Analysis of Hardware Descriptions

Satnam Singh

A Thesis
Submitted for the Degree of
Doctor of Philosophy
at the Department of Computing Science,
The University of Glasgow,
May 1991.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

The design process for integrated circuits requires a lot of analysis of circuit descriptions. An important class of analyses determines how easy it will be to determine if a physical component suffers from any manufacturing errors. As circuit complexities grow rapidly, the problem of testing circuits also becomes increasingly difficult.

This thesis explores the potential for analysing a recent high level hardware description language called *Ruby*. In particular, we are interested in performing testability analyses of Ruby circuit descriptions. Ruby is ammenable to algebraic manipulation, so we have sought transformations that improve testability while preserving behaviour.

The analysis of Ruby descriptions is performed by adapting a technique called *abstract interpretation*. This has been used successfully to analyse functional programs. This technique is most applicable where the analysis to be captured operates over structures isomorphic to the structure of the circuit. Many digital systems analysis tools require the circuit description to be given in some special form. This can lead to inconsistency between representations, and involves additional work converting between representations. We propose using the original description medium, in this case Ruby, for performing analyses. A related technique, called *non-standard interpretation*, is shown to be very useful for capturing many circuit analyses.

An implementation of a system that performs non-standard interpretation forms the central part of the work. This allows Ruby descriptions to be analysed using alternative interpretations such test pattern generation and circuit layout interpretations. This system follows a similar approach to Boute's system semantics work and O'Donnell's work on Hydra. However, we have allowed a larger class of interpretations to be captured and offer a richer description language.

The implementation presented here is constructed to allow a large degree of code

sharing between different analyses. Several analyses have been implemented including simulation, test pattern generation and circuit layout. Non-standard interpretation provides a good framework for implementing these analyses.

A general model for making non-standard interpretations is presented. Combining forms that combine two interpretations to produce a new interpretation are also introduced. This allows complex circuit analyses to be decomposed in a modular manner into smaller circuit analyses which can be built independently.

Acknowledgements

I am indebted to my supervisor, Dr. Mary Sheeran, for her patience, support and encouragement. Her undergraduate lectures on Ruby and VLSI design motivated my interest in this field. Prof. John Hughes introduced me to Miranda, which sparked off my interest in functional programming. My thesis has been an attempt to study both high level hardware description languages and implementation of these languages using functional programming.

This thesis has involved producing a large software system that runs over several different computers and operating systems. Many have helped me get to grip with some of the more obscure aspects of the Unix and Macintosh operating systems. In particular, Mark Dunlop has been very helpful.

I also wish to thank John O'Donnell for carefully reading the preliminary version of this thesis. John Launchbury provided much useful advice on how to formulate a good partial order for describing the D-Algorithm. Lars Rossen and Graham Hutton have made helpful remarks about the work on the D-Algorithm and testability transformations. Geraint Jones and Mary Sheeran have discovered simple but powerful decompositions for butterfly networks. These decompositions have made it very easy for me to make automatic layouts of butterfly networks.

This work was funded by the Science and Engineering Research Council of Great Britain. The Department of Computing Science at the University of Glasgow was a very fruitful place to work, and the functional programming group provided a useful forum for my ideas. The support staff have also been sympathetic to my bizarre system requirements. Finally, Susan Spence helped me in a variety of ways to complete this work.

Table of Contents

Chapter 1

Introduction

The design of an integrated circuit requires the use of many software analysis tools. Simulators can be used to check that the behaviour of the circuit corresponds with what was specified. Other tools are used to check that enough power is delivered to each part of the circuit, and others check that the timing behaviour of the circuit is correct. Testability analysis tools help generate test patterns and highlight areas of the circuit that are difficult to test.

All of these tools have to analyse some representation of the circuit. Current practice is for each of these analysis tools to use its own representation and notation for circuits. This requires translators to be written to convert between representations and can give rise to inconsistency between representations. Figure 1.1 (overleaf) shows the outline of a simple VLSI system that uses analysis tools which operate on a circuit description not supported by the design database. This problem would be avoided if these tools used the same representation. This also leads to a great deal of code sharing. The internal representation of a circuit and its associated operators need only be constructed once. They are then made available, perhaps as an abstract data type, to circuit analysis tools.

Of course, there are circuit analyses that operate on circuit descriptions at very different levels of abstraction for which a common circuit representation may not be possible. A good database system should keep track of the relationship between circuit representations at differing levels and present a data interface to CAD tools to allow them to operate over a wide spectrum of abstraction. A database system should also deal automatically with different file formats for the same information, presenting a transparent standard representation for CAD tools. However, this is an ideal which has not been achieved. Usually, a database system for a CAD product comprises a collection of files with very little management software.

This thesis presents a non-standard interpretation system, along with powerful operations that allow a large class of circuit analysis tools to be implemented. New

analysis tools are made by producing new functions that operate over circuit descriptions. It also implements a rudimentary database system which is used by the analysis tools to extract and submit information to and from design databases.

We note that analysis tools can be built more quickly and reliably if all circuit analyses operate over the same description. There are many ways of doing this, but in this thesis we concentrate on developing the technique of non-standard interpretation. Non-standard interpretation provides a natural framework for developing circuit analyses. The method works by redefining the semantics of leaf nodes in a circuit tree (or graph) description to yield a new analysis. Thus, the same structure is analysed, but with different (non-standard) semantics.

Non-standard interpretation is adapted for analysing high level hardware descriptions. The theoretical and practical aspects of this technique for analysing hardware descriptions are considered. This technique, along with the related technique of abstract interpretation has been used with great success to analyse computer programs, especially in functional languages [Peyton Jones 87].

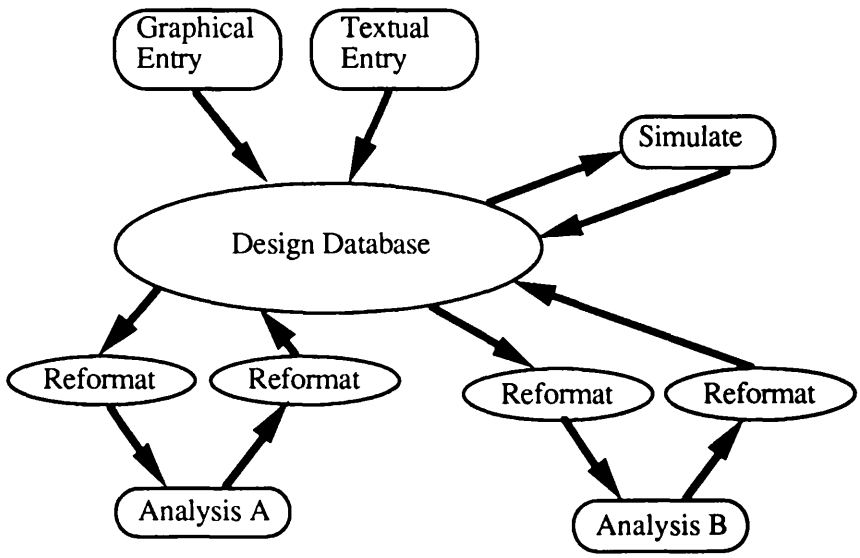


Figure 1.1: A simple VLSI CAD System

Figure 1.2 shows an abstract circuit which we shall use to outline the principle of non-standard interpretation. The logic elements are denoted by letters A to E and the wires are denoted by the numbers 1 to 10.

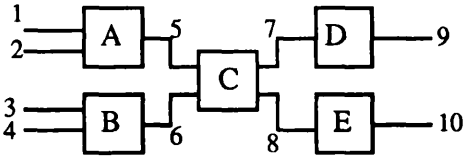


Figure 1.2: An abstract circuit.

Most circuit representations use a graph based approach to capture circuit connectivity. Logic elements are treated as nodes of a graph and the wires are treated as arcs. There is a small problem with primary inputs and outputs, but this can be dealt with by adding extra dummy nodes at the periphery of the circuit. Such a representation for the circuit above is shown in figure 1.3. The dummy nodes are shown as small shaded boxes.

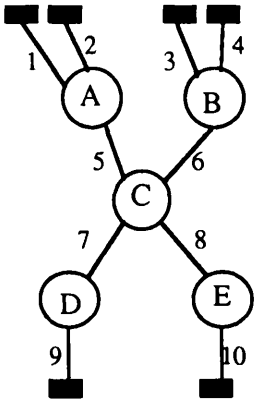


Figure 1.3: Graph representation of circuit in figure 1.2.

This graph can be used to simulate the circuit by thinking of the nodes as procedures that implement the actions of the circuit they model, and the arc values as representing parameter values that correspond to the signals on the wires. This graph is said to have an isomorphic structure to the circuit it represents: they are both the same shape. Non-standard interpretation essentially works by providing code for the nodes which performs some task other than simulation. The values on the arcs also change type, depending on the interpretation being performed. This gives a new circuit analysis which we are applying to exactly the same circuit structure that we used for specification and simulation. There is less likelyhood of inconsistency. When specifying a new analysis in this way, we do not have to write code to process the circuit representation since we share the code used by the simulation analysis.

We are particularly interested in developing testability analyses, and several are presented. The cost of testing a circuit is increasing as the sizes of circuits grow rapidly. The consequences of not testing circuits adequately can be grave. The analysis of

sequential circuits for testability is difficult, and non-standard interpretation seems to offer no advantages for analysing such circuits. Instead, we use non-standard interpretation to analyse combinational circuits only. We then present some transformations that improve the testability of a sequential design. Many of these transformations involve breaking the circuit into combinational blocks which can be tested separately from the sequential blocks. Thus, we combine a new method of analysis with successful traditional techniques for managing the testability of sequential digital systems.

We choose to analyse a rich algebraic hardware description language called Ruby. This is a relational language that allows regular synchronous digital circuits to be described succinctly. Ruby descriptions can be manipulated using existing laws about Ruby combining forms. Ruby also contains information about how circuits are laid out since it captures circuit structure. This is the key to the analyses that we present. Several tools have been implemented. Starting off with a simulator, we have used non-standard interpretation to build a testability measure tool, a fault simulator, test pattern generators, circuit layout tools and many others.

Non-standard interpretation is shown to be a good paradigm for capturing a wide variety of circuit analyses. This method promotes code re-use and modularity, and simplifies the implementations. This makes these tools easier to verify. Future work could involve building a proof system based around our non-standard interpretation system. This would check properties of circuit analysis tools to increase our confidence in their correctness.

Chapter 2 presents a brief introduction to the subset of Ruby that has been implemented by the author. This is a large subset, and is suitable for describing gate level circuits and arithmetic circuits. The standard meaning of Ruby is presented, giving behaviour as well as layout semantics.

An introduction to the problem of testability is presented in Chapter 3. We explain why circuits have to be tested and why this is a difficult problem. Various methods are proposed for analysing combinational circuits for testability, and some of these are implemented in later chapters by non-standard interpretation. Sequential circuits are dealt with by decomposing them into combinational sub-blocks which can be tested independently of the sequential components.

Some circuit analyses can be represented as abstract interpretations. Abstract interpretation is introduced in chapter 4, which shows how it has been used by functional programmers to analyse lazy functional languages. We present examples of circuit analyses that are abstract interpretations. Abstract interpretation supports the notion of safety which allows a particular abstraction to be proved correct with respect to the standard interpretation. This is very useful for proving the correctness of circuit analyses.

However, we show that abstract interpretation is not powerful enough to capture many of the testability analyses that we would like to do.

The more general but less disciplined technique of non-standard interpretation is introduced in chapter 5. Several methods for performing non-standard interpretation have been implemented. Some earlier methods are presented, along with one technique that we have settled on for making non-standard interpretations. One version allows only the outputs of circuits to be observed, while the other model allows internal nodes to be examined.

Having built non-standard interpretations, the next step is to show how they can be combined. The most useful way of making a new interpretation is to modify an existing one. Another way of combining two interpretations is to compose them in a serial manner. A symbolic simulator is given as an example non-standard interpretation.

Chapter 6 shows how two testability analyses can be cast as non-standard interpretations. The first is deductive fault simulation, which can be represented by one simple non-standard interpretation combined with the standard interpretation and a labelling interpretation. The second is SCOAP testability measure. This is expressed by using three interpretations: labelling, controllability measure and observability measure. Controllability is a forward interpretation in which information flows from the primary inputs to the primary outputs. Observability measure is a backward interpretation in which information flows from the primary outputs to the primary inputs. Since Ruby is a relational language with inverse, both types of information flows are dealt with easily.

The implementation is described in chapter 7. The architecture of the system is outlined. Actual test inputs and outputs for many interpretations are also presented. We show that for a relatively small amount of code, our system affords a very high degree of functionality. We also show how things would have been more difficult if the system was implemented in a powerful modern imperative language like Ada. Figure 1.4 represents a simplified view of the system architecture.

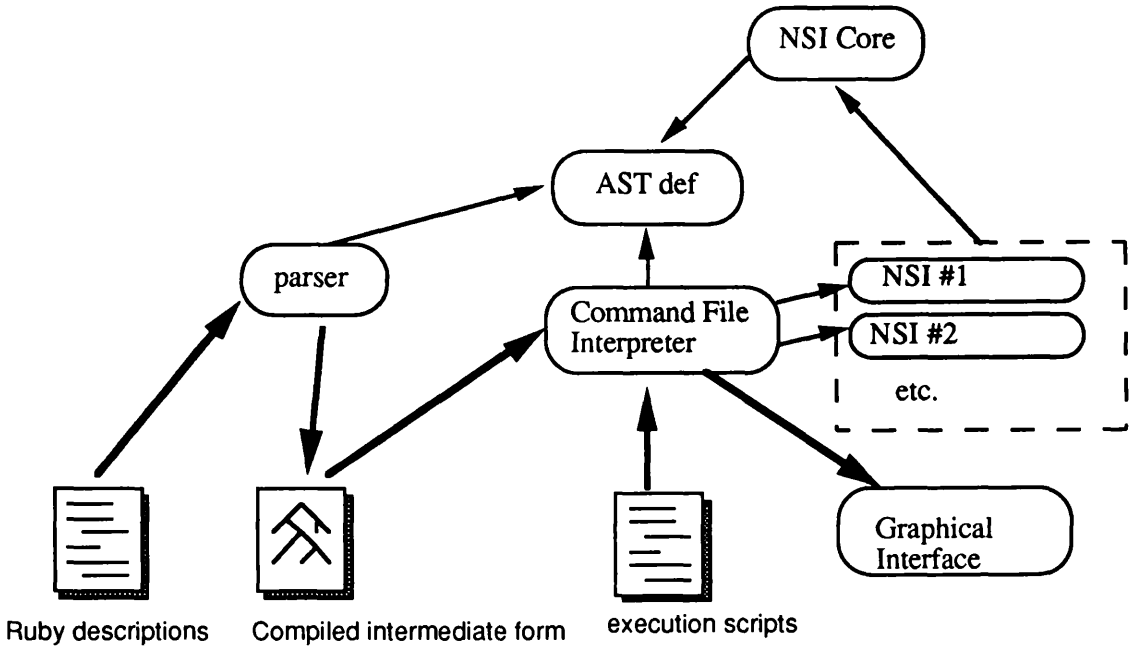


Figure 1.4: Simplified architecture of NSI system implemented.

One of the most important tasks carried out in testability analysis is test pattern generation. In chapter 8, we show that a popular automatic test pattern generation technique, called the D-algorithm, can be expressed as a non-standard interpretation. First, we re-express this complicated algorithm by giving a more formal description of each stage. The algorithm has complex backtracking information flow. We decompose the algorithm into several smaller problems which are easier to solve and have simpler information flow. Lazy evaluation is exploited to express backtracking implicitly, thus simplifying the algorithm implementation. This non-standard interpretation is then combined with a previously defined interpretation to produce a more efficient PODEM style test pattern generator.

Chapter 9 presents an interpretation for drawing butterfly circuits. The non-standard semantics employs functional geometry to help lay out these regular and recursive circuits. Several large butterfly and related network drawings are shown, all produced automatically by non-standard interpretation. Some of the pictures are drawn in colour to emphasise the wiring patterns involved. The ease with which these drawings were produced is convincing evidence in support of formal descriptions of high level circuits and non-standard interpretation as a circuit analysis.

Chapter 10 presents a set of transformations that improve the testability of circuits described in Ruby. We apply the traditional techniques for testability analysis, for example decomposing serial networks into parallel ones. We also try to spot replication so that large parts of the circuit can be tested in parallel, thus vastly reducing test effort. Transformations for dealing with sequential circuits are also presented. These involve

isolating the sequential and combinational parts of the design. The approach proposed is to use the non-standard interpreters presented in previous chapters to analyse combinational blocks and to apply scan path techniques to test chains of sequential components.

Chapter 11 concludes and discusses how non-standard interpretation is related to the objected-orientated notions of classes and inheritance. This is followed by an appendix listing the code for three non-standard interpretations. Finally, the references for the entire thesis are given.

Chapter 2

Describing Circuits Using Ruby

2.1 Introduction

The analysis techniques presented in later chapters operate on behavioural descriptions of circuits. For this reason it is important that we choose an expressive and powerful hardware description language. We also wish our analyses to take advantage of any repeated structure and hierarchy. For these reasons the relational hardware description language Ruby [Jones & Sheeran 90] has been chosen as a suitable high level representation.

Ruby is a powerful hardware description language which allows regular synchronous circuits to be described and manipulated easily. Only a subset of the language is presented here. The syntax of Ruby in the literature has been changed often. The notation introduced in this chapter shall be used consistently, although it may not correspond exactly with what others have written in Ruby.

Elementary combinational circuits are modelled by simple binary relations over tuples of boolean values. Larger combinational circuits are then composed using higher order combining forms like serial composition (relational forward composition) and parallel composition. Combining forms like serial and parallel composition also provide layout information.

Wiring circuits re-arrange wires without modifying the values being carried. A library of plumbing relations is introduced to allow the description of such circuits. The notion of lists of signals is used to help describe families of related wiring patterns e.g. zipping a 2-tuple of lists into a list of 2-tuples. Many of the wiring circuits have definitions very similar to tuple and list-manipulation functions found in most modern functional programming languages.

Streams are introduced to help describe sequential circuits. This method allows elegant description of sequential behaviour without using state variables. Ruby abstracts from explicit state and explicit time. This greatly simplifies the difficult task of reasoning about sequential circuits. The combinational primitives are extended to work over streams of values by spreading their combinational behaviour pointwise over signals of streams.

Other Ruby constructs for replicating circuits are also introduced. A convention for describing vertical as well as horizontal information flow is given. Finally, a small example of a Ruby design is presented.

Ruby is still being developed and has been used to describe a large variety of circuits e.g. butterfly circuits [Jones et. al. 90a] and FFT circuits [Jones 90]. Implementation work and type theory development is being carried out by Hutton [Hutton 90] and [Murphy 90]. Lars Rossen has implemented a large subset of Ruby in the Isabelle theorem prover [Rossen 90]. A more detailed description of Ruby can be found in [Jones et. al. 90b].

Although we have concentrated on Ruby, we could have used any hardware description language. Languages which provide powerful forms of composition are particularly suitable. The language Daisy [Johnson 83] would also have been a suitable candidate. This is functional in nature and is easy to manipulate. Johnson has shown how to synthesize designs in Daisy from recursion equations.

Hydra has been proposed by O'Donnell as a powerful hardware description language that can describe system at a behavioural or structural level of detail [O'Donnell 88]. It is easy to specify different parts of a system at different levels of abstraction. Hydra provides the designer with powerful tools like stream recursion equations, recursive circuit specification, functional geometry and higher order circuit combining forms.

2.2 Elementary Combinational Gates

Ruby describes the behaviour of a circuit by capturing the relation between the signals at the terminals of the circuit. Composite circuits can be described by composing relations. Circuits are designed using a library of ready built or elementary circuits which are combined to form larger circuits.

To describe combinational circuits, a suitably rich collection of elementary relations that implement logical operations is required.There must be enough relations to allow the description of any combinational circuit. The set of relations {NOT, AND, OR} is chosen to be the elementary set. This collection is *universal* i.e. combinations of these relations can describe any combinational circuit. Another suitable set is {NAND}. Although this set contains only one relation, the set {NOT, AND, OR} allows more natural definitions of many boolean expressions. The set of elementary relations is named *BASIC*.

$$BASIC = \{\text{NOT, AND, OR}\} \tag{2.1}$$

Extra elements can be added to this set when required. For example, to describe arithmetic circuits, it may be convenient to assume that a full-adder is an elementary relation. By using a suitably powerful set of combining operators, any boolean function can be described by composing the basic circuits.

Initially boolean algebra is used to describe the semantics of the gates in *BASIC*. One useful extension is to three valued logic (true, false and unknown). As usual, the boolean algebra possesses two values T (true) and F (false) and three logical operations: $\neg$ (logical negation), $\wedge$ (conjunction) and $\vee$ (disjunction).

Binary relations relate objects of one set to another. If X and Y are sets, then $X \leftrightarrow Y$ denotes the set of relations from X to Y. This may also be written as $\mathbb{P}(X \times Y)$ i.e. the powerset of the cartesian product of the sets X and Y. A binary relation on sets X and Y is a subset of $X \times Y$. Two elements $x \in X$ and $y \in Y$ are related to each other by a binary relation R if $(x, y) \in R$, where (x, y) is a 2-tuple or pair. We often abbreviate $(x, y) \in R$ to $x\ R\ y$.

