t_g}\}\,\{\mathtt{d}\triangleleft(in \wedge 1)\}\,(\mathit{G1}(\tilde{w}) \bullet \mathit{PB})) \\ &= \mathrm{WAIT}_{\mathtt{t_g}}(\{\mathtt{in}\triangleright\ \mathtt{t_g}\}\,\{\mathtt{d}\triangleleft in\}\,(\mathit{G1}(\tilde{w}) \bullet \mathit{PB})) \end{aligned}$$

Composing this simplified form of *G1* with *FF1* produces:

$$\begin{aligned} \mathit{PB} \bullet \mathit{G1} \bullet \mathit{FF1}(v) \ =\ & \\ & \mathrm{WAIT}_{\mathtt{t_g}}(\{\mathtt{in}\triangleright x\ \ \mathtt{t_g}\}\,(\ldots) + \{\mathtt{in}\triangleright x\ \ \mathtt{clk}\ \ \mathtt{t_g}\}\,(\ldots) \\ & \quad + \{\mathtt{clk}\ \mathtt{t_g}\}\,(\mathit{PB} \bullet \mathit{G1} \bullet \{\mathtt{t_g}\}\,\mathit{FF1}(v))) \end{aligned}$$

For this to match *AVAIL*, only the third branch can be used by the environment, so we have the following constraint:

$$C3 \;\Leftarrow\; \text{WAIT}_{\mathtt{t_g}}(\{\mathtt{clk}\ \mathtt{t_g}\}\,\{\mathtt{t_g}\}^2\,\text{WAIT}_{\mathtt{t_g}}(\{\mathtt{in}\triangleleft\ \mathtt{t_g}\}\,C3\,))$$

This is simply a timed version of constraint *C1*, given on page 193, that was needed during the derivation of *SLICE*. It said that an in must always follow a clk event. The timed version says that the input event will happen no sooner than two ticks after the clock tick, whenever one may occur. Each in channel is connected to the next channel of the previous slice and events on this channel are directly caused by the neighbor's out events which always happen two ticks after a clk, thus satisfying the constraint.

Applying *C3* to the gate and flip-flop combination yields:

$$\begin{aligned}
C3 \bullet PB \bullet G1 \bullet FF1(v) \;=\;& \\
&\text{WAIT}_{\mathtt{t_g}}(\{\mathtt{clk}\ \mathtt{t_g}\} \\
&\quad \{\mathtt{t_g}\}^2 \\
&\qquad [\{\mathtt{out}\triangleleft v\ \mathtt{t_g}\}\,(C3 \bullet PB \bullet G1 \bullet FF1(v)) \\
&\quad + \{\mathtt{out}\triangleleft v\ \mathtt{in}\triangleright\ \mathtt{t_g}\}\,(C3 \bullet PB \bullet \{\mathtt{d}\triangleleft in\ \mathtt{t_g}\}\,G1 \bullet FF1(v))])
\end{aligned}$$

Ignoring the $\mathtt{t_g}$ tick, this equation has much the same structure as *AVAIL*. The resultant, however, requires that in happen simultaneously or after out, whereas *AVAIL* specifies a full interleaving via the temporal permutation operator. Before resolving this discrepancy, let us see what happens when *G2G3* is added to the system.

The same steady-state condition that motivated the definition of *PB* also says that p will be low, thus making *G2G3* act like a two unit delay. The context process *PB* can easily be extended to include this extra information by simply adding p to its sort:

$$PPB \text{ of sort } \{\mathtt{p},\ \mathtt{pb},\ \mathtt{t_g}\} \;\Leftarrow\; \{\mathtt{t_g}\}\,PPB$$

Placing *PPB* in parallel with *G2G3* demonstrates the delay effect:

$$\begin{aligned}
PPB \bullet G2G3(out, in, 0) \;=\;& \text{WAIT}_{\mathtt{t_g}}(\text{ANY}(\{\mathtt{in}\triangleright\ \mathtt{t_g}\}, \{\mathtt{out}\triangleright\ \mathtt{t_g}\}) \\
&\quad \{\mathtt{t_g}\}\,\{\mathtt{next}\triangleleft out\ \mathtt{t_g}\} \\
&\quad (PPB \bullet G2G3(out, in, 0)))
\end{aligned}$$

Only communications on the out channel are passed with a delay. Signals on the in channel are simply absorbed, since the slice has not yet been picked (p is low).

Before composing *G2G3* with the flip-flop and input gate, let us simplify the expansion still further with another constraint. The behavior of *G2G3* is completely independent of the clock event, allowing next to interleave with it. *AVAIL*, however, does not allow this, so we require that:

$$C4 \;\Leftarrow\; \{\mathtt{clk}\}\,\{\mathtt{next}\}\,C4$$

The next channel is connected directly to the in channel of the succeeding slice via relabelling, so this constraint can be derived from *C3*. *C3* says that every in must happen after a clk, which means that next must also, since they will be the same channel. Here is how the expansion of the complete system looks—with simplifications—once the constraints have been applied:

$$\begin{aligned} T'(v) &\Leftarrow (PPB \bullet C3 \bullet C4 \bullet G1 \bullet FF1(v) \bullet G2G3) - \{\mathtt{d},\ \mathtt{t_g}\} \\ &= \{\mathtt{clk}\}\ \ (\{\mathtt{out}\triangleleft v\}\,(\{\mathtt{in}\triangleright x\}\,\{\mathtt{next}\triangleleft v\} \\ &\qquad\qquad + \{\mathtt{next}\triangleleft v\}\,\{\mathtt{in}\triangleright x\} + \{\mathtt{in}\triangleright x\ \ \mathtt{next}\triangleleft v\})T'(x) \\ &\qquad + \{\mathtt{in}\triangleright x\ \mathtt{out}\triangleleft v\}\,\{\mathtt{next}\triangleleft v\}\,T'(x)) \end{aligned}$$

Laws [rec$_3$] and [rec$_4$] were needed since hiding the passage of time (the WAIT$_{\mathtt{t_g}}$ operation) results in an unguarded recursion.