A relation R from X to Y does not have to relate *every* object in X to Y. The subset of X that R does relate to Y is called the **domain** of R. The elements of Y that are related by R form the **range** of R. It is useful to define two functions to extract the domain (*dom*) and range (*rng*) from a relation R:

$$dom\ R = \{x :: X \mid \exists y :: Y \bullet\ x\ R\ y\} \tag{2.2}$$
$$rng\ R = \{y :: Y \mid \exists x :: X \bullet\ x\ R\ y\} \tag{2.3}$$

One straightforward definition of the basic relations is:

$$x\ \text{NOT}\ y \Leftrightarrow y = \neg x \tag{2.4}$$
$$\langle a, b\rangle\ \text{AND}\ c \Leftrightarrow c = a \wedge b \tag{2.5}$$
$$\langle a, b\rangle\ \text{OR}\ c \Leftrightarrow c = a \vee b \tag{2.6}$$

The behaviour of the NOT gate (definition 2.4) is specified by saying that the value at the domain must always be logically opposite to the value on the range.

AND relates a pair of boolean values to a single boolean value. The behaviour of AND is specified by stating the value at the range c is always the logical conjunction of the two values at the domain a and b. OR is defined in a similar manner.

The basic gates and combining forms have types associated with them which give the relation between the kinds of data that can appear at the domain and range. The type of a relation is specified by giving a type expression for the domain and the range. For example, a NOT gate has the type:

$$\text{NOT} \quad : \text{bool} \sim \text{bool} \tag{2.7}$$

Type names appear in lower case, and polymorphic types are denoted by lower case Greek letters. The types of the other two basic gates are:

$$\text{AND} \quad : \langle \text{bool}, \text{bool} \rangle \sim \text{bool} \tag{2.8}$$
$$\text{OR} \quad : \langle \text{bool}, \text{bool} \rangle \sim \text{bool} \tag{2.9}$$

A useful wiring circuit is split which duplicates a value. This is defined as:

$$a \ \text{split} \ \langle a, a \rangle \quad \Leftrightarrow \quad \text{true} \tag{2.10}$$

The universal quantifiers are omitted for polymorphic types.

$$\text{split} : \alpha \sim \langle \alpha, \alpha \rangle \quad \Leftrightarrow \quad \forall \alpha.\ \text{split} : \alpha \sim \langle \alpha, \alpha \rangle.$$

This says that split relates a signal on the domain, which may be of any type (call it α) to a pair of signals on the range. Each element of the pair is of type α. Thus, split is a polymorphic relation.

As an alternative specification, the basic gates are now given explicitly as sets of pairs. The first element of the pair is a value in the domain and the second element is a value in the range. The following sets may be used to define the behaviour of the combinational gates.

$$\text{NOT} = \{(F, T), (T, F)\} \tag{2.11}$$
$$\text{AND} = \{(\langle F,F \rangle, F), (\langle F, T \rangle, F), (\langle T, F \rangle, F), (\langle T, T \rangle, T)\} \tag{2.12}$$
$$\text{OR} = \{(\langle F,F \rangle, F), (\langle F, T \rangle, T), (\langle T, F \rangle, T), (\langle T, T \rangle, T)\} \tag{2.13}$$

These sets just encode the truth tables for the basic gates. Definitions 2.11—2.13 attribute the same behaviour to the basic gates as definitions 2.4—2.6.

Why are relations and not functions being used to describe hardware? In a real combinational circuit built with the above primitives, information flow is unidirectional. To answer the question "what is the output for a given input" it is sufficient to use functions. One reason for using relations is that many of the transformations on circuits depend on *connectivity* and not the direction of information flow. The use of relations abstracts from the direction of data flow, concentrating on the connectivity. This simplifies many algebraic laws about circuits. We shall later present some circuit analyses which have a backward flow of information e.g. the SCOAP testability measure.Such analyses are rendered more naturally in a relational notation. These advantages are covered in more detail in [Sheeran 88a]. Ruby's algebraic properties are exploited later to aid transformations that improve testability.

2.3 Composing Circuits

It was decided earlier not to use NAND as the elementary relation. This relation may of course be constructed using the relations AND and NOT. To make the NAND relation using these two elementary relations, the range of the AND relation is used as the domain of the NOT relation. The range of the NOT relation is used as the range value of the composite relation. When two circuits are composed by using the range of one as the domain of the other then we call such a composition a **serial composition**. The serial composition of an AND gate and a NOT gate is shown schematically in figure 2.1.

In addition to the semantics presented above, Ruby descriptions also have a geometric interpretation. The elementary components are drawn with the domain on the left and the range on the right. Composition is represented by juxtaposition as shown in figure 2.1. Wires are drawn from the bottom up that is the first element of a tuple is below all the others. Capturing structure is very important for the analyses we present. This is especially the case for testability analyses, where the faults to be tested for depend on the structure of the circuit.

The structural information in Ruby descriptions is also useful for fault models that consider the possibility of adjacent wires shorting. However, this information is not used by the analyses presented in later chapters.

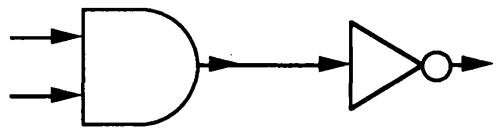


Figure 2.1: Serial composition of AND and NOT

The serial composition of two relations F and G is written as F ; G and is defined by:

$$a\,(F\,;G)\,c \quad \equiv \quad \exists b.\ (a\,F\,b)\ \&\ (b\,G\,c) \tag{2.14}$$

The infix ';' serial composition operator is associative. Bracketing can be omitted so that $A\,;B\,;C = A\,;(B\,;C) = (A\,;B)\,;C$.

The range type of F must be the same as the domain type of G for serial composition to be well typed. This relationship is expressed by the following type rule:

$$\frac{F : \alpha \sim \beta \qquad G : \beta \sim \chi}{F\ ;\ G : \alpha \sim \chi}$$

Serial composition is an example of a **higher order** combining form. It takes as parameters two circuits and yields a composite circuit.

The NAND relation built earlier may be expressed explicitly by the set:

NAND = AND ; NOT = {(⟨F,F⟩ T), (⟨F, T⟩, T), (⟨T, F⟩, T), (⟨T, T⟩, F)}
NAND : ⟨bool, bool⟩ ~ bool

The following example shows the result of 'simulating' the NAND gate with 'input' ⟨T, F⟩.

⟨T, F⟩ NAND x
≡ { def. NAND }

$\langle T, F\rangle$ AND ; NOT x
$\equiv$ { def. AND, def. ; }
F NOT x
$\equiv$ { def. NOT }
x = T

As expected, the input/output pair is a member of the defining set for the NAND relation: $(\langle T, F\rangle, T) \in$ NAND.

Serial composition is a natural way to combine two circuits which communicate information to each other. The communication occurs over the internal connection made by serial composition. It is also desirable to compose circuits which do not communicate with each other. One way to do this is by using **parallel composition.**

To demonstrate parallel composition, consider the specification of a circuit P defined by:

$$\langle x, \langle y,z\rangle\rangle \; P \; \langle a, b\rangle \quad \equiv \quad a = \neg x \;\&\; b = y \wedge z$$

Notice that a and b are not related. It is possible to relate x to a using NOT independently of relating $\langle y,z\rangle$ to b using AND. These two circuits may be placed in parallel in order to realise the specification of P by writing [NOT, AND].

Parallel composition of two circuits F and G is denoted by [F, G]. The type of the new circuit is defined in terms of the type of the constituent circuits:

$$\frac{F : \alpha \sim \beta \quad G : \chi \sim \delta}{[F, G] : \langle \alpha, \chi\rangle \sim \langle \beta, \delta\rangle}$$

Parallel composition of two relations is defined by independently relating the values on the terminals of the constituent relations.

$$\langle a, b\rangle \; [F, G] \; \langle c, d\rangle \quad \equiv \quad (a \; F \; c) \;\&\; (b \; G \; d)$$

The domain and range are defined in terms of the domain and range of constituent circuits:

dom F $\times$ dom G

Figure 2.2 shows a pictorial representation of [NOT, AND].

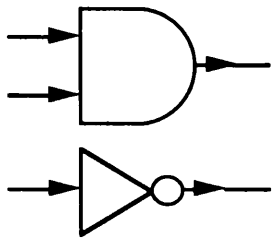


Figure 2.2: Parallel Composition [NOT, AND]

This new composite parallel circuit requires two values on its domain. These are represented by a tuple: the first element contains the value for the bottom circuit and the second element contains the value for the top circuit. The range is described similarly.

An example 'simulation' of [NOT, AND] is

$\langle T, \langle F, T\rangle\rangle$ [NOT, AND] $\langle x, y\rangle$
≡ {def. parallel composition }
T NOT x & $\langle F, T\rangle$AND y
≡ { def. NOT and AND }
x =F & y = F

As expected, $(\langle T, (F, T)\rangle, \langle F, F\rangle)$ is a member of the relation [NOT, AND].

Parallel composition extends naturally to compose more than two circuits, e.g

$$\langle a, b, c\rangle\,[P, Q, R]\,\langle d, e, f\rangle \;\equiv\; (a\,P\,d)\;\&\;(b\,Q\,e)\;\&\;(c\,R\,f)$$

There are many natural looking laws about parallel composition. For example:

[R, S] ; [T, U] = [R ; T, S ; U]

2.3 Relational Inverse

The **inverse** of a relation is defined as

$$a\,R^{-1}\,b \;\equiv\; b\,R\,a$$

$$\frac{R : a \sim b}{R^{-1} : b \sim a}$$

So what does it mean to talk about the inverse of a circuit? Ruby interprets this as flipping over the circuit along a vertical axis. The domain is then on the left and the range is on the right. Figure 2.3 shows the inverse of AND i.e. AND^{-1}.

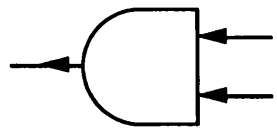


Figure 2.3: Inverse of AND.

Flipping a circuit over twice leaves it unaltered:

$$(R^{-1})^{-1} = R$$

The inverse of a serial circuit is formed by flipping the whole circuit so that the leftmost and rightmost circuits of the composition are swapped:

$$(R ; S)^{-1} = S^{-1} ; R^{-1}$$

As an example, consider the inverse of the NAND gate defined earlier:

$$(AND ; NOT)^{-1}$$
$$= NOT^{-1} ; AND^{-1}$$

The layout for this circuit is shown in figure 2.4.

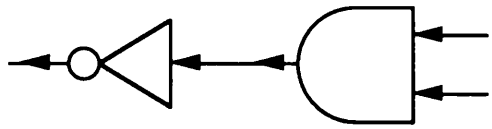


Figure 2.4: $(AND ; NOT)^{-1} = NOT^{-1} ; AND^{-1}$

The inverse of a NAND gate expects a 2-tuple from the right and delivers a single logic value at the left.

The inverse of a parallel composition of circuits $C_1,..C_n$ is simply the parallel composition of the inverse of the constituent circuits i.e. $C_1^{-1},..,C_n^{-1}$. For example:

$$[NOT, AND]^{-1} = [NOT^{-1}, AND^{-1}]$$

A diagram of this circuit appears in figure 2.5.

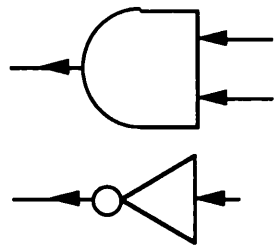


Figure 2.5: The inverse of [NOT, AND]

Inverse is used in the definition of the **conjugate** higher order function. The conjugate of two circuits R and S is denoted as R \ S and defined as:

$$R \backslash S \quad \equiv \quad S^{-1} ; R ; S$$

Figure 2.6 shows a picture of R\S. The following properties hold for conjugate:

$$(R \backslash S)^{-1} = R^{-1} \backslash S$$
$$(R \backslash S) \backslash T = R \backslash (S ; T)$$

The proofs are omitted— they may be found in [Sheeran 90] and are very simple.

Figure 2.6: R \ S

Conjugate is useful for expressing changes of representation.

We only have two values in our logic domain: high or low. Each wire in the circuit should only be driven by one output so that there is no possiblity of conflict. So, unlike other logic models, we do not have a value for high impedance. We also have to take care not to describe circuits which type-check but do not make physical sense. An example of such a circuit is:

$$AND ; AND^{-1}$$

We might mistakenly think this circuit makes sense at the physical level by assuming that applying a relation followed by its inverse should be like performing the identity relation, but reference to figure 2.7 shows that this circuit requires the outputs of two AND gates to be tied together: this will not always result in sensible electrical behaviour. However, the composition is well typed and does make sense at the abstract level. We have to impose extra structure over the meaning of composition to catch such badly formed circuits because they do not conform to physical reality.

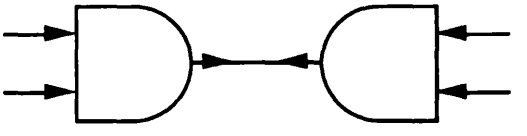


Figure 2.7: A badly formed circuit.

We have used arrows in previous pictures to hint at the desired information flow, although nothing in the semantics presented so far has enforced this. The two opposing arrows in the centre of the figure 2.7 tell us that something has gone wrong.

To describe sensible circuits, the output of one circuit must be the input of the next circuit. An analysis for checking this constraint is performed by Sheeran using

alternative interpretations. Such an interpretation is presented in more detail in chapter 4.

2.4 Wiring Relations

The circuits presented so far manipulate the information carried along wires in a non-trivial manner. These circuits manipulate data which is carried along either a single wire or a group of wires. As shown above, groups of wires are described by tupling. Often, the tupling structure has to be re-arranged to help fit circuits together.

This kind of re-arrangement is performed by an important class of circuits which are implemented as wiring relations. These circuits do not need to know exactly what the information being carried along the wires is. They simply re-arrange the tupling structure. Additionally, some wires may be lost while others may be introduced.

Consider the following specification:

$$\langle a, \langle b, c\rangle\rangle \; R \; \langle d, e\rangle \equiv (d = a) \,\&\, (e = b \wedge c)$$

The first element of the tuple in the domain is the same as the first element of the range tuple. How should these two values be related? To describe such relationships, we introduce the **identity** relation:

$$a \; \iota \; b \equiv a = b$$

In terms of hardware, this corresponds to a wire or wires which carry the 'input' signal to the 'output' signal unaltered.

The inverse of the identity relation is the identity relation:

$$\iota^{-1} = \iota$$

Flipping a horizontal wire or wires along a vertical axis does not change the wiring.

Using this relation, we can now realise R as:

$$[\iota, \text{AND}]$$

This can be proved to correctly realise R:

$$\begin{aligned}
&\langle a, \langle b, c\rangle\rangle \; [\iota, \text{AND}] \; \langle d, e\rangle \\
&\equiv \{ \text{ parallel composition } \} \\
&(a \; \iota \; d) \;\&\; (\langle b, c\rangle \; \text{AND} \; e)
\end{aligned}$$

≡ {definition of ι and AND }
(d = a) & (e = b ∧ c))
≡ { spec. of R }
⟨a, ⟨b, c⟩⟩ R ⟨d, e⟩

Figure 2.8 shows the layout for this circuit:

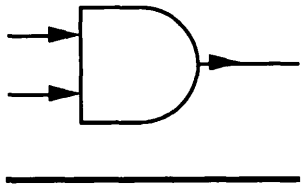


Figure 2.8: [ι, AND]

This is not the only realisation of R but this is the simplest and most natural implementation. A more complex realisation is [ι, [NOT, NOT] ; OR ; NOT].

Two common uses of the identity relation are abbreviated:

fst R = [R, ι]
snd R = [ι, R]

The example in figure 2.8 can now be re-expressed as **snd** AND.

Another common operation is to extract either the first or the second component of a pair. The relations π_1 and π_2 are defined for this purpose:

⟨x, y⟩ π_1 x ≡ true
⟨x, y⟩ π_2 y ≡ true

Consider the following specification S:

⟨a, ⟨b, c⟩⟩ S ⟨d, e⟩ ≡ (d = a ∧ b) & (e = c)

The bottom two wires a and b are fed into the domain of a 2-input AND gate whilst the top c wire passes through this circuit unchanged. The most obvious way to implement this circuit is by using the parallel composition of an AND gate and ι.

[AND, ι] :: ⟨⟨bool, bool⟩, β⟩ ↔ ⟨bool, β⟩

However, this circuit requires its domain to be of type ⟨⟨bool, bool⟩, β⟩ for some β, but the domain of S is of type ⟨bool, ⟨bool, χ⟩⟩. We wish to rearrange the elements of this tuple by altering the bracketing. The wiring circuit reorg performs the required manipulation in order to keep the types right.

$\langle a, \langle b, c\rangle\rangle$ reorg $\langle\langle a,b\rangle, c\rangle \quad \Leftrightarrow \quad$ true

reorg :: $\langle \alpha, \langle \beta, \chi\rangle\rangle \leftrightarrow \langle\langle \alpha,\beta\rangle, \chi\rangle$

This circuit is polymorphic in the sense that it will re-organise the input tuple for arbitrary substitutions for the types α, β and χ. By composing this circuit with [AND, ι] we obtain a suitable implementation for S:

$\langle a, \langle b,c\rangle\rangle$ reorg ; [AND, ι] $\langle d, e\rangle$

≡ { definition of reorg }

$\langle\langle a, b\rangle, c\rangle$ [AND, ι] $\langle d, e\rangle$

≡ { definition of AND and ι }

$(d = a \wedge b)$ & $(e = c)$

In the above example the type variables for reorg are α=β=bool and χ may be any type.

This reorganisation does not necessarily correspond to a physical wiring circuit. In the example above, reorg has three 'wires' going into its domain and the same three wires appear at its range in the same order. Here, reorg has been used to keep the types right, but other reorganisations will correspond to physical wiring circuits. We shall try to hint at the tupling structure in our diagrams by the spacing between the wires. Figure 2.9 shows the circuit we have proposed for S:

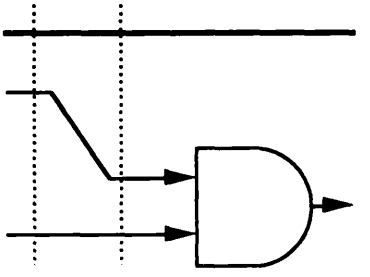


Figure 2.9: Reorganisation of tuples.

The top wire may actually be a tuple of wires. To distinguish between wires carrying single values and wires carrying unknown or composite values we shall use a heavier line for the latter. Notice also that the top signal does not have any arrows on it: this is because we cannot determine from the given context the direction of information flow over this wire. The reorganisation of the wires is shown between the dotted vertical lines.

In the preceding example, a custom built relation was used to solve a plumbing problem. There are certain wiring patterns that occur frequently. For example, extracting the first element of a tuple is a useful operation. Instead of defining one relation to extract the first element from 2-tuples, and another from three tuples etc. we can define generic tuple relations.

Some wiring circuits operate over lists of data rather than a fixed size tuple. However, making a clear-cut distinction often leads to a great deal of conversion between tuples and lists. We shall assume that a homogeneous tuple is as good as a list, so ⟨a, b, c⟩ could be a triple or a three element list, depending on the context.

In a picture there is no difference between an element of a given type and a singleton list of that type. However, to keep the types of compositions right, we have to distinguish between a signal and a list containing only one signal. Ruby provides an abstraction [-] for relating a signal to a list containing only that signal.

$$x \quad [-] \quad \langle x \rangle \quad \Leftrightarrow \quad \text{true}$$

A common operation on lists is to combine two lists pairwise. The name given to this operation is zip (an instance is shown in figure 2.10a) and it is described by:

$$\langle x, y \rangle \text{ zip } z \quad \Leftrightarrow \quad \forall i.\ z_i = \langle x_i, y_i \rangle$$

where z_i is the i^{th} element of z. This converts a pair of lists to a list to 2-tuples. Unzipping from a list of 2-tuples to a 2-tuple of lists may be done by using zip^{-1}.

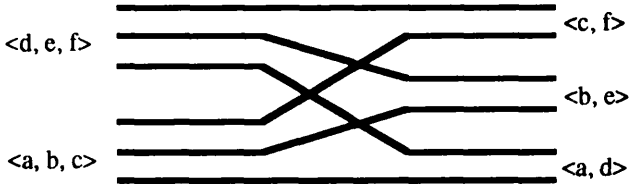


Figure 2.10a: ⟨⟨a, b, c⟩, ⟨d, e, f⟩⟩ zip ⟨⟨a, d⟩, ⟨b, e⟩, ⟨c, f⟩⟩

Another useful operation is transposition (trn). This interleaves a list of lists and is rather like matrix transposition. The definition is:

$$x \text{ trn } y \quad \Leftrightarrow \quad \forall i, j.\ x_{i,j} = y_{j,i}$$

Two lists may be combined to form a larger list by appending. The circuit app concatenates two lists: it is described by:

$$[R_0, R_1, \ldots, R_i, R_{i+1}, R_{i+2}, \ldots, R_n] = [[R_0, R_1, \ldots, R_i], [R_{i+1}, R_{i+2}, \ldots, R_n]] \setminus \text{app}$$

Figure 2.10b shows a three element list being appended to another three element list yielding a size six list.

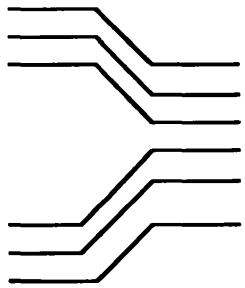


Figure 2.10b: Appending lists using app.

Lists may be built up one component at a time by using wiring relations that introduce a new signal either on the left (apl) or the right (apr):

```
apl = fst [-] ; app
apr = snd [-] ; app
```

Figure 2.11 shows an instance of apl and apr.

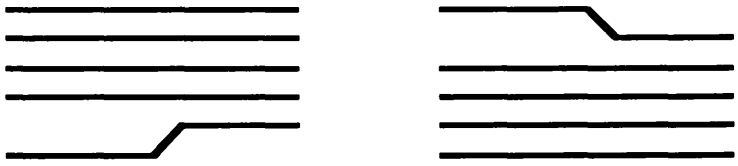


Figure 2.11: (a) append left (b) append right

A useful operation on lists is rev which reverses the elements of a list. It has the following defining properties:

```
[-] ; rev = [-]
app ; rev = [rev, rev] ; rev ; app
```

Reversing a list twice leaves it unaltered so rev is its own inverse. By restricting rev to work on lists or tuples of length two, we obtain the circuit that swaps its inputs. The restriction is denoted by \2 which has the effect of constraining the domain to be a 2-tuple.

```
swap = rev \ 2
```

The swap circuit shall be drawn as two wires crossing over, as in figure 2.12. Wires cross over without interfering with each other. Contacts between wires shall be shown explicitly.

Figure 2.12: Swapping the elements of a 2-tuple.

Swap is its own inverse and applying swap twice is like applying the identity.

$$\text{swap} ; \text{swap} \quad = \quad \iota \backslash 2 \quad = \quad \text{swap} ; \text{swap}^{-1}$$

Thus, swap = swap^{-1}.

A bus can be duplicated by using split.

$$x \quad \text{split} \quad \langle x, x \rangle \quad \Leftrightarrow \quad \text{true}$$

Multi-way forks can be made by repeated use of split.

split4 = split ; [split, split] ; app

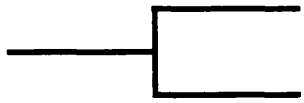


Figure 2.16: split

2.5 Replication of Circuits

Often many copies of the same circuit are combined to form a larger circuit. This section presents various ways to replicate circuits in Ruby.

For circuits which have the same domain and range types, it is possible to lay out horizontally many copies of the same circuit. This is represented by superscripting e.g. R^4 = R; R; R; R.

A common way to replicate a circuit is to apply it to each signal in a list of signals. This is analogous to applying a function to each element of a list. The higher order function that performs this task in functional programming languages is called **map** and this is also the name used for mapping a relation over lists of signals.

Map has the following properties:

n ; **map** R	=	**map** R ; *n*
[-] ; **map** R	=	R ; [-]
app ; **map** R	=	[**map** R, **map** R] ; app

Since **map** R represents an infinite class of circuits it is not possible to draw a finite picture of it. Figure 2.17 shows one representative of this class.

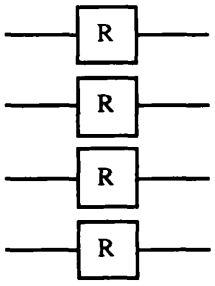


Figure 2.17: map R over a four element list.

Map has properties which are similar to those for parallel composition. For example:

map (R ; S)	=	**map** R ; **map** S
(**map** R)$^{-1}$	=	**map** (R^{-1})
(**map** R) \ (**map** S)	=	**map** (R \ S)

2.6 Sequential Circuits

The relations presented so far describe combinational circuits; the output of the circuit at any time depends solely on the inputs at that time. Most circuits have memory elements so the output depends not only on the current input, but also the past inputs whose history is encoded in the internal memory components.

First we need to augment the definition of a signal. For combinational circuits, a signal was just one value. For sequential circuits, a signal is a stream of values.

If s is a signal then s(t) is defined to be the value of the signal at time t. For example:

$$\langle a, b, \langle c, d\rangle\rangle\,(t) \quad = \quad \langle a(t), b(t), \langle c(t), d(t)\rangle\rangle$$

Notice that on the left we have a tuple of signals (where the basic element is a stream) and on the right we have a tuple of basic elements.

To describe sequential circuits requires information about the past. The output of a sequential circuit may depend on the current input at time index *i* and the previous value of the state element at time i-1. To make the past history of a signal available we delay the 'arrival' of the stream. This is accomplished by the use of a delay element $\mathcal{D}$ defined as:

$$a\,\mathcal{D}\,b \quad \Leftrightarrow \quad \forall t.\ a(t-1) = b(t)$$

Here, a and b must be streams. As an example, consider delaying a tuple of signals. This corresponds to delaying each individual signal.

$$\langle a, b, c\rangle \mathcal{D} \langle d, e, f\rangle \Leftrightarrow \forall t.\ \langle a, b, c\rangle (t-1) = \langle d, e, f\rangle (t)$$
$$\Leftrightarrow \forall t.\ \langle a(t-1), b(t-1), c(t-1)\rangle = \langle d(t), e(t), f(t)\rangle$$
$$\Leftrightarrow (a\ \mathcal{D} d)\ \&\ (b\ \mathcal{D} e)\ \&\ (c\ \mathcal{D}\ f)$$

So $\mathcal{D}$ does not necessarily work over just one 'wire': it may relate composite signals. Thus it is not sufficient to think of $\mathcal{D}$ as being implementable as just one bit level memory element. The symbol for the delay element is shown in figure 2.41.

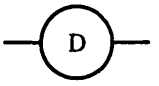


Figure 2.41: The delay element.

As usual, the inverse of this circuit is formed by flipping the domain and the range. If an anti-delay $\mathcal{D}^{-1}$ is 'driven' from left to right, then it predicts values rather than remembering them. The use of both $\mathcal{D}$ and $\mathcal{D}^{-1}$ facilitates reasoning about circuit timing and retiming [Sheeran 88]. In the final design, the anti-latches must be driven from right to left.

The combinational components defined so far can still be used in sequential circuits by 'lifting' their definitions to work on streams. Consider the example of the $\mathsf{AND}_{\mathsf{seq}}$ relation which is lifted so that it operates pointwise over elements of the signals in the domain and range.

$$\langle a, b\rangle\ \mathsf{AND}_{\mathsf{seq}}\ c \Leftrightarrow \forall t.\ \langle a, b\rangle(t)\ \mathsf{AND}\ c(t)$$
$$\Leftrightarrow \forall t.\ (a(t), b(t))\ \mathsf{AND}\ c(t)$$

Many sequential circuits require past values to be fed back into the circuit so that they may be used to determine the current output. To describe this kind of feedback we introduce a new circuit former loop:

$$a\ (\mathbf{loop}\ H)\ b \quad =_{def} \quad \exists c.\ (a, c)\ H\ (b, c)$$

$$\frac{H :: (\alpha, \beta) \sim (\chi, \beta)}{(\mathsf{loop}\ H) :: \alpha \sim \chi}$$

The **loop** relation takes as parameter a circuit which relates a 2-tuple to a 2-tuple. The second element of the range tuple is fed back and used as the second element of the domain. A schematic for the loop circuit former is shown in figure 2.19.

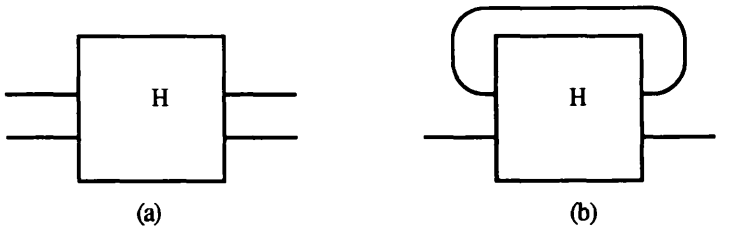


Figure 2.19: Feedback loop (a) H (b) loop H.

2.7 Four-Sided Tiles

So far circuits have been laid out like tiles with connections on only two sides, thus allowing only very horizontal layouts. Two dimensional circuits may be described by placing connections on all four sides of a rectangular tile. This is done in Ruby in a way that does not require the semantics already presented to be changed in any way.

Two sides of a tile are considered to be the domain of the circuit (the left and the top) and the other two sides form the range (the right and the bottom). Figure 2.21 shows a picture of a four sided tile and its inverse. Because of our convention about the position of the domain and range, the inverse is formed by flipping along a diagonal line running from the bottom left hand corner to the top right hand corner.

The domain of a four sided tile is always a 2-tuple. The first element describes information on the left of the tile and the second element refers to the top of the tile. Similarly, the range is also always a 2-tuple with the first element referring to the bottom of the tile and the second element referring to the right hand side of the tile.

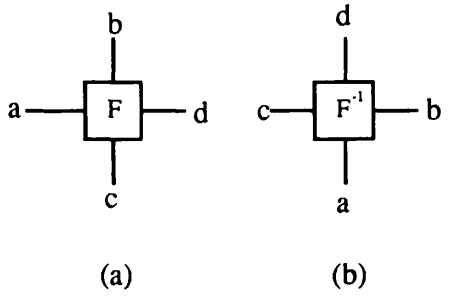


Figure 2.21: (a) Four sided tile (b) and its inverse.

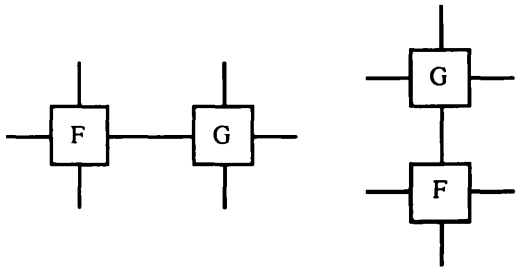
Both two sided and four sided tiles will be used in descriptions. It will usually be clear from the context which type of tile is being used.

New combining forms are required to compose four sided tiles. These tiles may be composed horizontally by using beside ($\leftrightarrow$) or vertically by using below ($\updownarrow$). The definitions of these combining forms are:

$$\langle a, \langle b, c\rangle\rangle \; F \leftrightarrow G \; \langle\langle d, e\rangle, f\rangle$$
$$=_{def} \quad \exists g.\ \langle a, b\rangle F \langle d, g\rangle \;\&\; \langle g, c\rangle G \langle e, f\rangle$$

$$\langle\langle a, b\rangle, c\rangle \; F \updownarrow G \; \langle d, \langle e, f\rangle\rangle$$
$$=_{def} \quad \exists g.\ \langle a, g\rangle F \langle d, f\rangle \;\&\; \langle b, c\rangle G \langle g, e\rangle$$

Figure 2.22 demonstrates these compositions pictorially.



(a) (b)

Figure 2.22: (a) beside (b) below

Two generic combining forms for composing many copies of the same tiles either horizontally or vertically are now presented. Let **row** take as a higher order parameter a four sided tile *F*: it will form a new circuit which contains many copies of F. Similarly, let col be the higher order combining form for making vertical arrays. Signal construction is denoted by ':' so for example a:⟨b, c⟩ = ⟨a, b, c⟩.

$$\langle a, \langle\rangle\rangle \quad \textbf{row}\ P \quad \langle\langle\rangle, a\rangle$$
$$\langle a, b{:}c\rangle \quad \textbf{row}\ P \ \langle d{:}e, f\rangle \quad =_{def} \quad \langle a, \langle b, c\rangle\rangle \quad (P \leftrightarrow \textbf{row}\ P) \quad \langle\langle d, e\rangle, f\rangle$$
$$\textbf{col}\ P \quad =_{def} \quad (\textbf{row}\ P^{-1})^{-1}$$

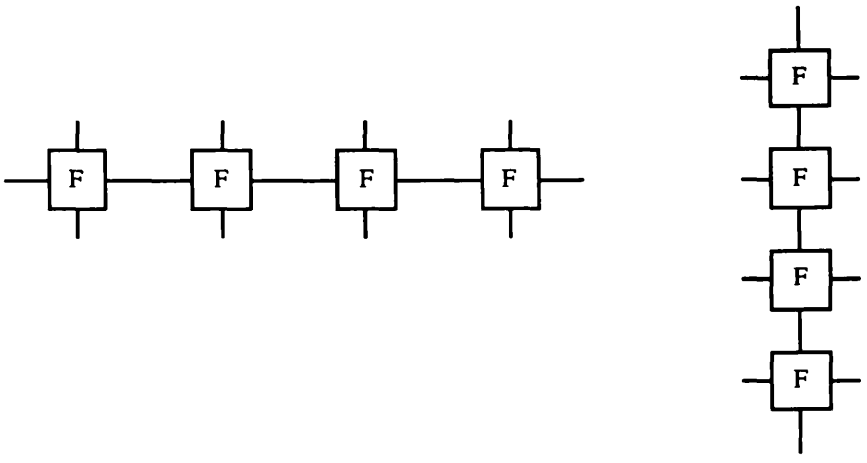
The following properties hold for these combining forms:

$$\textbf{snd}\ [\text{-}]\ ;\ \textbf{row}\ R \quad = \quad R\ ;\ \textbf{fst}\ [\text{-}]$$
$$\textbf{row}\ R\ ;\ \textbf{fst}\ [\text{-}]^{-1} \quad = \quad \textbf{snd}\ [\text{-}]^{-1}\ ;\ R$$
$$\textbf{snd}\ ([m, n]\ ;\ app)\ ;\ \textbf{row}\ R \quad = \quad ((\textbf{row}\ R\ ;\ \text{fst}\ m) \leftrightarrow (\textbf{row}\ R\ ;\ \textbf{fst}\ n))\ ;\ \textbf{fst}\ app$$
$$\textbf{row}\ R\ ;\ \textbf{fst}\ (app^{-1}\ ;\ [m, n]) \quad = \quad \textbf{snd}\ app^{-1}\ ;\ ((\textbf{snd}\ m\ ;\ \textbf{row}\ R) \leftrightarrow (\textbf{snd}\ n\ ;\ \textbf{row}\ R))$$

$$\textbf{fst}\ [\text{-}]\ ;\ \textbf{col}\ R \quad = \quad R\ ;\ \textbf{snd}\ [\text{-}]$$
$$\textbf{col}\ R\ ;\ \textbf{snd}\ [\text{-}]^{-1} \quad = \quad \textbf{fst}\ [\text{-}]^{-1}\ ;\ R$$

fst ([m, n] ; app) ; **col** R = ((**col** R ; **snd** m) ↕ (**col** R ; **snd** n)) ; **snd** app

Instances of a **row** and a **col** are shown in figure 2.23.



(a) (b)

Figure 2.23: (a) **row** F (b) **col** F

The following laws hold about row and col:

col R = (**row** R^{-1})$^{-1}$
(**row** F) ↕ (**row** G) = **row** (F ↕ G)
col F ↔ **col** G = **col** (F ↔ G)

A useful variant of **row** is **rdl** (reduce left, figure 2.24) which is defined as:

rdl R = **row** (R ; π_2^{-1}) ; π_2

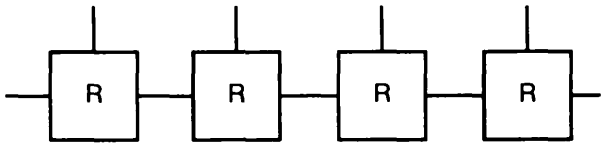


Figure 2.24: An instance of reduce left.

Reduce right (**rdr**) can be defined from **col**.

rdr R = **col** (R ; π_1^{-1}) ; π_1

Two to one relations can be cascaded using **rdl** or **rdr**. An example of such a cascade is **rdl** AND.

2.8 Distributing Signals

Distributing a signal across a tuple of signals is performed by adding the signal to be distributed either to the left or to the right of each element of the tuple to be distributed over. This leads to four possible patterns, two of which are abbreviated in Ruby as follows:

$\langle a, \langle\rangle\rangle\ dist_L\ \langle\rangle$

$\langle a, b\rangle\ dist_L\ c \quad \Leftrightarrow \quad \forall i.\ c_i = \langle a, b_i\rangle$

$\langle\langle\rangle, b\rangle\ dist_R\ \langle\rangle$

$\langle a, b\rangle\ dist_R\ c \quad \Leftrightarrow \quad \forall i.\ c_i = \langle a_i, b\rangle$

Two examples are:

$\langle a, \langle b, c, d, e\rangle\rangle\ dist_L\ \langle\langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle, \langle a, e\rangle\rangle$

$\langle\langle b, c, d, e\rangle, a\rangle\ dist_R\ \langle\langle b, a\rangle, \langle c, a\rangle, \langle d, a\rangle, \langle e, a\rangle\rangle$

A circuit for distribute left can be made by using four sided tiles. The value to be distributed is fed from right to left while the signals to the individual components to be distributed flow from top to bottom. One suitable implementation for $dist_L$ is then given by:

$lsh = \iota \updownarrow \iota$

$rsh = \iota \leftrightarrow \iota$

$dist_L$ = **row** (**fst** split; lsh ; swap) ; π_1

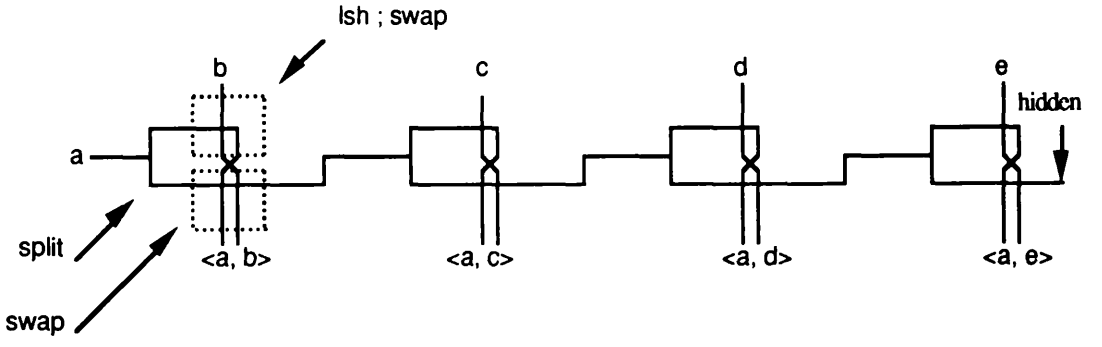


Figure 2.25: An instance of the distribute left circuit for the example.

Distribute right can be defined in terms of distribute left:

$dist_R$ = swap ; $dist_L$; **map** swap

2.9 Some Examples

In this section, some of the Ruby constructs defined above are used to build a 2 to 1 multiplexer and a binary adder. The specification for the multiplexer we want to build is:

$$\langle a, \langle b, c\rangle\rangle \; MUX \; d \quad \Leftrightarrow \quad d = a \wedge b \vee \neg a \wedge c$$

If a is true then d is connected to b; if a is false then d is connected to c. This is analogous to an if..then..else expression. This multiplexer may be implemented as:

$$MUX \quad = \quad dist_L \; ; [AND, \; [NOT, \iota] \; ; \; AND] \; ; OR$$

This description is shown to be faithful to the specification:

$\langle a, \langle b, c\rangle\rangle \; dist_L \; ; [AND, [NOT, \iota] \; ; AND] \; ; OR \; d$
$\equiv$ { definition of $dist_L$ }
$\langle\langle a, b\rangle, \langle a, c\rangle\rangle \; [AND, [NOT, \iota] \; ; AND] \; ; OR \; d$
$\equiv$ { definition of parallel composition and AND and ι }
$\langle a \wedge b, \neg a \wedge c\rangle \; OR \; d$
$\equiv$ { definition of OR }
$d = a \wedge b \vee \neg a \wedge c$

Figure 2.26(a) shows the symbol used for a MUX multiplexer and part (b) shows the implementation.

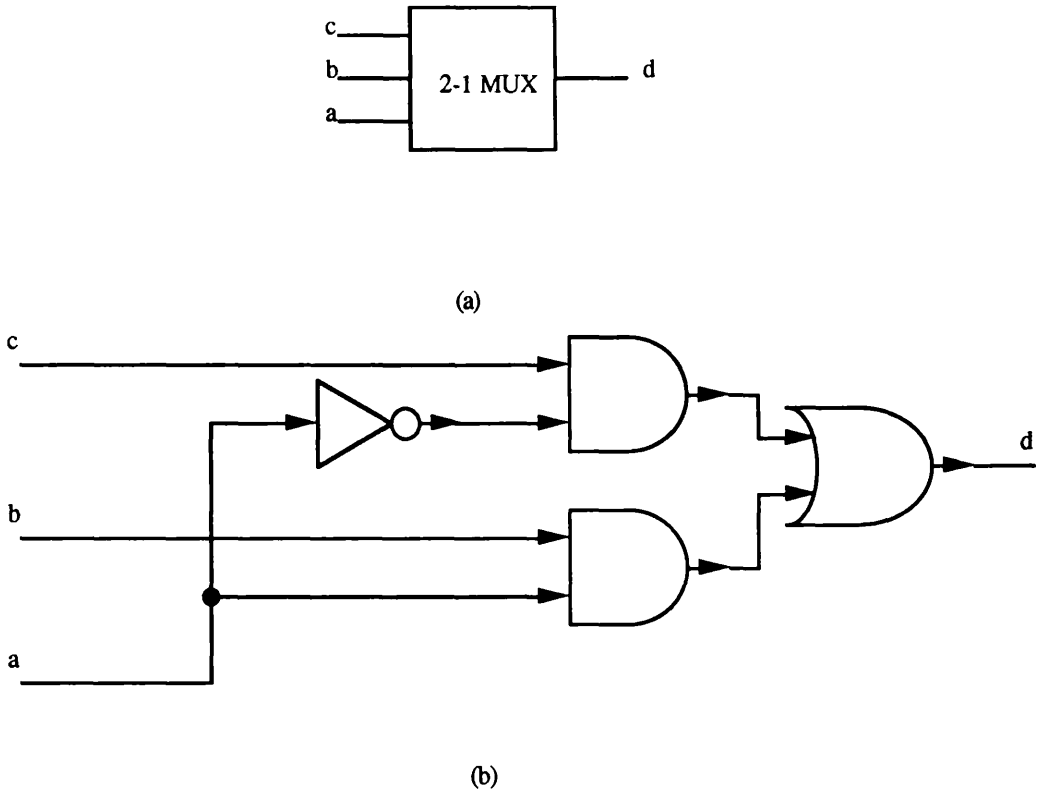


Figure 2.26: (a) A 2-to1 multiplexer symbol (b) and an implementation.