The only difference between this expression and the shifting portion of *AVAIL* is that instead of interleaving out and next, it indicates that they will occur sequentially. Provided that the environment cannot demand a different ordering, the two will be interchangeable. But will this requirement be met? To see, we must consider how the slice is to be used. Recall that the deck was made up out of a loop of slices with each in channel connected to the lefthand neighbor's next channel, as shown in the definition of *DECK* on page 195. For slices defined to act like *AVAIL*, the loop would allow the out events to happen in any order, with an effect similar to:

$$L \;=\; \{\mathtt{clk}\}\,\pi(\{\mathtt{out}_i\triangleleft\},\,\{\mathtt{in}_i\triangleleft\},\ i = 1\ldots 52)\,L$$

If the slices behave like T', on the other hand, the out events will all happen at exactly the same time, because each occurs two $\mathtt{t_g}$ ticks after the clock. The input events must occur after that because they are caued by next events which happen after {out} 's, with the following effect:

$$L' \;\coloneqq\; \{\texttt{clk}\}\,\{\texttt{out}_1\triangleleft\;\ldots\;\texttt{out}_{52}\triangleleft\}\,\{\texttt{in}_1\triangleleft\;\ldots\;\texttt{in}_{52}\triangleleft\}\,L'$$

The output channels of the loop are connected to the *ST_TO_CRD* transformer which can accept signals on them in any order (*i.e.*, will not distinguish between the L ordering and the L' ordering). No other process uses these channels, so we can conclude that T' will produce behavior similar to *AVAIL* when used with *ST_TO_CRD*.

Notice that to convince ourselves that the implementation was acceptable, we had to recourse to fairly high level knowledge about how the parts will be used. When using constraints it is quite often necessary to move back up through the hierarchy of design levels until they are resolved by some higher level knowledge. In the last step above, the difference in ordering on the output events between T' and *AVAIL* could have been resolved by constraining the environment to only accept the L' ordering since it is a subset of L. While valid, this constraint would not have been met by *ST_TO_CRD*, so we used the meta-knowledge to achieve a simpler result and saved having to redesign the transformer.

A.2 Case II: The Slice Has Been Picked

The next case to be considered is when pb is low and p is high. *G1* will be disabled, causing *FF1* to continually output a zero. *G2G3* will ignore this output and simply pass the value of in through unhindered with a delay of two t_g ticks. Since this is supposed to correspond to the card being picked, let us review the specification for that state:

$$\begin{aligned}\mathit{PICKED} \;\Leftarrow\;& (\{\texttt{clk}\} + \{\texttt{ButDwn}\} + \{\texttt{ButUp}\})\,\mathit{PICKED}\\ &+ \{\texttt{clr}\}\,\{\texttt{picked}\triangleleft 0\}\,\mathit{AVAIL}(0)\\ &+ \pi(\{\texttt{out}\triangleleft 0\},\,(\{\texttt{in}\triangleright x\}\,\{\texttt{next}\triangleleft x\}))\mathit{PICKED}\end{aligned}$$

Ignoring, for the moment, the transient ButUp, ButDwn and clr events, we see that the last choice branch is the one that determines the steady-state behavior of the part. Clock events do not change the state and are simply absorbed. We must show that this state is implemented by the top half of Figure A–1 with the controlling p and pb lines high and low respectively.

Much of the analysis is almost identical to the last case; the same *PPB* context is used to prevent the p and pb lines from changing. The difference lies in the values that are give initially to the state variables of the various parts. Starting from the left, *G1* should not contribute much to the behavior since it is disabled:

$$PPB \bullet G1(in, 0) \quad = \quad \mathrm{WAIT}_{\mathtt{t_g}}\big(\{\mathtt{in}\triangleright\ \mathtt{t_g}\}\ \{\mathtt{d}\triangleleft 0\ \mathtt{t_g}\}\ (G1(\tilde{w}) \bullet PPB)\big)$$

All that the disabled gate can do is while away the time, accept in communications and output 0 on the d channel.

The gate is combined with *FF1* to produce a process that absorbs in and clk events:

$$\begin{aligned}\big(C3 \bullet PPB \bullet G1 \bullet FF1(0)\big) - \{\mathtt{d},\ \mathtt{t_g}\} \quad = &\\ \{\mathtt{clk}\}\ (\{\mathtt{out}\triangleleft 0\}\ \{\mathtt{in}\triangleright\} + \{\mathtt{in}\triangleright\ \mathtt{out}\triangleleft 0\})&\\ \big((C3 \bullet PPB \bullet G1 \bullet FF1(0)) - \{\mathtt{d},\ \mathtt{t_g}\}\big)&\end{aligned}$$

Notice that despite values being input on in, the state never changes.

G2G3 can be simplified in exactly the same way as *G1*:

$$\begin{aligned}PPB \bullet G2G3(in, 0, 1) \quad = \quad & \mathrm{WAIT}_{\mathtt{t_g}}\big(\mathrm{ANY}(\{\mathtt{in}\triangleright\ \mathtt{t_g}\}, \{\mathtt{out}\triangleright\ \mathtt{t_g}\})\\ & \{\mathtt{t_g}\}\\ & \{\mathtt{next}\triangleleft f(in, 0, 1)\ \mathtt{t_g}\}\\ & (G2G3(in, 0, 1) \bullet PPB)\big)\end{aligned}$$

But $f(\tilde{u}) = (p \wedge in) \vee out = 1 \wedge in \vee 0 = in$ if we use the fact that *FF1* will always be outputting a 0 on out. The overall effect is to act as a two $\mathtt{t_g}$ tick delay on values sent on the in channel. Composing all the elements results in:

$$\begin{aligned}U(v) \quad \Leftarrow \quad & (PPB \bullet C3 \bullet C4 \bullet G1 \bullet FF1(v) \bullet G2G3) - \{\mathtt{d},\ \mathtt{t_g}\}\\ = \quad & \{\mathtt{clk}\}\ (\{\mathtt{out}\triangleleft 0\}\ \{\mathtt{next}\triangleleft in\}\ \{\mathtt{in}\triangleright x\}\ \{\mathtt{next}\triangleleft x\}\\ & + \{\mathtt{in}\triangleright x\ \mathtt{out}\triangleleft 0\}\{\mathtt{next}\triangleleft x\})U(x)\end{aligned}$$