The second example circuit is a binary addition circuit BINADD. This circuit is represented by a four sided tile so the domain and range are pairs. The first element of the domain (a vertical signal) is the carry in and the second element of the domain is a pair of lists of equal length. The lists represent binary values which are to be added pairwise. The first element of the range is a list representing the sum of the two lists on the domain and the second element is the carry out.

A full adder circuit FA is used to add two binary values with a carry in to produce a sum and a carry out. Let this be a four sided tile, with the carry in as the first element of the domain and the pair of binary values to be added as the second element. The first element of the range is the sum and the second element is the carry out.

A binary adder can now be implemented as:

BINADD = **snd** zip ; **row** FA

An instance of circuit of BINADD is shown in figure 2.27.

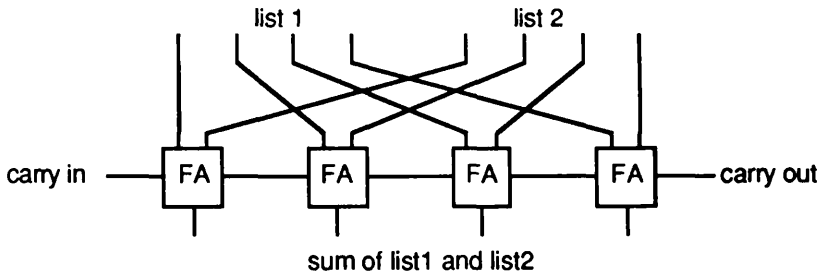


Figure 2.27: BINADD

A full adder can be built from two half adders. A half adder HA takes a pair on its domain representing two binary values. The range of a half adder is also a pair whose first element is the carry resulting from the binary addition of the two values in the domain. The sum itself is given in the second element of the range. Using this component, the definition of FA is:

```
FA  =  snd HA ; rsh ; fst swap ; lsh ; snd HA ; rsh ; fst OR ; swap
```

Figure 2.28 shows how the full adder is constructed.

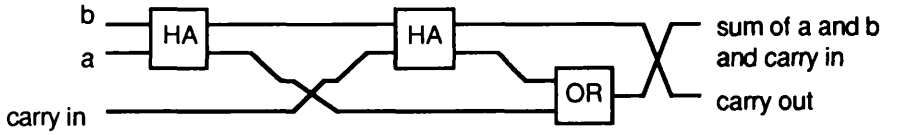


Figure 2.28: Construction of a full adder.

A half adder is made from an AND gate and an exclusive-or circuit:

```
HA    =  split ; [AND, EXOR]
exor  =  split ; [[ι, NOT], [NOT, ι]] ; [AND, AND] ; OR
```

2.10 Summary

A large subset of the Ruby hardware description language has been introduced. A circuit is viewed as a transformer of synchronous streams or signals. Circuits are composed using powerful yet simple combing forms. These combining forms convey structural and behavioural information. Ruby abstracts away from the notion of input and output by considering a circuit to be a binary relation between signals. This gives rise to new combining forms that exhibit symmetries which would not have been available in a purely functional model.

This chapter has presented the normal or *standard* semantics attributed to Ruby. We show in chapter 5 that by altering the semantics we can obtain additional interesting information about Ruby circuits.

Chapter 3

Testing Digital Circuits

3.1 Introduction

Most manufactured components have to be tested, but the testing of integrated circuits is a particularly difficult task. Traditional testing of assembled devices (e.g. chairs) relies on visual inspection and the application of common sense. The testing of electronic components like televisions is less straightforward, and involves the use of electronic measurement tools like oscilloscopes to measure electrical characteristics of internal connections. The internal workings of an integrated circuit are not usually accessible— the only way to test such a device is by exercising it through its primary inputs and observing the results at the primary outputs.

This problem is exacerbated by the fact that large digital systems are broken down into smaller subsystems which may not have an easily identifiable function. Common sense is no longer a viable technique for testing such complex circuits. Also, checking the *form* of the outputs is not enough: it will invariably be 1s and 0s. It is necessary to check the *pattern* of outputs. For most circuits it is not possible to apply all the input test patterns. A subset of the test patterns which results in a high degree of confidence in the circuit must be found. This is a very difficult task that requires large amounts of computer and human resources. The generation of test patterns for general sequential circuits is not fully automated— often an experienced test engineer has to find tests manually.

There are many reasons why circuits should be specified more formally and the challenge testing is one of the most compelling. Too often in the past the test engineer has had little information about the function of the circuits to be tested. A formal rigourous notation is required for describing circuits so that they can be easily understood. Automatic tools are used extensively in testing, and these tools require precise descriptions of the circuits they analyse. These are yet more reasons why hardware description languages with precisely defined semantics, like Ruby, are becoming

increasingly important for circuit description.

This chapter presents a brief introduction to the field of testability. Section 3.2 gives reasons why testing has become one of the most important stages of integrated circuit development. Section 3.3 classifies various types of test and how defects in circuits are described. Section 3.4 presents a formal description of notions like 'test pattern'. Some popular methods of generating tests for combinational circuits are presented. These include a path sensitization technique for manual test pattern generation (which is formalised and automated by the D-algorithm presented in a later chapter) and the method of boolean differences. Techniques like fault collapsing are introduced for reducing the large amounts of information that are handled by CAD (Computer Aided Design) tools performing test pattern generation. Section 3.5 shows how the very expensive task of generating tests for a circuit can be reduced by using each pattern to cover as many faults as possible. A technique called deductive fault simulation is presented which, given as input a circuit, a fault and a test T covering that fault, will produce a list of all the other faults which are exposed by the given test pattern T. Section 3.6 presents a method for estimating how testable a given design is. This could be used in the design stage to improve subcomponents that are difficult to test by making them more accessible. Section 3.7 presents various methods for improving the testability of circuits and shows how sequential circuits can be tested.

3.2 Why Circuits have to be Tested

Certain applications such as life critical systems require a high degree of reliability. Developers of such systems need a guarantee that the components they use will operate faithfully to their specification. This guarantee is usually provided by testing components before they are delivered to the customer. The procedure of testing occurs in two distinct phases of the design and production of integrated circuits. The techniques employed for testing at these two phases are different.

The design phase phase involves making a series of refinements from a specification of a circuit to a physical realisation. Circuit specifications can be very complex and physical realisations might require over a million components. Specifications encompass not only the intended logical behaviour, but also performance constraints like power and speed and resource constraints like area. There is a large scope for error or inconsistency between the specification and the derived design. Verification involves checking that implementations are consistent with specifications. Testing is one verification technique

for detecting such inconsistencies. A model of a design is simulated by a computer until a satisfactory degree of confidence in the behaviour of the design is achieved. A design which fails a test has to be redone, using any diagnostic information provided by the test procedure. The fabrication of integrated circuits is a very expensive task, so every effort must be made to ensure a design is correct before attempting to physically construct it.

We say very little about this kind of testing. Verification of designs is not easy to perform by testing because of the vast number of test patterns that have to be applied before a circuit can be *proved* to be correct. Sometimes, for practical reasons, not all faults can be tested for. A subset of likely faults are identified and a set of tests to expose these faults are generated. This method does not prove the correctness of a design. Much work has been done on the use of formal mathematical techniques to reason about designs in order to prove useful properties and ultimately correctness [Cohn & Gordon 86, Melham 87, Cohn 87].

Integrated circuits are manufactured on disks of silicon (called **wafers**) containing typically many copies of the same circuit. Each copy is called a **die**. A wafer is typically 75mm in diameter and contains one hundred 5mm square dies. Even if the design of the circuit has been proved to be correct, it is still possible that a physical realisation of the correct design does not meet its specification. The manufacturing process for integrated circuits is far from perfect— many of the dies may have been badly formed. For certain types of circuits such as large microprocessors like the Motorola 68000 as many as 70% of the dies may be damaged. Wafers are baked in furnaces which may be at the wrong temperature as well as being treated by various chemicals which may be of the wrong composition. A single speck of dust can render a die useless. These variations and imperfections decrease the ratio of working dies to the total number of dies on a wafer. This ratio is called the **yield**.

The quality control stage of production must isolate defective components so that they can be removed. The process of determining which dies on a wafer are working is called **wafer sort**. Preventing the shipment of broken circuits is becoming increasingly important as greater emphasis is placed on quality. Another reason for discarding defective dies is the high cost of **bonding** which is often as much as a third of the total production cost. Bonding is the setting the dies in ceramic packages and linking the tiny pads of each die with the pins of the chip.

Manufacturing errors modify the behaviour of a circuit in many ways. A circuit can still perform its intended logic function, but at the wrong speed, or perhaps it may consume too much power. **Parametric testing** involves measuring these analogue quantities to ensure that performance constraints are satisfied. Analogue quantities can

deviate from the expected values because of the variations in manufacturing process or because of bad design, e.g. a channel being too narrow to cope with the required current flow. **Functional testing** or **logic testing** is the checking of the logical behaviour of the circuit. Although both types of testing are essential, nothing is said here about parametric testing: the techniques presented in this thesis pertain mainly to functional testing.

The primary reason for performing testing at the post-production stage is to discard defective components. Since integrated circuits are encased in ceramic packages and dies contain features of the submicron scale, repair is not usually a viable option. The test procedure can provide useful diagnostic information which can be used to help locate a fault in some subcircuit. This information can be used to improve the fabrication process and the design process. For example, the temperature of a furnace can be reduced or a component that fails frequently redesigned using more reliable design rules.

3.3 Types of Test

The activity of producing a suitable collection of test patterns to exercise a circuit is called **test pattern generation** (often abbreviated as TPG). Ideally, test pattern generation should be performed automatically by CAD tools, but this has only been realised for a restricted class of circuits. Much test pattern generation is still done manually. This consumes valuable time of experienced test engineers and is very costly.

Tools that perform **automatic test pattern generation** (ATPG) are based on formalisations of manual techniques.

The obvious way to test a circuit is to see if it is operating correctly with respect to its specification. This is the approach taken by **functional test programs**. The word 'program' does not mean a piece of software, but a sequence of test patterns. The specification usually used is a truth table. The input part of each row of the truth table is applied in turn for combinational circuits. The output for each pattern is checked against the expected result in the truth table. Any deviation from the expected values indicates the circuit is faulty and should be discarded.

This approach is not very practical for various reasons. It is often very difficult to derive a truth table for a circuit. Even if a truth table is available, the number of test patterns required is related exponentionally to the number of primary inputs. If a

combinational circuit has n inputs (Figure 3.1(a)), then it will require 2^n test patterns to be applied in order to be tested exhaustively. For even fairly modest values of n, the number of test patterns required becomes prohibitively large.

The problem is amplified for sequential circuits. These circuits have to be tested with all possible input combinations for each possible combination of internal state variables. For a circuit with n primary inputs and m state elements (Figure 3.1(b)), this requires 2^{n+m} patterns.

If, say, $n = 24$ and $m = 20$, the resultant number of test vectors for exhaustive testing is 2^{44}. If we could generate test vectors at a rate of 106 vectors/sec, then testing will take six months at 24 hours per day!

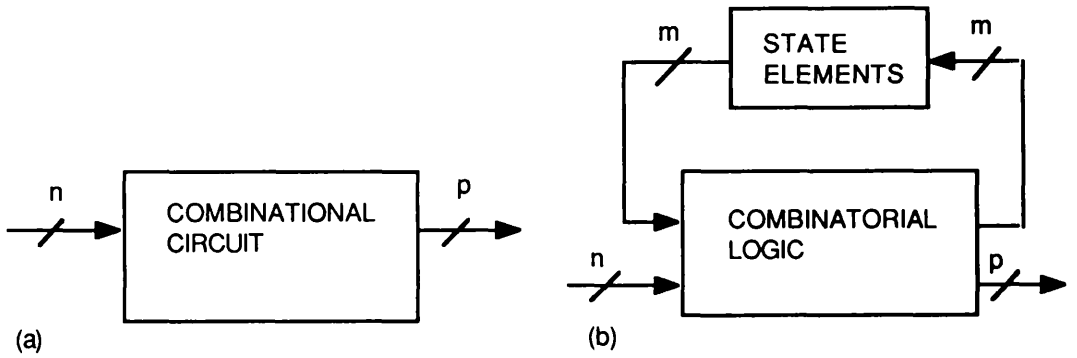


Figure 3.1: (a) Combinational circuit (b) Sequential circuit.

By applying a 'divide and conquer' technique, we can decompose a big circuit into smaller circuits and test these independently. If we can test the combinatorial and sequential elements separately, then the number of vectors required for exhaustive testing is reduced to 2^n+2^m. This could be done in under 20 seconds— a dramatic reduction in testing time.

Thus, by partitioning the circuit into smaller subunits and testing state elements independently we can make the problem of testing manageable. This involves thinking at the design stage about testability. The circuit has to be designed to allow access to the subunits and will require extra pins, increasing the packaging cost.

Design techniques to cope with testing address the following areas of importance:

- Test generation
- Test verification
- Design for test

Test generation entails finding and producing the smallest set of test vectors that will give the greatest coverage of faults. Test verification concerns assessing the fault coverage of a set of test vectors. Designing with testing in mind reduces the complexity of the two previous problems.

Not all circuits can be naturally described by a truth table. For example, a microprocessor is understood in terms of its instruction set or a set of register transfer rules. An exhaustive test would involve executing every instruction with every operand. This is clearly not acceptable. Different techniques are required for testing such circuits.

Instead of checking to see if a circuit is working, a test program can be constructed to check if a circuit is faulty. By considering the physical structure of the circuit, a set of possible **defects** is enumerated and tests constructed for each defect. A defect is a physical failure that causes functional failure. Clearly this requires more information than just the behaviour of the circuit: the physical layout is now important too. Test generation techniques that attempt to detect specific structural failures generate **structural test programs**. Another name for structural testing is **fault-oriented TPG**.

Several physical defects can have the same electrical effect on the circuit. These faults have the same effect on the observable outputs, making them indistinguishable. For this reason it is more profitable to think in terms of **faults** which are the electrical effects of physical defects.

A general type of physical failure can now be represented in terms of how it affects the logical operation of the circuit. The relationship between the physical defects and faults is expressed by a **fault-model**. Often, a fault arises from a variety of physical defects. The consequence of this is that fault-models can relate a fault to a list of physical defects.

Such a fault model is the **single-stuck-at-fault model** [Weste & Eshraghian 85]. This model makes two assumptions about how circuits can fail. The 'single' in the name refers to the assumption that only a single node in the circuit is directly affected by a fault. The second assumption is that the electrical effect of the fault is to cause a node to be 'stuck' at logic 0 or logic 1, irrespective of the stimuli applied at the primary inputs.

These assumptions are simplifications of the way in which circuits fail. A failure can be caused in some other way. One example is two lines being connected together so that they are always at the same logic level. Also, a circuit may fail at several nodes, not just one. However, for most circuits, the single-stuck-fault model gives surprisingly good results. This is the most widely used model in industry.

This model does not cover all faults. For example, the joining of two wires that were

not previously joined can radically alter the behaviour of a circuit. For CMOS, some faults may convert a combinatorial circuit into a sequential circuit. This happens when a node becomes permanently detached from source or drain due to a defective transistor. The value on this node will depend on its previous value i.e. the charge stored there due to capacitance.

Consider a 2-input AND gate f = a AND b. There are three wires associated with this gate: two input wires and one output wire. Each wire can have one of two faults (i.e. stuck at zero or stuck at one). So there are six possible faults. The notation a/0 is used to mean node a stuck-at logic 0— similarly for a/1. A truth table for the fault-free circuit and possible faults is:

a	b	c	a/0	b/0	c/0	a/1	b/1	c/1
0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	1	0	1
1	0	0	0	0	0	0	1	1
1	0	1	0	0	0	1	1	1

a/0 is the output expected if there was a stuck at zero fault on input wire a. Using this model, a/0, b/0 and c/0 are indistinguishable— they are said to be equivalent. Thus there are only four different fault classes. We only need enough test patterns to cover all the fault classes. The following tests form a cover:

Test ab=11 detects	a/0 b/0 c/0 (Expected output 1, get 0)
Test ab=10 detects	b/1 c/1 (Expected 0, get 1)
Test ab=01 detects	a/1 (Expected 0, get 1)

Notice how this model assumes that there is never more than one fault at a time; it does not model wire a and wire b stuck at zero simultaneously.

3.4 Test Pattern Generation

If C specifies the behaviour of a working circuit, then let C_f describe the behaviour of C under the influence of fault f. Here, C is a function from input patterns to output patterns. The fault f is exposed by finding some input T for which C produces a different result from C_f. An input exposes a fault when either of the following conditions are true:

$$C(T) \neq C_f(T) \quad (3.1)$$
$$C(T) \oplus C_f(T) \quad (3.2)$$

Exclusive-or is denoted by the symbol $\oplus$. For a given fault *f*, there may be no test patterns or multiple test patterns. Relationship 3.1 can be used to construct the sets of all test TESTS(C_f) for a circuit C under fault *f*.

$$\text{TESTS}(C_f) = \{T : T \in \text{INPUTS}(C) ; C(T) \neq C_f(T)\} \quad (3.3)$$

Note that TESTS is a two place operation taking a circuit C and a fault f. The set of all input patterns for a circuit C is given by INPUTS(C). For example:

$$\text{INPUTS(AND)} = \{00, 01, 10, 11\}$$

For an *n*- input circuit, there are 2^n patterns produced by INPUTS.

Not all faults are testable. A fault can occur in a redundant part of a circuit where certain failures will have no effect on the correct behaviour of the circuit. Redundancy is often introduced to avoid other problems like hazards so it cannot always be removed. If a fault *f* is untestable, then TESTS(C_f) will be an empty set i.e. TESTS(C_f) = { }.

Often, there is more than one member of TESTS(C_f) but only one member of this set is required to test for fault f. Any member can be chosen, but some choices are better than others. This is because some test patterns cover multiple faults, so a judicious selection can reduce the total number of test patterns required for a circuit.

A **fault-list** is the set of all possible faults in a circuit. The particular faults present in this set will depend on the fault model employed. For the AND gate in Figure 3.3a there are six possible faults (two for each node) if the single-stuck-at-fault model is used. The function FAULTLIST(C, p_1, ...,p_n) is defined to return the fault-list for a given circuit C with input p_1, ...,p_n assuming the stuck-at-fault-model. For example:

$$\text{FAULTLIST(AND, a, b)} = \{a/0, a/1, b/0, b/1, c/0, c/1\}$$

The fault-cover is the percentage of the faults in the fault-list that are covered by a test program. The ideal of 100% fault cover is not always realisable because some faults may be untestable or the circuit may be too large to make this practical. A test program to cover all testable faults can now be specified as follows:

$$\text{TESTPROGRAM(C, ...)} = \{(T, C_f(T)) : f \in \text{FAULTLIST(C, ...)} ; T \in \text{TESTS}(C_f)\} \quad (3.4)$$

The set expression selects one fault at a time from the fault list and then chooses one member (if it exists) from the set of tests that exposes that fault. Each input test pattern is paired with the result of the working circuit for input T.

This specification gives the largest test program that does not contain duplicate test patterns. Instead of choosing just any member T of TESTS(C_f) the selection could be made to prefer a T which exposes many other faults too.

A direct transcription of the above specification to code would not yield an efficient automatic test pattern generation program. Generating tests using the above specification uses no information about the construction of the circuit so is an example of a functional test program.

For some circuit F with input $i_1..i_n$ there are sometimes assignments to inputs which are called *enable* and *disable* values. A disable assignment to an input determines the output of the circuit, irrespective of the other inputs. Informally, the other inputs are assumed to be disabled. If an assignment is not an disable assignment, then it must be an enable assignment. This kind of assignment ensures that the value at the output does depend on the values at the other inputs.

Consider the circuit *C* in Figure 3.4 with inputs *a*, *b* and *c* and output *d* i.e. the output is a function g of the inputs:

d = g(a, b, c).

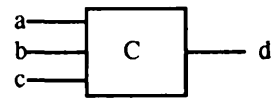


Figure 3.4

To generate a test pattern for a fault at node *a* , information about the value at node *a* must be 'transported' to node *d* so that it can be observed. This requires nodes b and c to assume values that do not make the output *d* independent of *a* i.e. enable input values. This establishes a **sensitive path** from *a* to *d*. This ensures that a change of logic value at node *a* is reflected by a change at *d*.

Another useful relationship between *a* and *d* is to make the logic value at node *d* independent of node *a*. This can be done by finding 'disabling' values for *b* and *c* which which produce a fixed value at *d*, no matter what value is present at *a*.

For the AND gate in Figure 3.3a, the output *c* can be made to always depend on the value at *a* by ensuring that *b* is 1. The relationship between *c* and *a* is then simple: *c* is always the same as *a*. To make the output independent of *a*, *b* is set to 0 which results in *c* always being 0 no matter what *a* is.

The OR gate in Figure 3.3b requires b to be 0 to make c depend on a and b to be 1 to mask the value at a. Some gates, like exclusive-or, can not be disabled.