It is only here that our casual pseudo event-driven approach was caught up with us. *FF1* and indeed the *SLICE* description itself output values in response to an input change regardless of whether that value is a true change on the line. This was done for simplicity and to reduce the size of the equations and has not been harmful until now. Barring the spurious {next◁*in*} event, *U* behaves identically to the *PICKED* state (with in not allowed to precede out), as required. The transient, however, invalidates a direct comparison between the two, even though it is just a re-signalling of the last change on the channel. This happens because the out◁0 event triggers *G2G3*, resulting in a next◁*in* event, where *in* is just the value last input on the in channel. To reconcile the two, a transformer could easily be attached to next that removes these redundent value signals. This is not as unrealistic as it seems, since the channel is connected to a *FF1* through a *G1* in the neighboring slice which can accept any number of events until a clock edge. All will be well provided that the transients have settled by the time the clock samples the input value. Such a step might be necessary anyway if *G2G3* is decomposed into two cascaded gates since the results of the first example in Chapter 6 warn us that transients should be expected.

A.3 Case III: Moving Between the States

For this case, we have to show that the circuitry below the dotted line in Figure A–1 together with the *BUTTON* part act to fulfill the *PPB* constraint used above to select the two steady-states. The constraint dictated what state the slice was in by setting the values of the p and pb lines. These lines are controlled by the JK flip-flop *FF2*, whose description is a bit more complicated than that of *FF1* since it has both two inputs and two outputs:

$$\textbf{part}\ \mathit{FF2}(out, clr, p)\ \{\texttt{out},\ \texttt{clr},\ \texttt{p},\ \texttt{pb}: \textbf{bit},\ \texttt{save}\}\ \langle$$

$$\begin{aligned}
\mathit{FF2}(\tilde{s}) \ &\Leftarrow\ \mathrm{WAIT}_{\mathtt{t_g}}\Big(\mathrm{ANY}(\{\mathtt{out}\triangleright\ \mathtt{t_g}\}, \{\mathtt{clr}\triangleright\ \mathtt{t_g}\})\ \mathit{FF2}(\tilde{s}) \\
&\qquad\qquad + \{\mathtt{save}\ \mathtt{t_g}\}\ \{\mathtt{t_g}\}^2\ \mathit{FF2}'(\tilde{s})\Big) \\
\mathit{FF2}'(\tilde{s}) \ &\Leftarrow\ (\textbf{if}\ \ h(\tilde{s}) \neq p\ \ \textbf{then}\ \ \circledR_D(\{\mathtt{p}\triangleleft h(\tilde{s})\ \mathtt{t_g}\}, \{\mathtt{pb}\triangleleft \bar{h}(\tilde{s})\ \mathtt{t_g}\})) \\
&\qquad\qquad \mathit{FF2}(out, clr, h(\tilde{s})) \\
h(out, clr, p) \ &=\ \textbf{case}\ \ out,\ clr\ \ \textbf{of} \\
&\qquad 1, 1: \ \ \bar{p} \\
&\qquad 1, 0: \ \ 1 \\
&\qquad 0, 1: \ \ 0 \\
&\qquad 0, 0: \ \ p \\
&\quad \textbf{esac}
\end{aligned}$$

$$\rangle$$

JK flip-flops complement their stored value if both inputs are high (in this case $out = clr = 1$) when the clock makes a transition. If this is not the case, the value of the p line will be set the same as the J input, in this case *out*.

Two further parts effect the state of the JK flipflop, namely G4 and G5. G4 is a fifty-two input AND gate with the same behavior as the *CLEAR* part described earlier:

$$\textbf{part}\ \mathit{G4}\ \{\mathtt{p}_i,\quad i = 1 \ldots 52,\ \mathtt{clr}: \textbf{bit},\ \mathtt{t_g}\}\ \langle$$

$$\begin{aligned}
\mathit{G4}(\tilde{p}, clr) \ &\Leftarrow\ \mathrm{WAIT}_{\mathtt{t_g}}\Big(\mathrm{ANY}(\{\mathtt{p}_i\ \mathtt{t_g}\},\quad i = 1 \ldots 52) \\
&\qquad\qquad (\textbf{if}\ \ \wedge\tilde{p} \neq clr\ \ \textbf{then}\ \{\mathtt{clr}\triangleleft\wedge\tilde{p}\ \ \mathtt{t_g}\}) \\
&\qquad\qquad\quad \mathit{G4}(\tilde{p}, \wedge\tilde{p})\Big)
\end{aligned}$$

$$\rangle$$

G5 calculates the exclusive-OR of the button and clear lines, but instead of outputting value changes it signals a low to high transition with the **save** event. The effect is that of connecting a transformer for converting **s◁1** events to **save** events to the output of a generic box personalized with the exclusive-OR function. This is makes the part compatible with *FF2* which is positive edge triggered and uses **save** as a clock line.

$$\textbf{part}\ \textit{G5}\ \{\texttt{clr},\ \texttt{But}:\textbf{bit},\ \texttt{save},\ \texttt{t}_\texttt{g}\}\ \langle$$

$$\begin{aligned} \textit{G5}(\tilde{b}, s) \ \Leftarrow\ & \mathrm{WAIT}_{\texttt{t}_\texttt{g}}\Big(\mathrm{ANY}(\{\texttt{clr}\triangleright\ \texttt{t}_\texttt{g}\}, \{\texttt{But}\triangleright\ \texttt{t}_\texttt{g}\}) \\ & \qquad (\textbf{if}\ (clr \text{ xor } But) = 1 \neq s\ \textbf{then}\ \{\texttt{save}\ \texttt{t}_\texttt{g}\} \\ & \qquad\qquad \textbf{else}\ \{\texttt{t}_\texttt{g}\}) \\ & \qquad \textit{G5}(\tilde{p}, (\ldots \text{xor} \ldots))\Big) \end{aligned}$$

$$\rangle$$

When **clr** is low, the **save** event will be generated by the button line going low (0 xor 1 = 0 and 0 xor 0 = 1 which is a low to high transition as is required to clock *FF2*). The button line going low happens when it is released (the ButUp event). To show the relationship between **save** and the button's behavior when **clr** is low, we can define a translation from the voltage changes caused by the button to ButUp and ButDwn events (ButDwn happening before ButUp as per the behavior of *BUTTON* given on page 215). This transformer is composed with *G5* and changes to the **clr** line are blocked to yield a process that shows **save** happening one tick after ButUp:

$$\begin{aligned} B \ &\Leftarrow\ \{\texttt{But}\triangleleft 1\ \ \texttt{ButDwn}\}\{\texttt{But}\triangleleft 0\ \ \texttt{ButUp}\}\,B \\ BG \ &\Leftarrow\ B \bullet \textit{G5}(0,0,1) \bullet \Delta_{\{\texttt{clr}\}} \\ &=\ \mathrm{WAIT}_{\texttt{t}_\texttt{g}}(\{\texttt{But}\triangleleft 1\ \texttt{ButDwn}\ \texttt{t}_\texttt{g}\}\{\texttt{t}_\texttt{g}\}) \\ &\qquad \mathrm{WAIT}_{\texttt{t}_\texttt{g}}(\{\texttt{But}\triangleleft 0\ \texttt{ButUp}\ \texttt{t}_\texttt{g}\}\{\texttt{save}\ \texttt{t}_\texttt{g}\})\ BG \end{aligned}$$

Conversely, when **clr** is high (the first state variable of *G5* is 1), the **save** event will be caused by ButDwn:

$$\begin{aligned} BG' \ &\Leftarrow\ B \bullet \textit{G5}(1,0,1) \bullet \Delta_{\{\texttt{clr}\}} \\ &=\ \mathrm{WAIT}_{\texttt{t}_\texttt{g}}(\{\texttt{But}\triangleleft 1\ \texttt{ButDwn}\ \texttt{t}_\texttt{g}\}\{\texttt{save}\ \texttt{t}_\texttt{g}\}) \\ &\qquad \mathrm{WAIT}_{\texttt{t}_\texttt{g}}(\{\texttt{But}\triangleleft 0\ \texttt{ButUp}\ \texttt{t}_\texttt{g}\}\{\texttt{t}_\texttt{g}\})\ BG' \end{aligned}$$

Figure A–2 shows the relationships between events on the But, clr and save lines. The idea is that the slice will usually change state when the button is released (save causes *FF2* to sample the out line). The exception is if this causes the clear line to go high (all the p lines are high) as would happen if the last slice were picked. All the slices must then be reset to the available state (the deck reshuffled) before clocking commences, which is signalled by ButDwn.

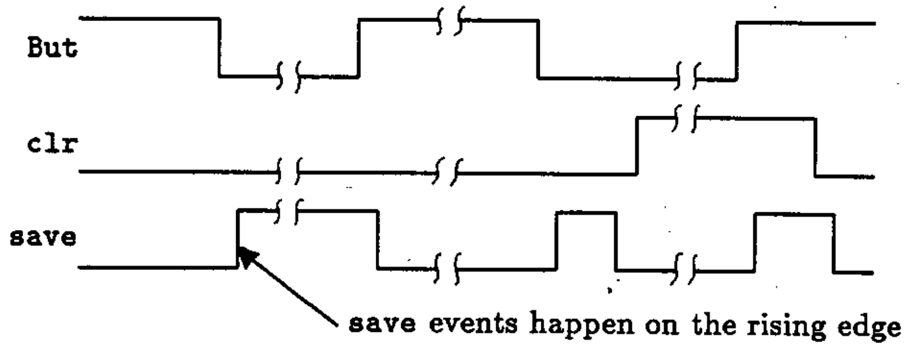


Figure A–2: Relationship between But, clr and save.

Choosing the correct point at which the button can be pressed or released with respect to the global clock is crucial for a clean transition between the two steady states. Button events happen one tick before save (as shown by *BG* and *BG′*) because of the delay through *G5*. The save event is used to clock *FF2* which should always have stable inputs when sampled. From the definition of *FF2*, we can see that for the J input to sample a stable out line, the save event must happen on the next tick after a value is output on that line at the very earliest. Events on the out channel happen on the third tick after a clk event, so save should not happen sooner then the fourth tick after a clock edge. The worst case therefore looks something like this:

$$\{\texttt{clk}\ t_g\}\,\{t_g\}\,\{t_g\}\,\{\texttt{But}\triangleleft\ t_g\}\,\{\texttt{save}\ t_g\}\ \ldots$$

Now we must determine the latest that the button can change before a clk. A save event might cause a change on the pb line, which is connected to the D input of *FF1* through *G1*. The D input should not change later than one tick before a clock edge, so adding the delays due to *G1* and *FF2* (one and four respectively) results in a requirement that save not happen later than five ticks before clk:

$$\{\texttt{save } \texttt{t}_\texttt{g}\}\,\{\texttt{t}_\texttt{g}\}^3\,\{\texttt{pb}\triangleleft\ \texttt{t}_\texttt{g}\}\,\{\texttt{d}\triangleleft\ \texttt{t}_\texttt{g}\}\,\{\texttt{clk } \texttt{t}_\texttt{g}\}\ \ldots$$

A But event happens one tick before this, giving a total of six ticks between the event and a clock edge at the minimum.

Together these worst case times form a window during which the button signal may safely change, as shown by the following constraint:

$$C5 \;\Leftarrow\; \text{WAIT}_{\texttt{t}_\texttt{g}}\Big(\{\texttt{But}\triangleleft\ \texttt{t}_\texttt{g}\}\,\{\texttt{t}_\texttt{g}\}^6 + \{\texttt{clk } \texttt{t}_\texttt{g}\}\,\{\texttt{t}_\texttt{g}\}^2\Big)$$

It says that no clk event may happen sooner than the seventh tick after the button has changed state and that this may not happen sooner than the third tick after a clock edge. No two But events can happen sequentially without a clk in between because of the behavior of the controlling *BUTTON* part defined on page 215.

C5 determines when the slice may safely make a state transition; the next step is to show that the correct states are entered. Since we are assuming that the slice is available, it must enter a state analogous to *BP* in the specification if the output is high when the button was released. From there it should move to the *PICKED* state (Case II above) unless clr goes high before the next ButDwn. If this does happen, no change should be made to the p and pb lines and the slice should remain in a state analogous to *AVAIL*(1) (Case I above). We shall consider each sub-case in turn.