Manual test pattern generation is presented first. Most automatic test pattern generation techniques are just formalisations of manual techniques, so many of the techniques in TPG and ATPG are essentially the same. Before considering composite combinational circuits, a test program is generated for a single gate.

The AND gate is to be tested for stuck-at faults. The first stage in manual test pattern generation is to prepare a fault-list. For the single-stuck-at-fault model, this means listing each node of the circuit for each stuck-at value. The list of faults to be covered in this case is:

$$\text{FAULTLIST(AND)} = \{a/0, a/1, b/0, b/1, c/0, c/1\}$$

Remember that if there are n nodes in a combinational circuit, then there will be $2n$ stuck-at faults. Figure 3.5 shows the six faulty circuits that correspond to the above faults. These faults modify the function of the AND gate: the modified function is shown next to each broken gate.

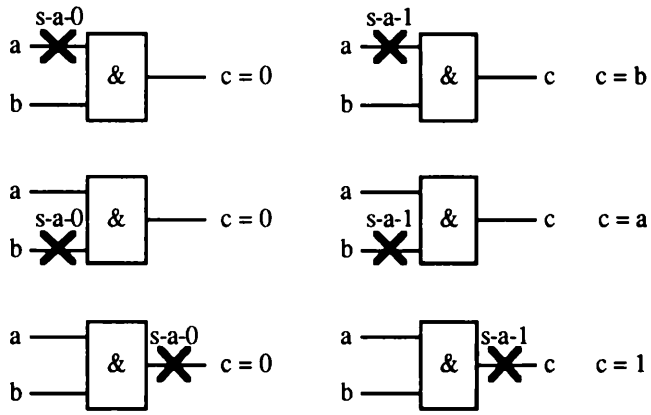


Figure 3.5: The stuck-at faults for an AND gate

If the fault $a/0$ occurs, then the output of the AND gate will always be 0. Consequently if the output is 1 then fault $a/0$ does not occur. A test pattern has been found for $a/0$ if the pattern produces a different output on a fault-free AND gate and an AND gate with fault $a/0$. To set the output node c of the AND gate to 1 requires a and b to be set to 1. Since this pattern produces differing outputs for a fault-free AND gate and an AND gate with fault $a/0$ then this is a test pattern for the fault $a/0$.

To test for $a/1$ the fault free condition a=0 must be established. As above, b must be set to 1 to make c depend on a. In a working circuit this would set c to 0 so a test for $a/1$ is abc=010. The reasoning behind generating tests for $b/0$ and $a/1$ is symmetrical: the tests are abc=111 for $b/0$ and abc=100 for $b/1$. To test for $c/0$ requires the fault free condition c=1 to be established. This can only be done by setting a=1 and b=1 so a test for this fault

is *abc*=111. To test for *c*/1 requires *c* to be set to 0. There are three different assignments to *a* and *b* that set *c* to 0: *ab*=00, *ab*=01 and *ab*=10. So any of *abc*=000, *abc*=010 and *abc*=100 are tests for *c*/1. These results are summarised in Table 3.1.

Faults	Test(s)
a/0	111
a/1	010
b/0	111
b/1	100
c/0	111
c/1	000 or 010 or 100

Table 3.1

The test 111 covers three faults: *a*/0, *b*/0 and *c*/0. For this gate using the single-stuck-at-fault model these faults are indistinguishable and form an equivalence class. A good choice to expose *c*/1 is the pattern 010 or the pattern 100 since these tests are needed anyway to expose other faults. A complete test program for an AND gate is {111, 100, 010}. This is only a saving of one test pattern compared to the test program generated by an exhaustive procedure i.e. {000, 010, 100, 111}. However, for more complex circuits the difference between the sizes of the test programs produced by functional and structural approaches becomes much greater. The structural method can take advantage of the connectivity information present to spot overlaps in tests and redundancies, whereas the functional approach has only the truth table or a boolean expression to work from.

The test pattern 111 above exposed three faults: *a*/0, *b*/0 and *c*/0. The reason for this is that these faults change the behaviour of the circuit in the same way i.e. transform it from $c = a \wedge b$ to $c = 0$.

This technique for testing an isolated gate extends naturally to the testing of composite combinational circuits. Tests for all the faults in the circuit C2 shown in figure 3.6 are now constructed by considering sensitive paths from the site of the fault to an observable output.

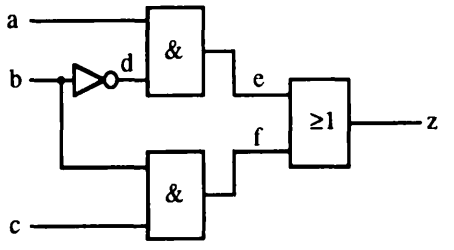


Figure 3.6: Composite Combinational Circuit C2

There are 7 nodes in this circuit so there are 14 possible faults. The fault list for circuit C2

is:

FAULTLIST(C2) = {*a*/0, *a*/1, *b*/0, *b*/1, *c*/0, *c*/1, *d*/0, *d*/1, *e*/0, *e*/1, *f*/0, *f*/1, *z*/0, *z*/1}

To test for *a*/0 the node a has to be set to the value opposite to the stuck-at value. There is no point in testing for *a*/0 with *a*=0 since there will be no difference between the fault-free and faulty outputs. The next step is to try and propagate the fault information towards an observable output, in this case *z*. The only way to get to *z* from *a* is through *e*, so the value at *e* must be made to depend in some way to the value at node *a*. The enabling input to an AND gate was shown earlier to be 1 so node *d* has to be set to 1 making *e*=*a*.

The assignment *d*=1 has to be justified by proceeding backwards towards the primary inputs to ensure that it is possible to set this node to 1. In this case it is easy to set *d*=1 by making the assignment *b*=0.

Having ensured that fault information can indeed be propagated from *a* to *e* the next step is to try and propagate fault information from *e* to *z*. The enabling input for an OR gate is 0 so node *f* must be set to 0. Since *b*=0 node *f* is 0 anyway, so no further assignments are required. The value at *c* is immaterial: neither 0 nor 1 will have any effect on the value of node *f*. The node *c* is assigned the value X to indicate that it can assume either logic value. Now the fault information at e is propagated to the observable output *z*: the relationship between *e* and *z* is *z*=*e*.

Putting all this together, the assignments *a*=1 and *b*=0 form a sensitive path from the site of the fault *a*/0 to *z* through *e*. The sensitive path is denoted by *a*=*e*=*z*. This states that in a working circuit C2, nodes *a*, *e* and *z* all have the same logic value. In a circuit which does not have the fault *a*/0 then *a*=*z*=1.

To emphasise the distinction between inputs and outputs, test patterns are written using a multiple assignment like *abc*/*d*=*pqr*/*s* where *a*, *b*, *c* are primary input nodes and *d* is a primary output node. The assignment states that node a is assigned logic value *p*, *b* logic value *q* etc. Since *c* can be any value, there are two tests for *a*/0: *abc*/*z*=100/1 (with c=0) and *abc*/*z*=101/1 (with c=1). Only one test is required to expose the fault. To test for *a*/0 the inputs are assigned *abc*=100 (or 101) and the value *z* observed. If *z*=0 (opposite from the fault-free value) then the fault *a*/0 is present.

The test for *a*/1 is similar to the test for *a*/0: the only difference is the fault-free condition *a*=0. As shown above, to propagate information from *e* to *z* the only assignment required is *b*=0. This forms the same sensitive path *a*=*e*=*z* so *z*=0 in a circuit that does not have the fault *a*/1. The tests for *a*/1 are *abc*/*z*=000/0 and *abc*/*z*=001/0.

Notice that it would have been possible to consider the tests for *a*/0 and *a*/1

simultaneously since finding a sensitive path does did not depend on the value at node *a*. The sensitive path *a*=*e*=*z* is highlighted by a heavy line in figure 3.7.

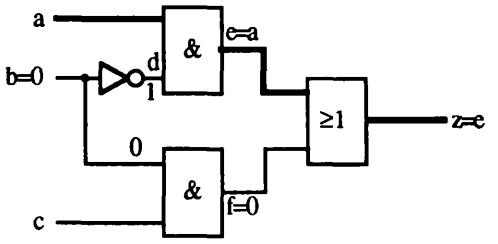


Figure 3.7: Sensitive path a=e=z

There are two possible sensitive paths from *b* to *z*: one through *e* and the other through *f*. The sensitive path through *e* is considered first. The value at node *d* is always opposite to the value at node *b*, so the first segment of the path is $b=\neg d$. To propagate information from *d* to *e* requires *a*=1 (the enabling input for an AND gate) so that $b=\neg d=\neg e$. To propagate fault information from *e* to *z* requires *f*=0 (the enabling input for an OR gate). There are three possible assignments that set *f*=0: *b*=0, *c*=0 and *bc*=00. However, the assignments *b*=0 and *bc*=00 commit *b* to be 0 requires using the value at node *b* which is the site of the fault so these patterns must be discarded. For this reason the assignment *c*=0 is chosen to establish the sensitive path $b=\neg d=\neg e=\neg z$. This states that in a working circuit the value at node *z* is always opposite to the value at node *b*.

The second possible sensitive path from *b* to *z* is through *f*. To make *f* depend on *b* requires *c*=1. To make *z* depend on *f* requires *e*=0. To set *e*=0 can be done by *a*=0. Again the value at node *b* cannot be used since this is the node under test. This establishes the sensitive path *b*=*f*=*z*, yielding the test *abc*/*z*=011/1 for b/0 and *abc*/*z*=001/0 for *b*/1.

To test for faults at node *b* two test patterns are required: one from the set {110/0, 011/1} to test for *b*/0 and one from the set {100/1, 001/0} to test for *b*/1. Thus there are four possible combinations of test patterns that expose both faults at node *b*.

To sensitize a path from *c* to *z* requires *b*=1 to make *c*=*f*. To make *z*=*f* requires *e*=0. No further assignments are required since the assignment *b*=1 causes *d*=0 which results in *e*=0. This gives the sensitive path *c*=*f*=*z*. The tests for *c*/0 are *abc*/*z*=X11/1 and the tests for *c*/1 are *abc*/*z*=X10/0.

The tests for the other faults are found in a similar manner.

This completes the first phase of manual test pattern generation for a very small circuit. Test patterns have been generated for all 14 possible faults. This technique is tedious and error prone. For large circuits, such manual calculations are not practical.

The faults and expanded test patterns that expose them are summarised in table 3.2.

Faults	Input test pattern
a/0	100 or 101
a/1	000 or 001
b/0	110 or 011
b/1	100 or 001
c/0	011/1 or 111/1
c/1	010/0 or 110/0
d/0	100 or 101
d/1	110
e/0	100 or 101
e/1	000 or 001 or 010 or 110
f/0	011 or 111
f/1	000 or 001 or 010 or 110
z/0	100 or 101 or 011 or 111
z/1	000 or 001 010 or 110

Table 3.2: Test for stuck-at faults in C2

This manual test pattern generation technique has produced all eight possible input patterns to test the three input circuit C2. However, not all eight test patterns need be used because most patterns expose more than one fault. By choosing patterns carefully, the number of test vectors required to test for every fault can be substantially reduced. The information in Table 3.2 is represented in Table 3.3 which shows the faults covered by each test pattern. Such a table is called a **fault-matrix**.

T/F	a/0	a/1	b/0	b/1	c/0	c/1	d/0	d/1	e/0	e/1	f/0	f/1	z/0	z/1
000		✓								✓		✓		✓
001		✓		✓						✓		✓		✓
010						✓				✓		✓		✓
011			✓		✓						✓		✓	
100	✓			✓			✓		✓				✓	
101	✓						✓		✓				✓	
110			✓			✓		✓		✓		✓		✓
111					✓						✓		✓	

Table 3.3 Fault matrix for circuit C2.

Only enough rows (tests) have to be chosen to ensure that there is at least one tick under each fault. The first step is to identify columns (faults) that have only one tick. These faults have only one test that exposes them. Such a test is called an **essential test** and must be used in the test program. In table 3.3 *d*/1 is only covered by one tick corresponding to test 110. Test 110 also covers the faults *b*/0, *c*/1, *d*/1, *e*/1, *f*1 and *z*/1. The table is now reduced by removing columns *d*/1, *b*/0, *c*/1, *d*/1, *e*/1, *f*/1 and *z*/1 and the row 110. This results in table 3.4:

T/F	a/0	a/1	b/1	c/0	d/0	e/0	f/0	z/0
000		✓						
001		✓	✓					
010								
011				✓			✓	✓
100	✓		✓		✓	✓		✓
101	✓				✓	✓		✓
111				✓			✓	✓

Table 3.4: Fault-matrix after removing the essential test 110.

There are no essential tests in this table i.e. each fault is covered by more than one test. By inspection it can be deduced that three tests are required to cover all the remaining faults. One suitable choice of test patterns might be 100 (because it covers all the faults except *a*/1 and *f*/0), 000 (because it covers *a*/1) and 111 (because it covers f/0). Alternatively, a boolean expression can be derived from the table which can be reduced to show that at least three tests are required and that there are six ways to choose them.

By simplifying a fault-matrix the test program has now been reduced from eight patterns to four, namely {110, 100, 000, 111}. This simplification technique is directly analogous to the technique used in Quine-McCluskey [McCluskey 62] boolean simplification to find prime implicants.

Test pattern generation for realistic circuits involves manipulating vast amounts of information. A useful preprocessing stage to test pattern generation is **fault-collapsing**. This technique reduces the size of the fault-list by identifying faults which are **indistinguishable**. Two faults are indistinguishable if they are covered by the same test patterns. Table 3.3 is rearranged in table 3.5 to highlight indistinguishable faults. The faults {*a*/0, *d*/0, *e*/0} are indistinguishable, so they can be replaced by just one fault in the fault-list.

T/F	a/0	d/0	e/0	e/1	f/1	z/1	c/0	f/0	a/1	b/0	b/1	c/1	d/1	z/0
000				✓	✓	✓			✓					
001				✓	✓	✓			✓		✓			
010				✓	✓	✓						✓		
011							✓	✓		✓				✓
100	✓	✓	✓								✓			✓
101	✓	✓	✓											✓
110				✓	✓	✓				✓		✓	✓	
111							✓	✓						✓

Table 3.5: Groups of indistinguishable faults.

The other non-singleton groups are {*e*/1, *f*/1, *z*/1} and {*c*/0, *f*/0}. By choosing just one representative from each set, the fault-list can now be reduced to:

{*a*/0, *e*/1, *c*/0, *a*/1, *b*/0, *b*/1, *c*/1, *d*/1, *z*/0}

This has removed five faults from the fault-list which results in a substantial saving in test pattern generation effort. The fault-list can be reduced even further by finding **fault dominance** in table 3.6. A fault R is dominated by a fault S if the ticks in R's row are a subset of the ticks in S's row and is denoted by S → R. There are several instances of fault dominance in table 3.5:

a/0 → *z*/0
a/1 → *e*/1
c/0 → *z*/0
c/1 → *e*/1
d/1 → *b*/0, *c*/1, *e*/1

Once the dominated faults have been removed the fault-list is reduced to just five elements:

{*a*/0, *a*/1, *b*/1, *c*/0, *d*/1}

The reduction of the fault list from 14 to 5 is a large saving but not a typical one. Fault collapsing usually halves the size of the fault list.

Instead of building tables from scratch for each circuit to be analysed for fault-collapsing, it is possible to determine characteristics about isolated gates and combine these to deduce information about a circuit made from these gates. Table 3.6 shows fault-collapsing information for AND, OR and NOT gates (Figure 3).

Gate	Indistinguishable faults	Fault Dominance
AND	{a/0, b/0, z/0}	a/1, b/1 → z/1
OR	{a/1, b/1, z/1}	a/0 → b/0 → z/0
NOT	{a/0, z/1}; {a/1, z/0}	None

Table 3.6: Fault-collapsing information for isolated gates.

The techniques of sensitive path analysis and fault collapsing have to be applied with care to circuits containing reconvergent fanout. An example of such a circuit is shown in figure 3.8.

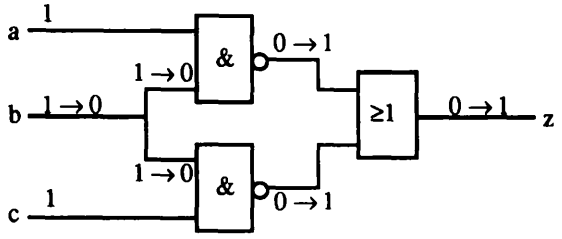


Figure 3.8: Positive reconvergence under fault b/0.

The test pattern is *abc*/*z*=111/0 and the fault under consideration is *b*/0. The circuit is annotated with differences between the fault-free and faulty circuit. 1 → 0 means that a node which has logic value 1 in a fault free circuit assumes logic value 0 when the fault under consideration is present and similarly for 0 → 1. The interesting aspect of the fault propagation in this circuit is that there are two paths simultaneously sensitized from the site of the fault to the primary output *z*. The fan-out is responsible for allowing more than one path and the reconvergence at the last NAND gate combines the results of the two sensitive paths to produce a sensitive output. The term **dual-path sensitization** is used to describe this situation. Both the inputs to the last NAND gate must be sensitive to the fault *b*/0 to sensitize the output. One change in only one input does not cause a change in the output. **Positive reconvergence** occurs when two sensitive paths reconverge to reinforce each other. Each path alone does not create a sensitive output: both must be sensitive.

The circuit in figure 3.9 shows an example of negative reconvergence: this is where information from two sensitive paths reconverge in a manner which makes it impossible to extend the sensitive path.

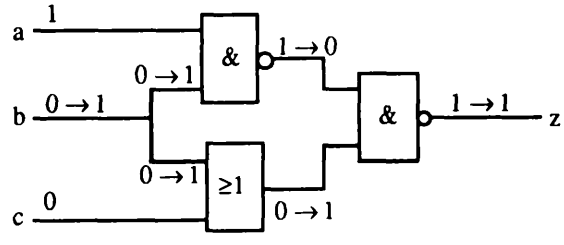


Figure 3.9: An example of negative reconvergence

The test *b*=0 (sensitize the site of the fault), *a*=1 (enable top NAND gate) and *c*=0 (enable OR gate) is applied to the primary inputs. Two sensitive paths exist up to the last NAND gate. Each node assumes value 0 instead of 1 when fault *b*/0 occurs). Unfortunately, these sensitive paths combine at the last NAND gate to produce an output of 1 which is the same as the output for the fault-free version of the circuit. Consequently the input pattern *abc*=100 is not a test pattern for the fault *b*/1.

The implication of fanout and reconvergence is that care must be taken when examining paths through such circuits. If a path is blocked through two alternative paths independently then it may not be blocked if both paths are sensitized simultaneously. Also, tests generated for reconvergent fanout circuits have to be checked to ensure that negative reconvergence does not take place.

Fanout and reconvergence also affects the results obtained for fault-collapsing. Applying the rules in table 3.6 to the circuit in figure 3.8 the indistinguishable faults for the bottom AND gate are found to be {*b*/0, *c*/0, *f*/0}. However, comparing this with the fault matrix reveals that these faults are not actually indistinguishable: *b*/0 is different from {*c*/0, *f*/0}.

The problem is that the test 111 which covers *f*/0 does not cover *b*/0 because negative reconvergence causes the fault-effect of *f* to be cancelled by the fault-effect at *e*. Also, the test 110 which covers b/0 does not cover f/0 because it uses a different sensitive path through *e* instead of *f*. Both of these problems are a result of the fan-out that occurs at node *b*. One solution is to remove all the implications resulting from the fanout node, giving the correct result at gate 3: {*c*/0, *f*/0} ; *c*/1 $\rightarrow$ *f*/1 i.e. *c*/0 and *f*/0 are equivalent and *c*/1 dominates *f*/1.

3.1 Boolean Differences

Another technique for generating tests for faults in combinational circuits is the **boolean difference method** [Sellers 68]. This method employs differential equations to describe test patterns. There is a strong analogy between differential equations over boolean values and those over real numbers and they share many common properties. This is because boolean algebra and the real number system are both examples of rings in mathematics.

Consider the problem of generating a test for a single output circuit characterised by the function $F(x_1,...x_n)$ where $x_1..x_n$ are the primary inputs. First, the problem of generating tests for primary inputs is solved. This is then extended to internal nodes.

A fault is testable if a change of logic value at the site of the fault also produces a change at an observable output. Assume that input x_i is to be tested for a fault. The set of tests that expose faults at x_i are captured formally by using the exclusive-or operation:

$$F(x_1,...,0,...x_n) \oplus F(x_1,...,1,...x_n) = 1$$

If the exclusive-or of the outputs with and without the fault is 0 then this means that there is no difference in the circuit response between the fault-free and faulty circuits. The left hand side of the above equation is the **boolean difference**, and is written as:

$$\frac{dF(X)}{dx_i} = F(x_1,...,0,...,x_n) \oplus F(x_1,...,1,...,x_n)$$

where $X = x_1,...x_n$.

The boolean difference describes all the conditions (i.e. values of the inputs) for which the output of F depends only on x_i. A test for a fault at the site of the primary input x_i exists if $dF(X)/dx_i = 1$ i.e. the output of the function is negated by the presence of the fault for certain input assignments. However, if $dF(X)/dx_i = 0$ then there are no input assignments that cause the output to be complemented when the fault is present. This makes the fault at site x_i undetectable.

The boolean difference sensitizes a path from the site of the fault to an observable output. To generate a test pattern, the site of the fault has to be sensitized by setting it to the opposite value of the fault. This is also done by choosing suitable assignments to the primary inputs. A test is a consistent combination of patterns generated by the boolean difference and the sensitization of the site of the fault i.e. the logical conjunction of the boolean difference and the condition for sensitizing the site of the fault.