The transition to the *PICKED* state should happen if the card is not the last in the deck (at least one other $\texttt{p}_i$ is low), is selected (out high), and is available (p low, pb high). Here is the state determination portion (parts below the dotted line in Figure A–1) initialized with these conditions and connected to the button stimulus *B* defined above:

$$\begin{aligned} V(p) \;&\Leftarrow\; [B \bullet C5 \bullet G4(\langle 0\ldots p\ldots 0\rangle, 0) \bullet G5(0,0,1) \bullet FF2(1,0,p)] \\ &\qquad \div \{\texttt{clk}, \texttt{out}, \texttt{p}_i, \quad i \in \text{others}\} \\ &= \; BG \bullet C5 \bullet (FF2(1,0,p) \div \{\texttt{clk}, \texttt{out}\}) \end{aligned}$$

The clk line (and hence out) are removed because we do not want the slice to shift its selected state on to the right while the button is held down. This is done purely for the purposes of allowing us to generate a ButUp event without danger of the slice deselecting itself. Similarly, we want none of the other p_i lines to change so all are hidden using $\div$. At least one of the other p_i lines is low, which means that *G4* cannot produce a clr event and therefore acts like the $\Delta_{\{\mathtt{clr}\}}$ in *BG* defined above.

The slice is available, so *V* starts out with the p line low ($p = 0$) and consequently should generate a save event when the button is next released. The p line will then go high to indicate that the card has been picked. Expanding $V(0)$ demonstrates this behavior

$$\begin{aligned}
V(0) &= \mathrm{WAIT}_{\mathtt{t_g}}\left(\{\mathtt{But}\lhd 1\ \mathtt{ButDwn}\ \mathtt{t_g}\}\,\{\mathtt{t_g}\}^6\, V'\right)\\
V' &= \mathrm{WAIT}_{\mathtt{t_g}}\Big(\{\mathtt{But}\lhd 0\ \mathtt{ButUp}\ \mathtt{t_g}\}\,\{\mathtt{save}\ \mathtt{t_g}\}\,\{\mathtt{t_g}\}^2\\
&\qquad \circledast_D(\{\mathtt{p}\lhd 1\ \mathtt{t_g}\}, \{\mathtt{pb}\lhd 0\ \mathtt{t_g}\})\{\mathtt{t_g}\}\, V(1)\Big)\\
V(1) &= \mathrm{WAIT}_{\mathtt{t_g}}\left(\{\mathtt{But}\lhd 1\ \mathtt{ButDwn}\ \mathtt{t_g}\}\right)\\
&\qquad \mathrm{WAIT}_{\mathtt{t_g}}\left(\{\mathtt{But}\lhd 0\ \mathtt{ButUp}\ \mathtt{t_g}\}\,\{\mathtt{save}\ \mathtt{t_g}\}\, V(1)\right)
\end{aligned}$$

Abstracting away $\mathtt{t_g}$, But, save, and pb and identifying p with picked should produce a behavior identical to the transition from *AVAIL*(1) through *BP* to *PICKED* in the specification *SLICE*:

$$V(0) - \{\mathtt{t_g}, \mathtt{But}, \mathtt{save}, \mathtt{pb}\} \;=\; \{\mathtt{ButDwn}\}\,\{\mathtt{ButUp}\}\,\{\mathtt{p}\lhd 1\}\,(V(1) - \{\ldots\})$$

The relevant portion of the specification looks like this:

$$\begin{aligned}
\mathit{AVAIL}(v) \;\Leftarrow\;& \quad \ldots\\
&+ \{\mathtt{ButDwn}\}\,\mathit{AVAIL}(v)\\
&+ \{\mathtt{ButUp}\}\,(\text{if } v = 1 \text{ then } \{\mathtt{picked}\lhd 1\}\,\mathit{BP}\\
&\qquad\qquad\qquad\quad \text{else } \mathit{AVAIL}(v))\\
\mathit{BP} \;\Leftarrow\;& \{\mathtt{clr}\}\,\{\mathtt{ButDwn}\}\,\{\mathtt{picked}\lhd 0\}\,\mathit{AVAIL}(1) + \{\mathtt{ButDwn}\}\,\mathit{PICKED}\\
&\vdots
\end{aligned}$$

The variable v is 1 because the slice is selected, so picked◁1 is produced as required. Following the implementation's transition into state $V(1)$, the values of the p and pb lines (high and low, respectively) will not change. This fulfills the conditions required for Case II above, which was the analogue of *PICKED* as required.

An almost identical analysis can be used to check the second sub-case: that when the slice is the last available, it will clear itself and return to the available state with the next ButDwn event. The initial conditions are the same, except that all the other p_i lines are high:

$$\begin{aligned} W(p) \;&\Leftarrow\; [B \bullet C5 \bullet G4(\langle 1\ldots 0\ldots 1\rangle, 0) \bullet G5(0,0,1) \bullet FF2(1,0,p)] \\ &\quad \div \{\mathtt{clk},\ \mathtt{out}\} \end{aligned}$$

Again the slice starts off unpicked ($p = 0$) and is expanded as before:

$$\begin{aligned} W(0) \;&=\; \mathrm{WAIT}_{\mathtt{t_g}}\big(\{\mathtt{But}\triangleleft 1\ \mathtt{ButDwn}\ \mathtt{t_g}\}\,\{\mathtt{t_g}\}^6\, W'\big) \\ W' \;&=\; \mathrm{WAIT}_{\mathtt{t_g}}\Big(\{\mathtt{But}\triangleleft 0\ \mathtt{ButUp}\ \mathtt{t_g}\}\,\{\mathtt{save}\ \mathtt{t_g}\}\,\{\mathtt{t_g}\}^2 \\ &\qquad \circledT_D(\{\mathtt{p}\triangleleft 1\ \mathtt{t_g}\}\,\{\mathtt{clr}\triangleleft 1\ \mathtt{t_g}\},\ \{\mathtt{pb}\triangleleft 0\ \mathtt{t_g}\})W''\Big) \\ W'' \;&=\; \mathrm{WAIT}_{\mathtt{t_g}}\Big(\{\mathtt{But}\triangleleft 1\ \mathtt{ButDwn}\ \mathtt{t_g}\}\,\{\mathtt{save}\ \mathtt{t_g}\}\,\{\mathtt{t_g}\}^2 \\ &\qquad \circledT_D(\{\mathtt{p}\triangleleft 0\ \mathtt{t_g}\}\,\{\mathtt{clr}\triangleleft 0\ \mathtt{t_g}\},\ \{\mathtt{pb}\triangleleft 1\ \mathtt{t_g}\})V(0)\Big) \end{aligned}$$

Notice that after the clr line goes low in W'', the system evolves to $V(0)$ because all the p lines will have gone low. The flipflop toggles when the button is pressed for the second time because its two inputs are high (*FF1*'s output and the clr line), thus clearing the output. All the other slices will have only their K lines high (corresponding to clr), so they too will be cleared.

Abstracting away $\mathtt{t_g}$, But, save, and pb as before reveals that $W(0)$ produces the same behavior as the fragment of the specification given above:

$$\begin{aligned} W - \{\mathtt{t_g},\ \mathtt{But},\ \mathtt{save},\ \mathtt{pb}\} \;&=\; \{\mathtt{ButDwn}\}\,\{\mathtt{ButUp}\}\,\{\mathtt{p}\triangleleft 1\}\,\{\mathtt{clr}\triangleleft 1\} \\ &\qquad \{\mathtt{ButDwn}\}\,\{\mathtt{p}\triangleleft 0\}\,\{\mathtt{clr}\triangleleft 0\}\,(V(0) - \{\ldots\}) \end{aligned}$$

With p low and pb high, we have the conditions for Case I. Since the slice's out cannot change until after the $\{\mathtt{clr}\triangleleft 0\}$ event (*i.e.*, at least seven ticks after the ButDwn), it will still be high when the slice becomes available, corresponding to the transition to *AVAIL*(1) in *BP*.

This completes the proof that the transition from the *AVAIL* state to *PICKED* and back again is handled correctly. The initial conditions and the *PPB* constraint that determined the previous two cases have been satisfied by the state determination portion of the implementation.

A.4 Summary

We have seen the essence of a proof that the circuit pictured in Figure A–1 on page 216 correctly implements the specification for a card slice developed in the second ~~third~~ example of Chapter 6.

Constraints were used liberally to simplify expressions as soon as possible. They allowed analysis of the implementation to be divided up into three fairly independent cases that parallel the operation of the slice. Some of the constraints were formulated with knowledge of how the slice will be used, which is defensible when one considers that this is part of a top-down design started in Chapter A–1. A more simple-minded verification could have been done by expanding the composition of all the constituent parts and trying to manipulate it into a form similar to that of the specification. It was felt, however, that the opaque juggling of symbols which this would have involved could serve no useful purpose and would probably have been very difficult to conduct.

Although the comparison between the specification of a card slice and its implementation was not presented in full detail, some important information about the circuits operating conditions was still discovered. The information takes the form of constraints that are not satisfied by other portions of the system and thus must be determined by the environment. Most important of these, perhaps, is the constraint that determines the maximum rate at which the clock may run:

$$C2 \;\Leftarrow\; \mathrm{WAIT}_{\mathtt{t_g}}(\{\mathtt{clk}\ \ \mathtt{t_g}\}\,\{\mathtt{t_g}\}^{106}\,C2)$$

The rate is given in terms of unit gate delays since that is smallest unit of time used. It works out to be 106 $\mathtt{t_g}$ ticks between clock events (positive edges of the clock waveform) at the very minimum.

The button is operated asynchronously by the user, so it is assumed that there is some circuitry to force its effects to happen at a known time. The circuitry must ensure that the clock starts no sooner than six ticks after the button is pressed and that the button is released no sooner than two ticks after a clock edge:

$$C5 \Leftarrow \mathrm{WAIT}_{\mathtt{t_g}}\left(\{\mathtt{But}\triangleleft\ \mathtt{t_g}\}\{\mathtt{t_g}\}^6 + \{\mathtt{clk}\ \mathtt{t_g}\}\{\mathtt{t_g}\}^2\right)$$

Figure A–3 shows these constraints in terms of waveforms.

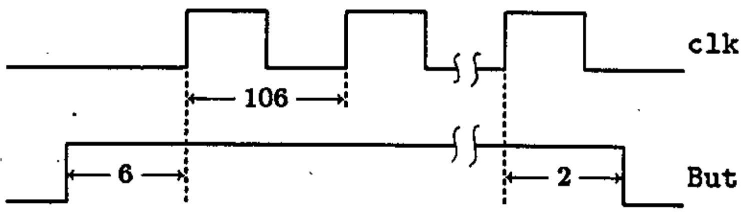


Figure A–3: Relationship between the button and clock signals (in units of $\mathtt{t_g}$)

Provided that the environment in which the card deck is to be used obeys these constraints, no unexpected behavior will happen. We have shown not only that our design is functionally correct, but have also determined the minimum operating conditions for it to remain so. These types of results are particularly important in digital designs where speed is critical, so it is encouraging to find that they can be derived using our constraint method.

Bibliography

[Anderson 77] E. Anderson and F. Belz.
Issues in the Formal Specification of Programming Languages.
In E. Neuhold, editor, *Formal Description of Programming Concepts*, pages 1–28, St. Andrews N.B., Canada, August 1977.

[Antognetti *et al.* 81] P. Antognetti, D. Pederson, and H. D. Man.
Computer Design Aids for VLSI Circuits.
Volume 48 of *NATO Advanced Study Institute Series*, Sijthoff and Noordhoff, 1981.

[Ayres 83] R. Ayres.
Silicon Compilation and the Art of Automatic Microchip Design.
VLSI, Prentice Hall, 1983.

[Backhouse 83] R. Backhouse.
Specification and Proof of a Regular Expression Recogniser in Synchronous CCS.
Internal Report CSR–130–83, University of Edinburgh, March 1983.

[Barrow 83] H. G. Barrow.
VERIFY: A Program for Proving Correctness of Digital Hardware Designs.
Technical Report, Fairchild Laboratory for Artificial Intelligence Research, November 1983.
Submitted to the journal Arificial Intelligence.

[Barrow 84] H. Barrow.
Proving the Correctness of Digital Hardware Designs.
VLSI Design, 5(7):64–77, July 1984.