Consider the example of testing for the fault x_i stuck-at-0. The condition required to sensitize the site of the fault is $x_i = 1$. Tests for this fault are given by solutions to the expression:

$$\frac{dF(X)}{dx_i} \cdot x_i = 1$$

This derivative describes the conditions required to from a sensitive path from the site of the fault (i.e. the primary input x_i) to the output and the x_i term sensitizes the site of the fault by requiring x_i to be 1 (opposite of the stuck-at value). By similar reasoning, the tests for x_i stuck-at 1 are given by the expression:

$$\frac{dF(X)}{dx_i} \cdot \overline{x_i} = 1$$

This expression requires x_i to be sensitized by assigning to it the value 0.

Tests can be generated for internal nodes by thinking of the node to be tested as being an extra primary input to the circuit. Consider the generation of a test for an internal node k using boolean differences. First, the logic value at k is expressed in terms of the primary inputs i.e. $k = g(x_1,...,x_n)$ where g is a boolean function. The reason for this is

that k might not depend on all n inputs. Now k can be added to the parameter list:

$$\frac{dF(X,k)}{dk} = F(x_1,\ldots,x_n,k) \oplus F(x_1,\ldots,x_n,k)$$

Now k can be replaced in the expansions of the above expression by expressing it in terms of the primary inputs. This relationship is given by g.

To illustrate boolean differences, this method is used to generate tests for the circuit in figure 3.6. Tests for this circuit have already been generated by using sensitive paths. The function of the circuit is

$$F(a, b, c) = (a \wedge \neg b) \vee (b \wedge c)$$

The primary inputs are dealt with first. Let X = (a, b, c).

• Node a.

$$\frac{dF(X)}{da} = F(0,b,c) \oplus F(1,b,c)$$

$F(0,b,c) = (0 \wedge \neg b) \vee (b \wedge c) = 0 \vee (b \wedge c) = b \wedge c$
$F(1,b,c) = (1 \wedge \neg b) \vee (b \wedge c) = \neg b \vee (b \wedge c)$

$F(0,b,c) \oplus F(1,b,c) = \neg b$

This is only 1 if b=0. Note that this is the condition that is required to establish a sensitive path from a to z. Let X denote a don't care assignment. To test for a/0 requires a=1 so a test for this fault is abc/z=10X/1. Testing for a/1 requires a=0 so the test pattern is abc/z=00X/0.

• Node b.

$$\frac{dF(X)}{db} = F(a,0,c) \oplus F(a,1,c)$$

$F(a,0,c) = (a \wedge 1) \vee (0 \wedge c) = a$
$F(a,1,c) = (a \wedge 0) \vee (1 \wedge c) = c$

$F(a,0,c) \oplus F(a,1,c)$
$= a \oplus c$ { defn. of F }
$= a \wedge \neg c \vee \neg a \wedge c$ { defn. of $\oplus$ }

The boolean difference is 1 when either a is 1 or c is 1 but not both. To test for b/0 requires b=1 giving the tests abc/z=110/0 (with a=1) and abc/z=011/1 (with c=1). To test for b/1 requires b=0 giving the tests abc/z=100/1 and abc/z=001/0.

• Node c.

$$\frac{dF(X)}{dc} = F(a,b,0) \oplus F(a,b,1)$$

$F(a,b,0) = (a \wedge \neg b) \vee (b \wedge 0) = a \wedge \neg b$
$F(a,b,1) = (a \wedge \neg b) \vee (b \wedge 1) = (a \wedge \neg b) \vee b$

$F(a,b,0) \oplus F(a,b,1) = b$

The solution to the differential is b=1. Tests for c/0 are abc/z=X11/1 and test for c/1 are abc/z=X10/0.

- Node d.

Make d a pseudo-input $d = \neg b$ so $F(a,b,c,d) = a \wedge d \vee b \wedge c$.

$$\frac{dF(X,d)}{dd} = F(a,b,c,0) \oplus F(a,b,c,1)$$

$F(a,b,c,0) = 0 \vee b \wedge c = b \wedge c$
$F(a,b,c,1) = a \vee b \wedge c$

$F(a,b,c,0) \oplus F(a,b,c,1) = a \wedge (\neg b \vee \neg c)$

For d/0 this yields the tests abc/z=10X/1. The test for d/1 is abc/z=110/0.

The tests for the other faults are obtained in a similar manner.

3.5 Deductive Fault Simulation

3.5.1 Introduction to Deductive Fault Simulation

The generation of a test pattern for a given fault is very expensive. Once a test pattern for a particular fault has been generated, it is often the case that this test pattern will also reveal other faults. Employing a test pattern generation system to rediscover test patterns at great cost is not necessary. It is possible to perform an analysis which examines a circuit for a given test pattern in order to ascertain which faults it exposes.

A fault simulator takes as input a test pattern and a circuit description and produces as output a list of faults that can be detected by this pattern. Some fault simulators work by simulating defective versions of the circuit, whilst others simulate the working version and deduce from the correct behaviour which faults are detectable at the primary outputs. A deductive fault simulator [Armstrong 72] belongs to the latter category. A circuit

represented as a product of sums of AND, OR, NAND and NOR gates may be transformed into a set expression which yields the faults required.

Deductive fault simulation works by propagating lists which represent faults detected at predecessor gates. For each gate, the subset of faults that is passed is modified to describe what faults are propagated to the output of the gate. We assume that the single stuck-at fault model is employed.

The output of each gate is the true logic value and a set of faults that the output line is sensitive to. A set X is 'negated' w.r.t. another set Y by complementing it with the union of X and Y.The circuit is transformed into a set expression by using the following rules.

1. Replace all AND gates by set intersection $\cap$
2. Replace all OR gates by set union $\cup$
3. Negate a fault set if its true value is 1
4. Add to each output the appropriate stuck-at-fault
5. Simplify the resulting expression

It is not obvious why these rules describe a method for correctly propagating detectable faults. For deductive fault simulation, wires carry fault propagation information as well as logic values. The fault information is represented as a set of faults that a given wire is sensitive to.

The faults that are propagated through a 2-input AND gate for all 4 possible input values are characterised by set expressions. Let the logic inputs to the AND gate be x and y and let the fault sets be A and B respectively.

• Pattern 00/0. Any fault that causes the output to be different from its true value is a detectable fault. In this case, the output has to be 1 for the effect of some previous fault to be detected. This requires both inputs to be 1 for the output to be 1 i.e. we want any fault that changes from 0 to 1 ($0 \rightarrow 1$) the first and second input. This means that we want the faults that are common to sets A and B i.e. $A \cap B$.

• Pattern 01/0. We want to choose those faults that cause the output to change to 1 i.e. those faults that change the first input. It is wrong to simply choose all the faults in set A because some of these faults may also be in set B. Consider the effect of a fault in $A \cap B$: this causes the first input to be faulty ($0 \rightarrow 1$) and the second input to be faulty ($1 \rightarrow 0$). The result is that the output is still 0 (not different from its true value) so such faults are not detectable. We want those faults that are in A but not in B i.e. $A - (A \cap B)$. We may rewrite this as $A \cap \neg B$.

• Pattern 10/0. By a similar argument it can be seen that the faults propagated are represented by the set expression $B - (A \cap B)$ i.e. $\neg A \cap B$.

• Pattern 11/1. We now want to pass any faults that cause the output to become 0 i.e. faults that cause either of the inputs to be 0. The effect of any fault in A or B is to set one or more of the inputs to 0 so we can pass all the faults in A and B i.e. A ∪ B. This may be rewritten as follows: $A \cup B = \neg\neg(A \cup B) = \neg(\neg A \cap \neg B)$.

To each of the set expressions above we must remember to add the fault detectable at the output. The following table summarises the results:

Pattern	Set Expression	
00/0	$(A \cap B) \cup \{z/1\}$	1
01/0	$(A \cap \neg B) \cup \{z/1\}$	2
10/0	$(\neg A \cap B) \cup \{z/1\}$	3
11/1	$\neg(\neg A \cap \neg B) \cup \{z/0\}$	4

Notice the pattern:

- if x=1 then A is complemented (lines 3 and 4)
- if y=1 then B is complemented (lines 2 and 3)
- if z=1 then the first part of the set expression is complemented (line 4)

This now provides an explanation for the rules given earlier for deductive fault simulation. These rules are represented using Venn diagrams in figure 3.10.

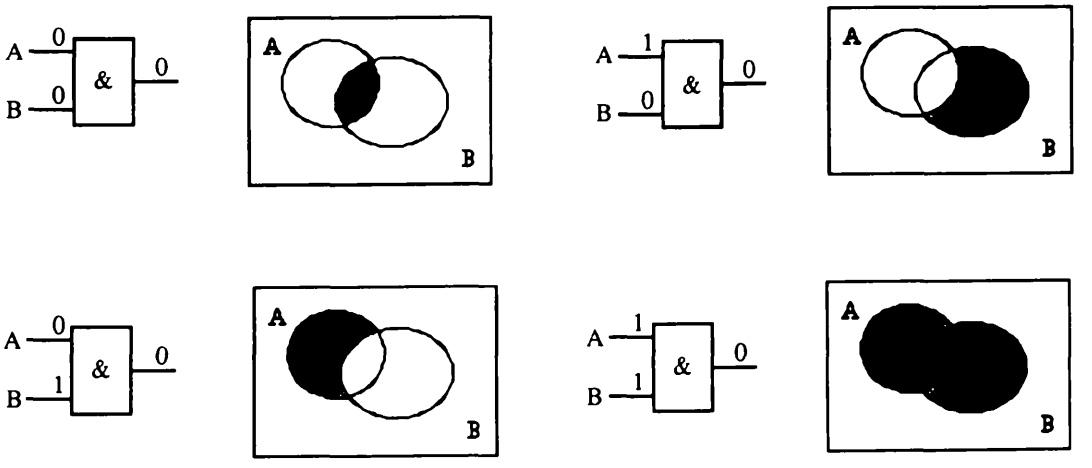


Figure 3.10: Venn Diagrams corresponding to the deductions rules for an AND gate.

We can deduce similar rules for an OR gate:

Pattern	Set Expression
00/0	$(A \cup B) \cup \{z/1\}$
01/1	$\neg(A \cup \neg B) \cup \{z/0\}$
10/1	$\neg(\neg A \cup B) \cup \{z/0\}$
11/1	$\neg(\neg A \cup \neg B) \cup \{z/0\}$

These rules are similar to the rules for an AND gate except that intersection has been replaced by union and the output fault is different.

An inverter will pass all faults at its input and add to the fault list the fault that can be detected at its output. Rules for other gates can be easily derived. Alternatively, any combinational circuit can be re-expressed in terms of AND, OR and NOT and the analysis carried out using the rules given above.

3.5.2 An Example of Deductive Fault Simulation

Figure 3.11 illustrates deductive fault simulation with an example circuit. The input pattern is 110. Each arrow is a node named by the letter in the centre and the faults propagated along this node are shown in the set above. The fault-free logic value of each node is also shown.

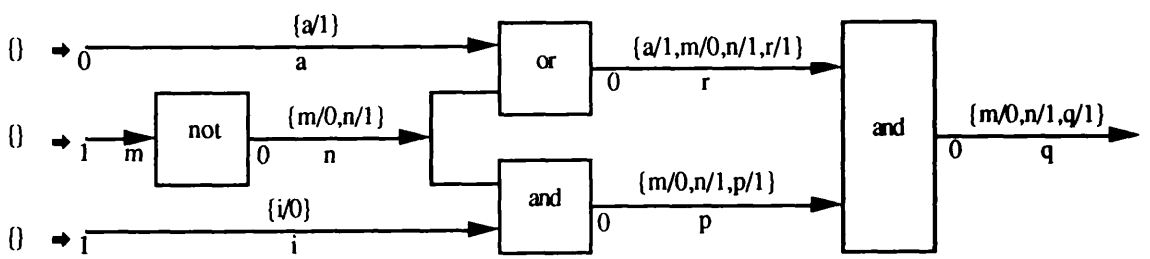


Figure 3.11

Consider the AND gate with output node q. The true output is 0 so we want to pass those faults that will set both the inputs to 1 i.e. we have to 'fault' both the inputs. These faults are obtained by taking the intersection of the two input fault sets (see the rules for and gate with pattern 00/0). Since this is the primary output of the circuit, the test pattern 110 detects the faults given by the following set expression:

$$\{m/0, n/1, p/1\} \cap \{a/1, m/0, n/1, o/1\} \cup \{q/1\} = \{m/0, n/1, q/1\}$$

The complete sequence of deductions is presented below. The function **ded** describes the deduced faults at a given node.

ded(n)	= ded(m) $\cup$ {n/1}	(NOT gate)
	= {m/0} $\cup$ {n/1}	
	= {m/0, n/1}	
ded(r)	= **ded**(a) $\cup$ **ded**(n) $\cup$ {r/1}	(OR gate)
	= {a/1} $\cup$ {m/0, n/1} $\cup$ {r/1}	
	= {a/1, m/0, n/1, r/1}	
ded(p)	= **ded**(n) $\cap$ $\neg$**ded**(i) $\cup$ {p/1}	(AND gate)
	= {m/0, n/1} $\cap$ $\neg${i/0} $\cup$ {p/1}	
	= {m/0, n/1} $\cup$ {p/1}	
	= {m/0, n/1, p/1}	
ded(q)	= **ded** (r) $\cap$ **ded**(p) $\cup$ {q/1}	(AND gate)
	= {a/1, m/0, n/1, r/1} $\cap$ {m/0, n/1, p/1} $\cup$ {q/1}	
	= {m0, n/1, q/1}	

Because deductive fault simulation works with a correct version of the circuit, reconvergent fanout problems do not arise.

3.6 Testability Measure

3.6.1 Introduction

Since our objective is to increase the testability of digital circuits, we should have some precise quantitative measurement of 'testability'. In the literal sense, most designs are testable, since it is possible to apply all input combinations and observe the output. However, we feel that if a design can be tested to a high degree by applying a much smaller set of test patterns, then it must be more testable. This section describes a few measures of testability.

3.6.2 ATPG Approach

An Automatic Test Pattern Generation program is used to generate tests and to compute

the fault coverage. The running time of this program gives an idea of how difficult it is to test a particular circuit. However, the run time can be very long, and there is no data about how to improve the testability of the circuit.

Several programs have been developed which examine the structure of a circuit in order to estimate its testability without having the incur the expense of running an ATPG program.

3.6.3 Testability Measure (TM) Programs

These testability measure programs analyse the circuit to estimate the running cost of generating test patterns (which in turn gives an idea of how testable the circuit is). As they accumulate this data, they are able to pin-point areas of the design that are difficult to test. The components in these areas may then be redesigned to allow greater testability (e.g. by incorporating asynchronous set/reset lines).

There is no simple link between circuit characteristics and testability. The circuit parameters used by testability measure programs are heuristic and based on the experience of studying ATPG programs. Thus, different testability measure programs use different circuit characteristics to estimate testability.

Testability measure programs are assessed by running them on circuits which have already been analysed by an ATPG program. A monotonic relation between the testability program run time and the ATPG run time is offered as 'proof' that the testability measure program produces a good measure of testability.

Not surprisingly, all the testability measure programs are based around the ideas of controllability and observability.

3.6.4 TMEAS

In TMEAS [Grason 79], each link has associated with it an observability value OY and a controllability value CY. These are normalised between 0 (the worst) and 1 (the best). Thus, for primary inputs, CY = 1 and for primary outputs OY = 1. Each component in the circuit has associated with it a controllability transfer factor, CTF, and an observability transfer factor, OTF. These are used to build two systems of N (N = the number of components) simultaneous equations which are used to compute the CY and

OY values for internal links.

Sequential components are dealt with by introducing implicit feedback loops (to represent state transitions) into the circuit. For a particular component, the input controllability is defined to be the average of the input link controllabilities and the output controllability is defined to be the average of the output link controllabilities.

The CTF is defined by considering the uniformity of the input-output mapping, normalised between 0 and 1. A circuit whose output was 0 for half the possible input values and 1 for the other half would have a CTF of 1. For an n-input single-output component that has output = 0 for only one component, the CTF is 2^{1-n}.

3.6.5 The SCOAP Testability Measure

The SCOAP [Goldstein 79] testability measure assigns a 6-element vector to each node of the circuit. The six elements describe how easy it is to set a combinational/sequential node to 0 or 1 and how easy it is to propagate the value on some combinational/sequential node to an observable output. For the present we shall restrict ourselves to combinational circuits, so we shall only be interested in obtaining 3 values for each node[1]:

(a) **set0** (n) - a measure of how easy it is to set node n to logic 0.
(b) **set1** (n) - a measure of how easy it is to set node n to logic 1.
(c) **obsv** (n) - a measure of how easy it is to observe the value at node n.

The larger the value for the above measures, the greater is the degree of difficulty for controlling/observing a given node. The following rules are used for calculating the SCOAP values for the 2 input nodes and 1 output node of a 2 input AND gate.

SCOAP Rules for ⟨*x*,*y*⟩AND *z*

set0	(*z*) = min [**set0** (*x*), **set0** (*y*)] + 1	Rule 1
set1	(*z*) = **set1** (*x*) + **set1** (*y*) + 1	Rule 2
obsv	(*x*) = **set1** (*y*) + **obsv** (*z*) + 1	Rule 3
obsv	(*y*) = **set1** (*x*) + **obsv** (*z*) + 1	Rule 4

Rule 1 describes how easy it is to set the output *z* of an AND gate to 0 i.e. **set0** (*z*).

---

[1]These values were called CC0 (**set0**), CC1 (**set1**) and CO (**obsv**) in the original literature.

This can be done by setting either of the inputs to 0. The SCOAP rules choose the input which is easier to set to 0 (i.e. has the lowest measure/cost associated with it) and then adds 1 as a penalty for propagating the result past the AND gate. To set the output of an AND gate to 1 requires both the inputs to be set to 1. Thus, the formula for **set1** (z) adds the difficulty of setting both x and y to 1 and then adds a fixed penalty of 1 for the AND gate.

Rules 3 and 4 describe the observability costs for the input nodes x and y. To observe the value at node x, node y has to be set to 1 so that the output depends only on x. Thus a cost of **set1** (y) has to be incurred. Then we have to add the cost of transporting the value from the output of the AND gate z to an observable output. This can be recursively specified as **obsv** (z). Finally we add a penalty of 1 for propagating the value across the AND gate. Rule 4 is similar.

The table below show the rules for OR gates and NOT gates:

SCOAP Rules for ⟨x,y⟩OR z (def 2)

set0 (z) = **set0** (x) + **set0** (y) + 1
set1 (z) = min [**set1** (x), **set1** (y)] + 1
obsv (x) = **set0** (y) + **obsv** (z) + 1
obsv (y) = **set0** (x) + **obsv** (z) + 1

SCOAP Rules for x NOT y (def 3)

set1 (y) = **set0** (x) + 1
set0 (y) = **set1** (x) + 1
obsv (x) = 1

For inputs, **set0** and **set1** are 1 and for outputs **obsv** is 0. This reflects that fact that only one assignment has to be made to set a primary input to a particular value. Also, no assignments are required to observe an output. The controllability of the primary outputs and the observability of the primary inputs are values of little interest.

The actual costs returned by the SCOAP measure represent the number of combinational node assignments required to control/observe a given node plus some notion of depth. This is a heuristic that tries to estimate the difficulty of generating test patterns for the given node (i.e the *testability* of a node). SCOAP gives good values for small to medium circuits, but deviates from true values for larger circuits.

Figure 3.12 shows an example circuit for which SCOAP values shall be computed.

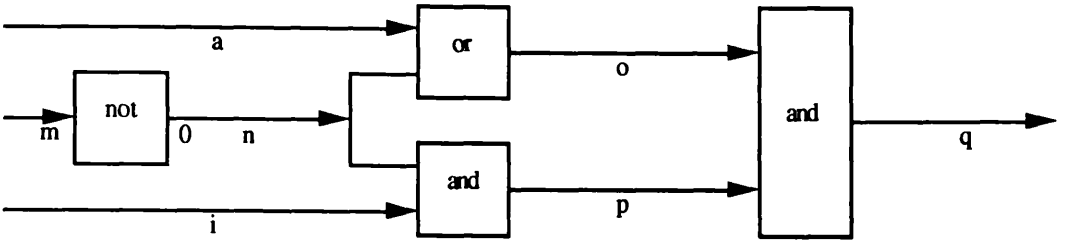


Figure 3.12

This circuit can be described by the following Ruby expressions:

[ι, NOT ; split, ι] ; reorg1 ; [AND, OR] ; AND

⟨a, ⟨b,c⟩, d⟩ reorg1 ⟨⟨a,b⟩, ⟨c,d⟩⟩

The SCOAP values are computed by first evaluating the controllability measures (**set0** and **set1**) and then using these to compute the observability measure (**obsv**).

set0(a) = 1	**set1**(a) = 1	Inputs
set0(m) = 1	**set1**(m) = 1	
set0(i) = 1	**set1**(i) = 1	
set0(n) = 1 + 1 = 2	**set1**(n) = 1 + 1 = 2	(NOT gate)
set0(o) = **set0**(a) + **set0**(n) + 1 = 1 + 2 + 1 = 4		(OR gate)
set1(o) = min [**set1**(a), **set1**(n)] + 1 = min [1, 2] + 1 = 2		
set0(p) = min [**set0**(n), **set0** (i)] + 1 = min [2, 1] + 1 = 2		(AND gate)
set1(p) = **set1**(n) + **set1**(i) + 1 = 2 + 1 = 4		
set0(q) = min [**set0** (o), **set0**(p)] + 1 = min [2, 2] + 1 = 3		(AND gate)
set1(q) = **set1**(o) + **set1**(p) + 1 = 2 + 3 + 1 = 6		

Although a controllability value has been computed for the output q, the SCOAP rules define the outputs to be 'uncontrollable' by setting them to infinity.

set0(q) = ∞ **set1**(q) = ∞

The controllability information calculated is now used to compute the observability values:

obsv(q)	= 0	Output
obsv(o)	= **set1**(p) + **obsv**(q) + 1 = 3 + 0 + 1 = 4	(AND gate)
obsv(p)	= **set1**(o) + **obsv**(q) + 1 = 2 + 0 + 1 = 3	
obsv(n)	= min [**set0**(a) + **obsv**(o), **set1**(i) + **obsv**(p)] + 1	(Split)

$$= \min [5, 4] + 1 = 5$$

$$\textbf{obsv}(m) = 1 + \textbf{obsv}(n) = 1 + 5 = 6 \qquad \text{(NOT gate)}$$

$$\textbf{obsv}(a) = \textbf{set0}(n) + \textbf{obsv}(o) + 1 = 2 + 4 + 1 = 7 \qquad \text{(OR gate)}$$

$$\textbf{obsv}(i) = \textbf{set1}(n) + \textbf{obsv}(p) + 1 = 2 + 3 + 1 = 6 \qquad \text{(AND gate)}$$

A numerical value has been obtained for the observability of the primary inputs. However, the SCOAP rules define the primary inputs to be infinitely unobservable:

$$\textbf{obsv}(a) = \infty \qquad \textbf{obsv}(m) = \infty \qquad \textbf{obsv}(i) = \infty$$

This completes the calculation of the SCOAP values for a simple combinational circuit. These values are used to find areas of poor controllability and observability so that the circuit can be redesigned to make it more testable. The information computed above is shown graphically in figure 3.13.

Node o has the largest 0-controllability measure at 4 and node p has the highest value for 1-controllability (also 4). These values are not much larger than the average value of 1.83 so in this case redesign is not necessary.

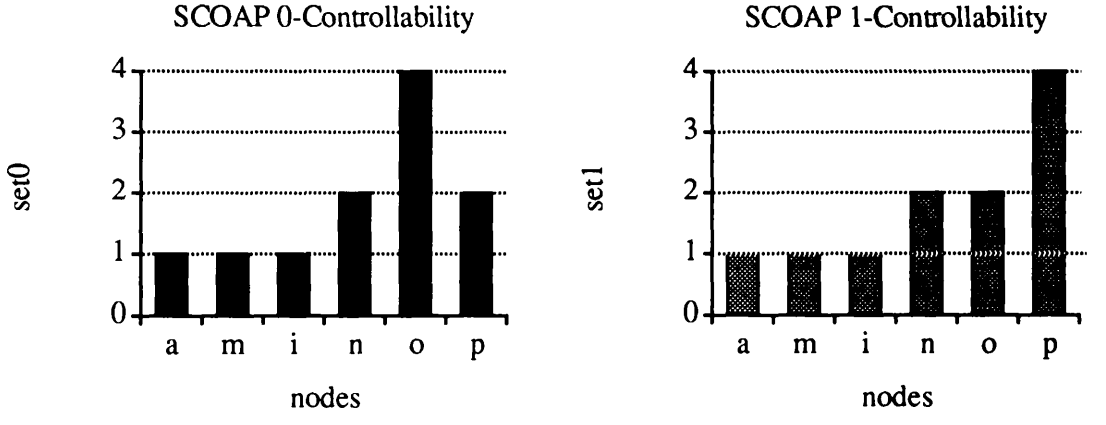


Figure 3.13: SCOAP controllability for example circuit.

The observability values are shown in figure 3.14. As expected, the nodes closer to the primary inputs are the most difficult to observe. The increase in observability measure from the outputs to the inputs (right to left in the figure) is small and constant so there are no nodes that need special treatment in this example.

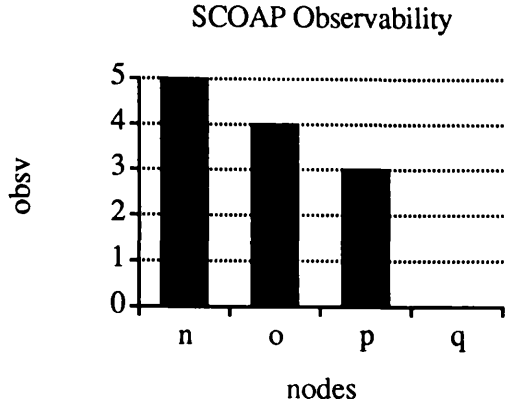


Figure 3.14: SCOAP controllability measure for the example circuit.

3.7 Design for testability techniques

Design for testability implies some modification to the circuit to enhance the process of test pattern generation and application. The techniques to enhance testability have been categorized into three main groups:

1 *ad hoc* methods
2 Structured approaches
3 Built in test and self-test methods

3.7.1 Ad hoc methods

These methods evolved due to the need to solve particular testing problems, rather than trying to solve the task of testing in general by using a design methodology.

3.7.1.1 Test point insertion

Test points are routed into the circuit to make certain internal nodes more accessible in order to either control or observe the signal value at the node.

3.7.1.2 Pin amplification

It is desirable to reduce number of pins used by a design. Testing requires extra data to be

input/output and therefore extra pins. This cost can be reduced by multiplexing input/output pins to perform the additional function of acting as test input and outputs. The disadvantage of this approach is that it slows down the circuit.

3.7.1.3 Blocking or degating logic

In this technique additional gates are incorporated into the design to inhibit data flow along certain paths, thus partitioning the circuit into smaller modules for the purposes of testing. Blocking gates are two input gates, one input is the normal data line whilst the other is the controlling or blocking signal which can be controlled from a test input.

3.7.1.4 Control and observation switching

In this technique signal lines whose logic values are either easily controlled or observed are identified in the circuit and these are used in conjunction with demultiplexers/multiplexers to improve access to nearby nodes, whose logic values are difficult to control or observe.

3.7.1.5 Test state registers

Test state registers can be attached to various internal nodes. These registers may have values shifted into them to set these nodes to a particular value or they may be shifted out so the value present at the node may be examined.

Ad hoc methods for improving the testability of a circuit have the advantage of not imposing severe constraints on the designer. However, a disadvantage is that these methods cannot be automated, and consequently there is no software support for these techniques of designing for testability.

3.7.2 Structural Approaches

These design methods are incorporated into the design from the outset rather than as an afterthought as with *ad hoc* methods. Most structural techniques use hard and fast rules allowing software support.

The objective in developing the structural approach was to facilitate the testing of complex sequential circuits. These methods increase the controllability and observability of the internal state elements, essentially transforming the testing of a sequential circuit into the simpler task of testing a combinational circuit.

The level sensitive scan design and the scan/set design are two of the more popular methods in industry.

3.7.2.1 Level sensitive scan design (LSSD)

This method combines two design concepts, namely level sensitivity and scan path. The concept of a level sensitive design requires that the operation of circuit be independent of dynamic characteristics of the logic elements. This simplifies testing because it abstracts away from rise and fall times and propagation delays within gates. Furthermore in a level sensitive design the next state of the circuit is independent of the order in which changes occur when a state change involves several input signals.

The major element in a level sensitive design is the polarity hold shift register latch (SRL), which is used to implement all storage elements in the circuit. The SRL is similar to a master slave flip flop and is driven by two non-overlapping clocks. These clocks can be readily controlled from the primary inputs to the circuit.

The register also has the important characteristic of being configurable into a long shift register which forms a scan path. Nodes may be set to some predetermined value by shifting values into the SRLs and values of state elements may be examined by shifting out values in the SRLs. An SRL is shown symbolically in Figure 3.15.

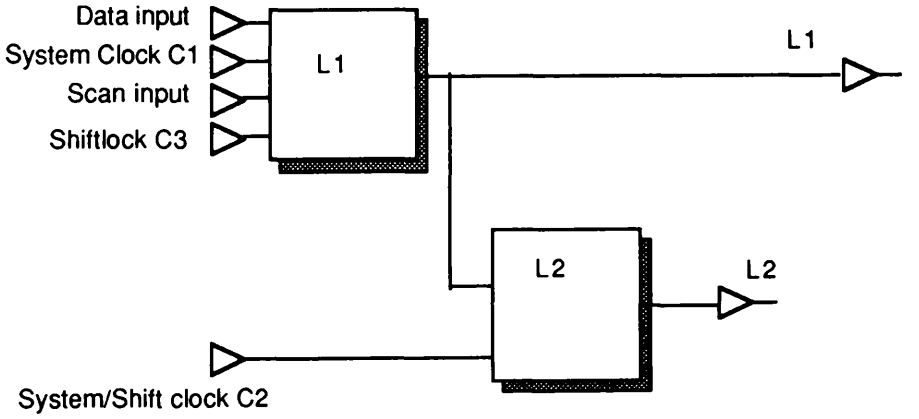


Figure 3.15: LSSD SLR Latch.

Testing using the LSSD approach proceeds as follows: first the individual registers are tested by using simple flush and shift tests. Then, the combinatorial subfunctions are tested. This involves switching the circuit into test mode. The SRLs are then preloaded

with a test pattern which is shifted in via the scan in port. This pattern is successively stepped through each element in the scan path by pulsing clocks C3 and C2.

The circuit is then switched into its normal operating mode and clock C1 is then pulsed on and then off. The result of the combinatorial subfunction is thus stored in the L1 latches of the SRL, and by pulsing C2 these values are duplicated in the L2 latches.

Finally, the circuit is switched back into test mode. The values in the L2 latches are shifted out by using the scan path. Thus by using the scan path, future states can be set up independently of the present state of the system. Internal states can be easily observed, so reducing the problem of testing a sequential circuit to that of testing a combinational circuit (as demonstrated by the LSSD configuration in Figure 3.16).

LSSD removes the necessity of performing detailed timing analysis on the circuit since it is is level sensitive. Automatic test pattern test generation is simplified since tests need only be generated for combinational circuits. Since LSSD is a disciplined design methodology a design can be checked for compliance to the design rules.

However, the designer is constrained to implement his system as a synchronous sequential circuit. Test times are increased since input and output data must be scanned serially and also the system must be switched between normal and test modes. Additional input/output pins are required for the scan-in/scan-out ports and clocks. Two clock pulses are required before data can pass from one partition to the other. This problem may be overcome by modifying the double latch. Despite these disadvantages, the LSSD scan path technique has been widely used in industry.

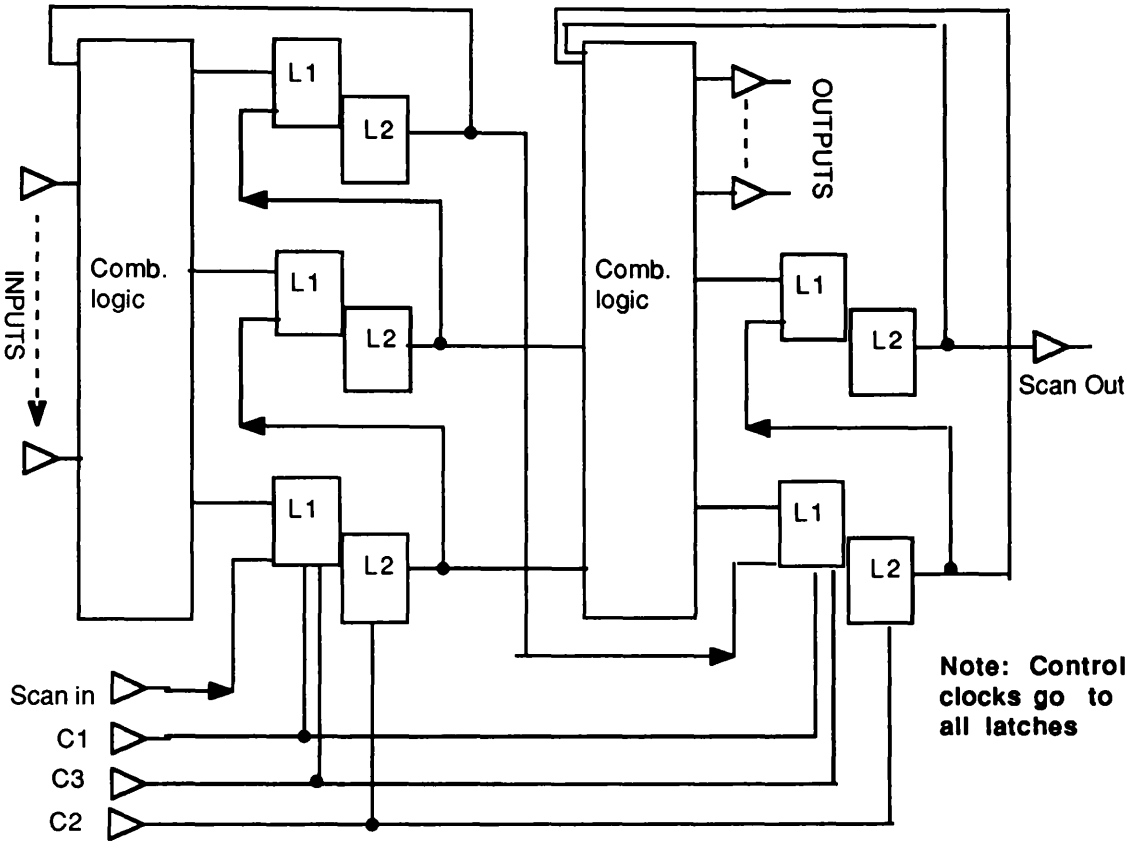


Figure 3.16: LSSD Configuration.

3.7.2.2 Scan-set logic

This technique entails selecting nodes of interest whose values can be recorded in a shift register. The same register can be used to alter these node values. Unlike the LSSD method, this method does not place shift registers in the main data path, as shown in Figure 3.17. Only a small number of nodes may be tested. These nodes may be set or examined by shifting values into or out of the shift register. The nodes to be examined are determined by using the results of a testability analysis program.

This method does not partition the circuit into combinational blocks. The scan/set register can be used to apply signals to blocking gates to partition the circuit into smaller modules to ease the testing problem.

The advantage of this method over LSSD is that the state of the system latches may be examined without interrupting the normal operation of the circuit.

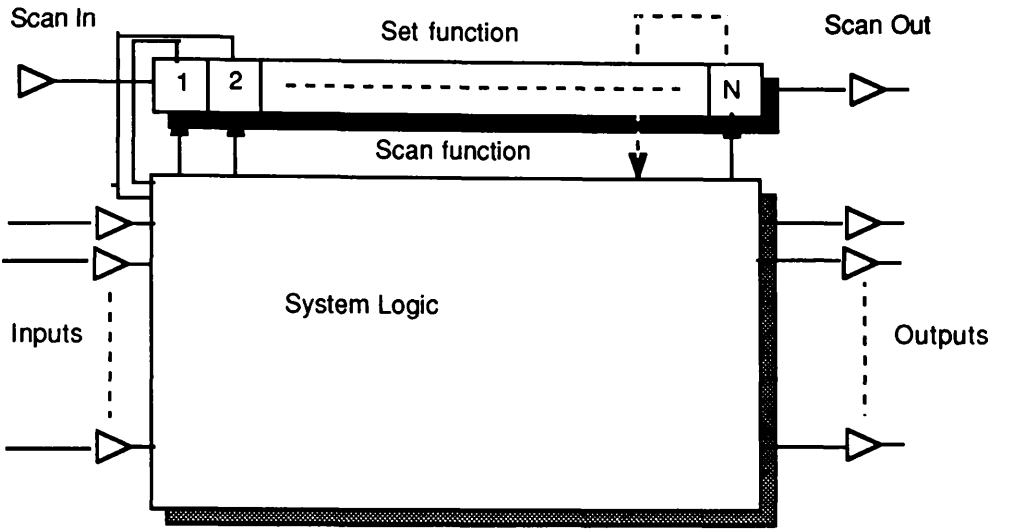


Figure 3.16: Scan/Set Configuration.

3.7.3 Built-in-test and self-test methods

Scan path methods simplify the task the testing, but vast amounts of test data must still be processed. Input test patterns have to be generated, true value output responses computed and stored, and output responses of the circuit under test stored and analysed.

Various techniques have been tried to tackle this problem by using data compression methods eg. transition counting and signature analysis. Transition counting is a relatively poor method, so we shall concentrate on signature analysis— a built-in-test method which is later incorporated into the self-test technique developed for VLSI circuits called BILBO.

3.7.3.1 Signature analysis

The main functional element used in signature analysis is the Linear Feedback Shift Register (LFSR) shown in Figure 3.18. This comprises of a series of latches in which signal taps are taken from certain stages, exculsive-ORed and returned to the input of the first latch. This configuration will generate a repetitive PN (pseudo-random noise) sequence.

In the signature analysis configureation stage the output of the exclusive-OR gate is not returned directly to the input of the first stage but is subsequently exclusive-ORed with a signal from some other source, as shown in Figure 3.18. At any time the contents of the register will not contain the values defined by the PN sequence, but will be modified in

some way characteristic of the signal coming from the other source. The modified bit pattern in the register is called the *signature* of the input source.

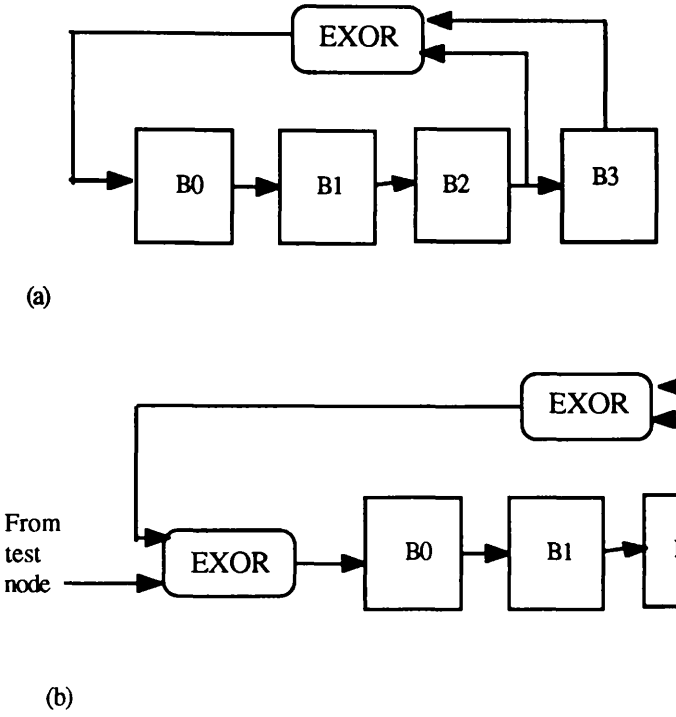


Figure 3.18: (a) PN Sequence generator (b) Signature analyser register

If the LFSR is initialised to give a pattern and then mixed with a signal coming from a node in a fault free circuit, after a prescribed number of clock cycles a signature characteristic of the fault-free circuit will be stored in the LFSR. Faulty circuits will have a different signature.

3.7.3.2 Built-in-logic block observation (BILBO)

BILBO is a built-in test generation scheme which uses signature analysis with a scan path. The major component in this self-test technique is a multi-mode shift register. This allows the BILBO to be set up in the following three ways:

1 as a long shift register forming a scan path
2 as a regular latch for normal operation
3 as a LFSR having multiple inputs for signature analysis

4 and under a certain control to be reset.

Figure 3.19 shows a BILBO configuration. A slightly different configuration is used in bus architectures.

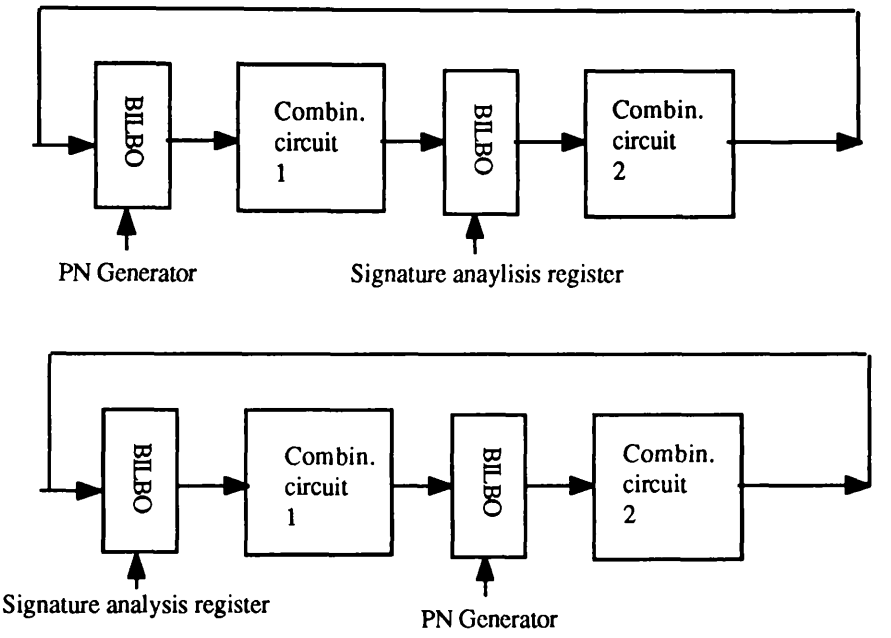


Figure 3.18: BILBO In test configuration.