[Bergmann 84] N. Bergmann.
A Case Study of the F.I.R.S.T. Silicon Compiler.
Internal Report CSR–159–84, University of Edinburgh, February 1984.

[Boehm *et al.* 84] B. Boehm, T. Gray, and T. Seewaldt.
Prototyping Versus Specifying: A Multiproject Experiment.
IEEE Transactions on Software Engineering, SE-10(3), May 1984.

[Breuer 72] M. Breuer.
Design Automation of Digital Systems.
Volume 1, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.

[Breuer 75] M. Breuer.
Digital Systems Design Automation: Languages, Simulation & Data Base.
Digital Systems Design Series, Computer Science Press, Inc., Woodland Hills, California, 1975.

[Breuer 76] M. Breuer and A. Friedman.
Diagnosis and Reliable Design of Digital Systems.
Digital System Design Series, Computer Science Press, 1976.

[Brookes 83] S. Brookes.
A Model for Communicating Sequential Processes.
Technical Report CMU-CS-83-149, Carnegie-Mellon University, January 1983.
Reprint of a PhD thesis submitted to Oxford University.

[Buchanan 80] I. Buchanan.
Modelling and Verification in Structured Integrated Circuit Design.
PhD thesis, University of Edinburgh, July 1980.

[Cardelli 81] L. Cardelli.
Sticks and Stones: An Applicative VLSI Design Language.
Internal Report CSR-85-81, University of Edinburgh, July 1981.

[Cardelli 82] L. Cardelli.
An Algebraic Approach to Hardware Description and Verification.
PhD thesis, University of Edinburgh, April 1982.

[Chen 83] M. Chen.
Space-Time Algorithms: Semantics an Methodology.
PhD thesis, California Institute of Technology, May 1983.

[Chen 84] C. Chen, C. Lo, H. Nham, and P. Subramaniam.
The Second Generation MOTIS Mixed-mode Simulator.

In *Proceedings of the 21st D.A.C.*, pages 10–17, IEEE, June 1984.

[Clocksin 81] W. Clocksin and C. Mellish.
Programming in Prolog.
Springer-Verlag, 1981.

[Coelho 84] D. Coelho.
Behavioral Simulation of LSI and VLSI Circuits.
VLSI Design, 42–51, February 1984.

[Cullyer 85] W. Cullyer and C. Pygott.
Hardware Proofs Using LCF–LSM and ELLA.
Memorandum 3832, Royal Signals and Radar Establishment, September 1985.

[De Man *et al.* 81] H. De Man, G. Arnout, and P. Reynaert.
Mixed Mode Circuit Simulation Techniques and their Implementation in DIANA.
In D. Antognetti, P. and H. D. Man, editors, *Computer Design Aids for VLSI Circuits*, pages 113–175, Sijthoff and Noordhoff, 1981.

[Deas 83] A. Deas.
The UNIT Silicon Compiler.
Internal Report CSR–145–83, University of Edinburgh, October 1983.

[Deas 84] A. Deas.
UNIT and LEGO: Database Languages for the U2 Silicon Compiler.
Internal Report CSR-180-84, University of Edinburgh, December 1984.

[deNicola 82] R. de Nicola and M. Hennessy.
Testing Equivalences for Processes.
Internal Report CSR–123–82, University of Edinburgh, August 1982.

[Foderaro 80] J. Foderaro and K. Sklower.
The FRANZ LISP Manual.
University of California, Berkeley, 1980.

[Friedman 77] A. Friedman and P. Menon.
Theory and Design of Switching Circuits.
Digital Systems Design Series, Pitman, 1977.

[Goldberg 84] A. Goldberg.
Smalltalk-80: The Interactive Programming Environment.
Addison-Wesley, 1984.

[Gordon 81] M. Gordon.
A Model of Register Transfer Systems with Applications to Microcode and VLSI Correctness.
Technical Report CSR–82–81, University of Edinburgh, March 1981.

[Gordon 83a] M. Gordon.
LCF - LSM.
Technical Report 41, University of Cambridge, 1983.

[Gordon 83b] M. Gordon.
Proving a Computer Correct.
Technical Report 42, University of Cambridge, 1983.

[Gordon 84] M. Gordon.
Multilevel Verification Using Higher Order Logic.
Lecture notes. Conference on Computer Hardware Description Languages, Darmstat, W. Germany, 1984.

[Gordon 85] M. Gordon.
HOL: A Machine Oriented Formulation of Higher Order Logic.
Technical Report 68, Cambridge University, May 1985.

[Hailpern 82] B. Hailpern.
Verifying Concurrent Processes using Temporal Logic.
Volume 129 of *Lecture Notes in Computer Science,* Springler-Verlag, 1982.

[Hanna 84] K. Hanna.
The VERITAS Project.
Lecture notes. Conference on Computer Hardware Description Languages, Darmstat, W. Germany, 1984.

[Hennessy 84] M. Hennessy.
Proving Systolic Systems Correct.
Internal Report CSR–162–84, University of Edinburgh, June 1984.

[Hoare 78] C. Hoare.
Communicating Sequencial Processes.
Communications of the ACM, 21(8):666–677, August 1978.

[Johnson 84] S. Johnson.
Synthesis of Digital Designs from Recursion Equations.
ACM Distinguished Dissertation, MIT Press, 1984.

[Kane 80] G. Kane.
CRT Controller Handbook.
OSBORNE/McGraw-Hill, Berkeley, CA., 1980.

[Kelly 84] V. Kelly.
The CRITTER System: Automated Critiquing of Digital Circuit Designs.
In *ACM IEEE 21st Design Automation Conference*, pages 419–425, IEEE Computer Society, June 1984.

[Kloos 86] C. Kloos.
Towards a Formalization of Digital Circuit Design.
PhD thesis, Technische Universität München, February 1986.

[Langdon 74] G. Langdon.
Logic Design; A Review of History and Practice.
ACM Monograph, Academic Press, 1974.

[Larsen 86] K. Larsen.
Context-Dependent Bisimulation Between Processes.
PhD thesis, University of Edinburgh, May 1986.

[Lattice 82] Lattice Logic Ltd.
Designing with Gate Arrays.
Lattice Logic Ltd., 3.2 edition, 1982.

[Marshall 83] R. Marshall.
A Compiler for Large Scale.
Internal Report CSR–150–83, University of Edinburgh, October 1983.

[McCarthy 82] O. McCarthy.
MOS Devices and Circuit Design.
Wiley-Interscience, John Wiley and Sons, 1982.

[Mead 80] C. Mead and L. Conway.
Introduction to VLSI Systems.
Addison-Wesley Series in Computer Science, Addison Wesley, 1980.

[Milne 77] G. Milne.
A Mathematical Model of Concurrent Computation.
PhD thesis, University of Edinburgh, 1977.

[Milne 80] G. Milne.
The Representation of Communication and Concurrency.
Technical Report, California Institute of Technology, September 1980.

[Milne 83a] G. Milne.
CIRCAL and the Representation of Communication, Concurrency and Time.
Technical Report, University of Edinburgh, 1983.
Also published in ACM Transactions on Programming Languages and Systems, Vol. 7, No. 2, April 1985.

[Milne 83b] G. Milne.
A Formal Basis for the Analysis of Circuit Timing.
Technical Report, University of Edinburgh, August 1983.

[Milne 83c] G. Milne.
A Simple Silicon Compiler and its Correctness.
In *Proc. 6th Int. Sym. on Computer Hardware Description Languages*, Pittsburgh, May 1983.

[Milne 84a] G. Milne.
A Model for Hardware Description and Verification.
In *ACM IEEE 21st Design Automation Conference*, pages 251–257, IEEE Computer Society, June 1984.

[Milne 84b] G. Milne.
Simulation and Verification Related Techniques for Hardware Analysis.
Internal Report CSR–174–84, University of Edinburgh, November 1984.

[Milner 80] R. Milner.
A Calculus of Communicating Systems.
Lecture Notes in Computer Science, Springer-Verlag, 1980.

[Milner 83] R. Milner.
Calculi for Synchrony and Asynchrony.
Theoretical Computer Science, (25):267–310, 1983.
Also published as Departmental Report CSR–104–82 of the University of Edinburgh, August, 1982.

[Mitchell 85] K. Mitchell.
Implementations of Process Synchronization and Their Analysis.

PhD thesis, University of Edinburgh, 1985.

[MM 79] G. Milne and R. Milner.
Concurrent Processes and Their Syntax.
Journal of the ACM, 26(2):302–321, April 1979.

[Morison 85] Morison.
The Design Rationale of ELLA.
In *CHDL–85*, IFIP, Tokyo, Japan, 1985.

[Moszkowski 83] B. Moszkowski.
Reasoning about Digital Circuits.
PhD thesis, Stanford University, July 1983.

[Moszkowski 84] B. Moszkowski.
Executing Temporal Logic Programs.
Technical Report 55, Cambridge University, August 1984.

[Nelson 81] G. Nelson.
Techniques for Program Verification.
Technical Report CSL–81–10, Xerox Palo Alto Research Center, June 1981.

[Newton 81] A. Newton.
Timing, Logic and Mixed-mode Simulation for Large MOS Integrated Circuits.
In D. Antognetti, P. and H. D. Man, editors, *Computer Design Aids for VLSI Circuits*, pages 175–241, Sijthoff and Noordhoff, 1981.

[Newton 85] M. Newton.
A Combined Logical and Functional Programming Language.
Master's thesis, California Institute of Technology, 1985.

[Nichols 83] K. G. Nichols.
Simulation From System Design to Chip Design.
In P. B. Denyer, editor, *Systems on Silicon*, pages 20–23, Institute of Electrical Engineers, September 1983.

[Prasad 84] K. Prasad.
Specification and Proof of a Simple Fault Tolerant System in CCS.
Internal Report CSR–178–84, University of Edinburgh, December 1984.

[Rem 81] M. Rem.

The VLSI Challenge: Complexity Bridling.
In J. Gray, editor, *VLSI 81*, pages 65–73, 1981.

[Rem 84] M. Rem.
Concurrent Computations and VLSI Circuits.
In M. Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming*, pages 399–437, NATO ASI, Marktoberdorf, W. Germany, August 1984.

[Rowson 80] J. Rowson.
Understanding Hierarchical Design.
PhD thesis, California Institute of Technology, April 1980.

[Sanderson 82] M. Sanderson.
Proof Techniques for CCS.
PhD thesis, University of Edinburgh, November 1982.

[Sheeran 83] M. Sheeran.
μFP — An Algebraic VLSI Design Language.
PhD thesis, Oxford University, November 1983.

[Shostak 83] R. Shostak.
Formal Verification of Circuit Designs.
In T. Uehara and M. Barbacci, editors, *Computer Hardware Description Languages and their Applications*, pages 13–30, IFIP, Pittsburgh, PA., May 1983.

[Subrahmanyam 83] P. Subrahmanyam.
Synthesizing VLSI Circuits from Behavioral Specifications: A Very High Level Silicon Compiler and its Theoretical Basis.
In F. Anceau and E. Aas, editors, *VLSI 83*, pages 195–210, IFIP, 1983.

[Traub 83] N. Traub.
A LISP Based CIRCAL Environment.
Internal Report CSR-152-83, University of Edinburgh, November 1983.

[Uehara 83] T. Uehara and M. Barbacci, editors.
Proceedings of the IFIP WG 10.2 Sixth Int. Symp. on Computer Hardware Description Languages and Their Applications, IFIP, North Holland, Pittsburgh Pennsylvania, May 1983.

[Unger 69] S. Unger.
Asynchronous Sequential Switching Circuits.
Wiley-Interscience, John Wiley and Sons, 1969.

[Valdimirescu 81] A. Vladimirescu, K. Zhang, A. Newton, D. Pederson, and A. Sangiovanni-Vincentelli.
SPICE Version 2G User's Guide.
Department of Electrical Engineering and Computer Science, University of California Berkeley, 1981.

[VanCleemput 79] M. Van Cleemput.
Computer Hardware Description Languages and their Applications.
In *Proceedings of the 16th Design Automation Conference*, pages 554–559, IEEE, June 1979.

[Winkel 80] D. Winkel and F. Prosser.
The Art of Digital Design.
Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1980.