A BILBO register is used to generate a PN-sequence which is applied to the combinational block under test. A second BILBO is used as a signature analyser register, which after N cycles will contain a signature peculiar to the state of the circuit. The BILBO containing the signature is then reconfigured as a scan-out register and the signature shifted out. The roles of the BILBOs are then reversed so that the next section of the circuit can be tested.

This technique effectively eliminates the need for test pattern generation, although fault simulation may be required to determine the fault coverage of the PN-sequences. The circuit will also have to be simulated to determine the fault free signature values.

3.8 Discussion

Testing for manufacturing errors in integrated circuits is an increasingly important task as

greater emphasis is placed on reliability and quality. However, testing circuits is also becoming increasingly difficult. This is due to the complexity of modern designs and the difficulty of examining the internal workings of chips. Most automated test methods are just formalisations of manual techniques. Reasonably good tools are available for generating tests for most combinational circuits, but sequential circuits are still very difficult to test. The usual approach is to decompose a sequential design into a set of combinational circuits which can then be tested using traditional techniques.

Automation of test pattern generation is essential if circuits are to be tested economically. This requires very precise descriptions of circuits for use by analysis tools. As designs become more complex, the need to describe systems hierarchically and at high levels of abstraction arises. Formal algebraic languages like Ruby have been shown to be suitable for such high level descriptions.

In addition to test pattern generators, many other tools are required to reduce the complexity of the problem. This chapter has shown the value of deductive fault simulators and testability measure programs. These analysis tools must be reliable—hopefully proved correct by formal verification techniques. Analysis tools also have to cooperate with each other in a harmonious fashion to create a usable design system. Many of the tasks performed by analysis tools are of a similar nature, so any re-use of code would be beneficial. Later chapters present a technique which allows a great deal of code re-use.

Chapter 4

Abstract Interpretation

4.1 Introduction

One method which has been used to analyse hardware descriptions and computer programs is abstract interpretation. This chapter introduces this technique and presents a common application in the field of strictness analysis of functional programs. Abstract interpretation is then shown to be useful for hardware descriptions too. A review of how others have used abstract interpretation for analysing hardware descriptions is also presented.

Given the task "find the sign of 34 * (-5) * (-3993)" one straightforward way to proceed is to evaluate the expression and then examine the sign of the result, ignoring the rest of the answer. Alternatively, we can use some simple rules about the signs of numbers. Since we know that when two numbers of the same sign are multiplied together, the result is positive and when two numbers of opposite sign are multiplied together, the result is negative, we can abstract way from the values of numbers. All we need to know about a number is its sign.

Let +ve denote "positive" and -ve denote "negative" and let them be of type sign. We can define $\times$, an abstract version of the multiplication operator * over +ve and -ve to describe what happens when numbers of various sign combinations are multiplied together:

```
-ve   ×   -ve   =   +ve
-ve   ×   +ve   =   -ve
+ve   ×   -ve   =   -ve
+ve   ×   +ve   =   +ve
```

If we can convert numbers to either -ve or +ve then we can use the above rules to compute the sign of the multiplication. We need an abstraction function abs, which removes from a number everything except the sign. The signature of this function is

$$abs : number \rightarrow sign$$

Now, the sign of 34 * (-5) * (-3993) may be computed as follows:

sign (34 * (-5) * (-3993))
= abs (34) × abs (-5) × abs (-3993)
= (+ve × -ve) × -ve
= -ve × -ve
= +ve

Performing the above calculation is much cheaper than working out the arithmetic and then throwing away most of the result. It is a shortcut to performing the full evaluation: it does less work and is simpler. The * operator has been replaced by an abstract operator × and numbers have been replaced by the abstract values +ve and -ve.

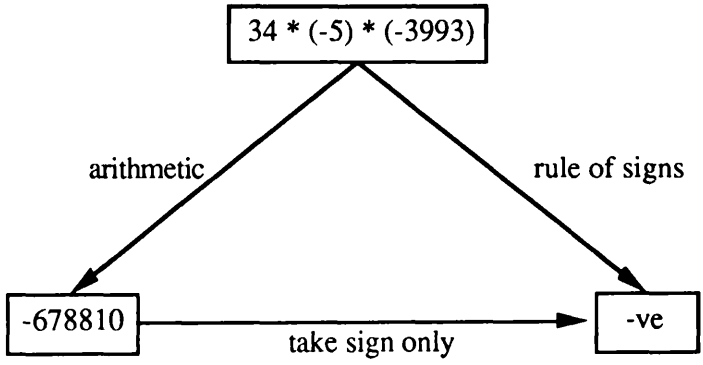


Figure 4.1: Using an abstraction over arithmetic and integers.

The **standard interpretation** above performs the arithmetic and then throws away everything except the sign. The **abstract interpretation** provides a shortcut which gives us the same result as applying the standard interpretation and then performing an abstraction (i.e. ignoring the sign).

Will the shortcut always give the right answer? We have to prove the abstract interpretation is correct with respect to the standard interpretation. In other words, we have to prove the **safety** of our abstract interpretation. For the example above, this could be done by showing that the rule of signs always delivers a result which is consistent with performing the arithmetic and then taking the sign.

4.2 Strictness Analysis

The technique of abstract interpretation has been used to compute strictness information for lazy functional programs. This information is used to compile more efficient code and to help spot parallelism.

In a lazy functional language, a function f will only evaluate its argument if it has to. If the argument is a complex composite object then a closure has to be created for this expression and passed to f. This is an expensive overhead. If we know that a particular argument will always be used, then it is cheaper to evaluate it first and then pass the resulting value. The function is said to be *strict* in the corresponding parameter. It is not always safe to evaluate the argument before passing it. For example, if an argument represents a non-terminating computation, but is not used in a call of a function, then it would be wrong to attempt to evaluate this argument before making the call.

We now formally define exactly what we mean by a strict function. A function f is **strict** in its argument if and only if

$$f \perp = \perp$$

where $\perp$ denotes bottom (or non-termination). This means that if f is given a non-terminating argument, then f will not terminate.

For a function of several arguments, we speak of strictness in a particular argument. Consider the function g of three arguments x, y and z. We say that g is *strict in* y if:

$$g\ x\ \perp\ z = \perp \qquad \textit{for any } x \text{ and } z$$

Consider the following definition of a first order function f.

```
f p q r =    if p=0 then
                 q+r
             else
                 q+p
```

Which parameters will f always need? The expression p=0 is always evaluated, so p is always evaluated, since = is a strict operation. Thus, f is strict in p. The function only uses r when p=0 so f is not strict in this parameter since it does not need always to evaluate it. However, q is always evaluated, no matter what the result of the test p=0 is, so f is strict in q.

The strictness information above was derived manually by inspection. It is possible to use a mechanical technique to analyse the strictness of a given function. This is done by executing an abstract version of f. Let the abstract values be 0 and 1 denoting non-termination and possible termination. To test for strictness in p, we compute the value of f 0 1 1. Informally, we are testing to see if f terminates when its first argument p does not terminate. If this value is 0 then f is strict in p, if it is 1 then we have no information about its strictness. The following abstract interpretation may be used to compute the desired strictness information.

abs ⟦constant⟧ = 1
abs ⟦variable⟧ = variable
abs ⟦a+b⟧ = a ∧ b
abs ⟦a=b⟧ = a ∧ b
abs ⟦**if** c **then** t **else** f⟧ = abs ⟦c⟧ ∧ (abs ⟦t⟧ ∨ abs ⟦f⟧)

Using these rules we can compute an abstract version of f, called f#.

f# p q r = (p ∧ 1) ∧ (q∧r ∨ q∧p)
= p ∧ q ∧ (r ∨ p)

This abstract version of f may now be executed with appropriate abstract values to yield strictness information about the parameters.

f# 0 1 1	=	0 ∧ 1 ∧ (1 ∨ 0) = 0	f strict in p
f# 1 0 1	=	1 ∧ 0 ∧ (1 ∨ 1) = 0	f strict in q
f# 1 1 0	=	1 ∧ 1 ∧ (0 ∨ 1) = 1	f is of unknown strictness in r

Why can we not conclude that f is not strict in r? The above interpretation only gives an approximate answer. Consider the following definition:

```
g x y =  if y=y then
             x+2
         else
             0
```

Informally, we see that g is strict in x because the true branch of the if statement is executed since the conditional part of the if expression is always true. This means that the value of x is always needed. But the abstract interpretation given above yields the following results:

g# x y	= y ∧ ((x ∧ 1) ∨ 1) = y	
g# 0 1	= 0	g strict in y
g# 1 0	= 1	g is of unknown strictness in x

We cannot hope to find all instances of strictness using an approximating technique

like abstract interpretation. However, it is important that the abstractions used are safe i.e. if an argument is analysed to be strict using the approximation, then it is also strict in the standard semantics.

Abstract interpretation has been very successful in analysing strictness and is the standard technique employed for this purpose [Peyton-Jones 87, Mycroft 83]. The technique can be improved by using a non-flat abstract domain to help reason about the strictness of composite data types [Hughes 86, Wadler 87]. It has been shown to deal adequately with higher order functions [Burn 86, Hudak 85] and also works for polymorphic languages [Abramsky 86]. The author has also proposed an alternative view of strictness analysis as a differencing operation akin to boolean differences [Singh 91].

4.3 Abstract Interpretation of HDLs

Strictness analysis is just one example from the programming language field that employs abstract interpretation to analyse programs. Other examples include life time analysis and compiler optimizations like register allocation. All of these examples work well with abstract interpretation because the underlying 'structure' of the interpretation is the same as that of the programs analysed.

Hardware descriptions can also be analysed by abstract interpretation. The analyses performed will be very different since strictness analysis and CPU register allocation are not relevant to hardware design. Instead, many useful measures like area, speed and power can be estimated quickly by using abstract interpretation. Others measures include longest and shortest delay and combinational nesting.

The use of a high level description language makes abstract interpretation a more formal process since the interpretation can be stated with respect to a precisely defined standard semantics. The interpretations we present are based on the standard semantics of Ruby, as defined by Sheeran [Sheeran 88]. Also, it is argued that performing abstract interpretation over high level descriptions will result in more accurate information. Most abstractions are approximations— more information about the purpose of a design is likely to lead to a more precise analysis. In a logic diagram, the purpose of individual gates may be very unclear. The use of a high level description language encourages modular hierarchical design where the purpose of subcomponents is stated clearly.

4.4 An Alternative Interpretation in Ruby

An alternative interpretation has been used by Sheeran to analyse Ruby circuit descriptions [Sheeran 86]. Left to right information flow is denoted by $\Rightarrow$ and right to left by $\Leftarrow$. The symbol $\Rightarrow$ is used to describe the case where the inputs are in the domain and the outputs are in the range. Similarly, $\Leftarrow$ describes the case where the inputs are in the range and the outputs are in the domain.

Each primitive is replaced by a relation describing the allowable directions of information flow. This relation is represented by a set of possible direction assignments to the domain and range. For gates like AND this will give a singleton set since there is only one allowable manner of information flow.

$$\text{AND}^* = \{((\Rightarrow,\Rightarrow),\Rightarrow)\}$$

To distinguish between the standard AND and the abstract version, the abstract version has been named AND*. Other primitives are annotated similarly.

Since NOT is its own inverse, it can always be driven from either direction giving a two element set.

$$\text{NOT}^* = \{(\Rightarrow,\Rightarrow), (\Leftarrow,\Leftarrow)\}$$

The abstract identity relation is defined to be the identity over $\Rightarrow$ and $\Leftarrow$ and tuples of these values.

$$\text{ID}^* = \mathcal{TU}\ \{(\Rightarrow,\Rightarrow), (\Leftarrow,\Leftarrow)\}$$

The operation $\mathcal{TU}$ is introduced to extend a relation over arbitrary tuples as well as atomic values. This operation can be defined schematically as:

$a\ \mathcal{TU}(R)\ b$	$\Leftrightarrow$ true	where $a\ R\ b$ and a, b atomic
$\langle a\rangle\ \mathcal{TU}(R)\ \langle b\rangle$	$\Leftrightarrow$ true	where $a\ \mathcal{TU}(R)\ b$
$\langle a, b\rangle\ \mathcal{TU}(R)\ \langle c, d\rangle$	$\Leftrightarrow$ true	where $a\ \mathcal{TU}(R)\ c\ \&\ b\ \mathcal{TU}(R)\ d$
etc.		

A wire or bus places no constraints on information flow. Latches have to be run forwards so they are replaced by $\Rightarrow$ or tuples containing only left to right arrows.

Abstract versions of the combining forms must also be given. The inverse of a circuit

should reverse all the information flows as well as flipping the circuit. Let the direction reversing relation be rev-dir:

$$\text{rev-dir} = \mathcal{TU}\ \{(\Rightarrow,\Leftarrow), (\Leftarrow,\Rightarrow)\}$$

Note that rev-dir is its own inverse i.e. rev-dir = rev-dir^{-1}. The abstract version of relational inverse should reverse the direction of flow, compose this with the inverse of the abstract circuit and reverse the direction of flow again..

$$(F^{-1})^* = \text{rev-dir}\ ;\ (F^*)^{-1}\ ;\ \text{rev-dir}$$

The definitions of serial and parallel composition remain unaltered. This interpretation can be used to check that information flows in only one direction.

4.5 Combinational & Sequential Depth

An estimate of how long information takes to propagate from the inputs of a circuit to the outputs is a useful piece of information. One technique for measuring this delay is to count the maximum number of delay elements between the inputs and outputs.

A suitable abstract interpretation for performing this task is defined as:

$a\ \mathcal{D}^*\ b$	$=_{def}$	$b = a + 1$
$a\ \text{NOT}^*\ b$	$=_{def}$	$b = a$
$\langle x, y\rangle\ \text{AND}^*\ z$	$=_{def}$	$z = max\ [x, y]$
$\langle x, y\rangle\ \text{OR}^*\ z$	$=_{def}$	$z = max\ [x, y]$

The semantics of the other language constructs remain unaltered: only the meaning of the basic components has to be changed.

As an example, consider the analysis of the following circuit:

$$\text{F} = \text{split}\ ;\ [\text{AND}\ ;\ \mathcal{D},\ \ [\iota, \mathcal{D}]\ ;\ \text{AND}\ ;\ \mathcal{D}]\ ;\ \text{OR}$$

A circuit diagram for this circuit is shown in figure 4.2.

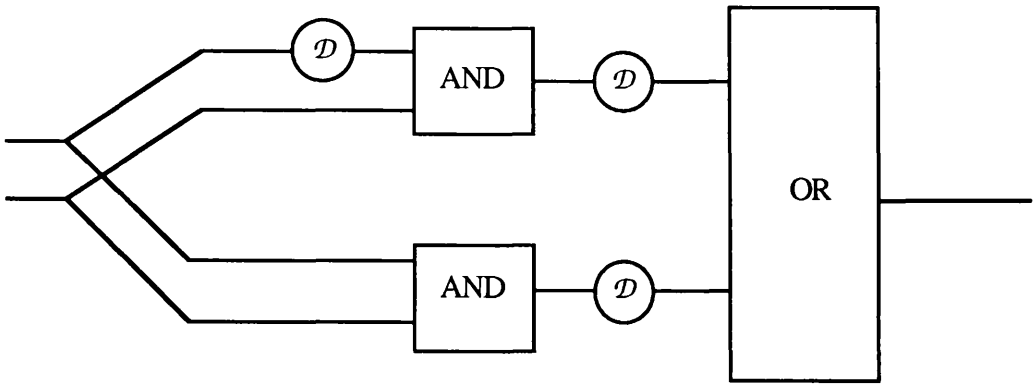


Figure 4.2: Example circuit F.

The maximum delay computation proceeds as follows. We start off with a delay of 0 on each primary input.

$$
\begin{aligned}
\langle 0, 0\rangle\ F^*\ b \quad &\Leftrightarrow \quad 0\ \mathrm{split}^*\ ;\ [\mathrm{AND}^*\ ;\ \mathcal{D}^*,\ [\iota,\ \mathcal{D}^*]\ ;\ \mathrm{AND}^*\ ;\ \mathcal{D}^*];\ \mathrm{OR}^*\ b \\
&\Leftrightarrow \quad \exists c.d.\ 0\ \mathrm{AND}^*\ ;\ \mathcal{D}^*\ c \wedge 0\ [\iota,\ \mathcal{D}^*]\ ;\ \mathrm{AND}^*\ ;\ \mathcal{D}^*\ d \wedge \\
&\qquad\qquad \langle c, d\rangle\ \mathrm{OR}^*\ b \\
&\Leftrightarrow \quad b = \mathit{max}\ [1, 2] \\
&\Leftrightarrow \quad b = 2
\end{aligned}
$$

This computation is shown in figure 4.3. Note that we rely on the laws:

$$
\begin{aligned}
(R\ ;\ S)^* &= R^*\ ;\ S^* \\
[R,\ S]^* &= [R^*,\ S^*]
\end{aligned}
$$

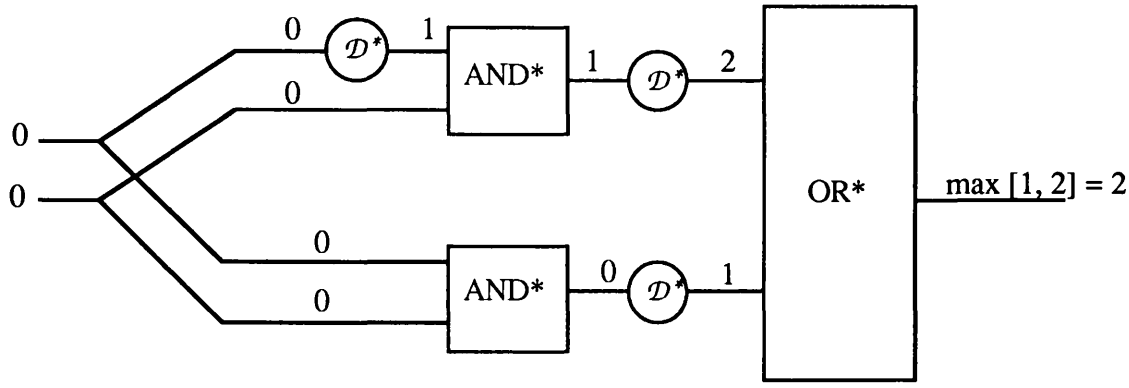


Figure 4.3: Sequential depth calculation of F.

A similar interpretation can be used to find the shortest sequential delay by using *min* instead of *max* in the definitions of AND* and OR*.

The combinational depth of a circuit can be computed by associating a weight for each combinational component. Consideration of the testability of the basic gates yields very rough weightings of 1 for inverters, 2 for OR gates and 3 for AND gates. This gives the

simple interpretation :

$$
\begin{array}{lll}
a\ \mathcal{D}^*\ b & =_{def} & b = a \\
a\ \mathrm{NOT}^*\ b & =_{def} & b = a + 1 \\
\langle x, y\rangle\ \mathrm{AND}^*\ z & =_{def} & z = max\ [x, y] + 3 \\
\langle x, y\rangle\ \mathrm{OR}^*\ z & =_{def} & z = max\ [x, y] + 2
\end{array}
$$

Using these rules with the above circuit gives:

$$
\begin{array}{lll}
0\ \mathrm{F}^*\ \mathrm{b} & \Leftrightarrow & 0\ \mathrm{split}^*\ ;\ [\mathrm{AND}^*\ ;\ \mathcal{D}^*,\ [\iota,\ \mathcal{D}^*]\ ;\ \mathrm{AND}^*\ ;\ \mathcal{D}^*];\ \mathrm{OR}^*\ \mathrm{b} \\
 & \Leftrightarrow & \exists \mathrm{c.d.}\ 0\ \mathrm{AND}^*\ ;\ \mathcal{D}^*\ \mathrm{c} \wedge 0\ [\iota,\ \mathcal{D}^*]\ ;\ \mathrm{AND}^*\ ;\ \mathcal{D}^*\ \mathrm{d} \wedge \\
 & & \qquad \langle \mathrm{c, d}\rangle\ \mathrm{OR}^*\ \mathrm{b} \\
 & \Leftrightarrow & \mathrm{b} = max\ [3, 3] + 2 \\
 & \Leftrightarrow & \mathrm{b} = 5
\end{array}
$$

Information about maximum combinational depth is useful for analysing the timing behaviour of a circuit.

4.6 Related Work

4.6.1 Simulating Circuits in Miranda

[Hill 86] has shown how Miranda can be used to simulate digital sequential circuits. He relies on Miranda's lazy evaluation to support a simple model of streams. However, his analysis was only applied to flattened gate-level descriptions with no support for hierarchy or geometry.

Hill represents gates by Miranda functions. Larger circuits are then built by using Miranda's ordinary functional composition. The gate functions operate over a three valued logic (true, false and unknown). This is represented by the data type `bit`:

```
bit ::= ON | OFF | UH
```

The function used to specify the behaviour of an OR gate is given as:

```
b_or x y    =   ON, x = ON \/ y = ON
            =   UH, x = UH \/ y = UH
            =   OFF, otherwise
```

Note that this is a function of two arguments. It is not possible to use this function in

expressions that employ Miranda's function composition notation. This is because function composition is defined only for functions for one argument.

Signals are represented simply as lists of these three valued bits. Clocks are distributed throughout the circuit explicitly. Constant signals are defined as infinite lists based on `bit`. Type synonyms are written using == and list concatenation (like LISP's CONS) is written using :

```
signal == [bit]
on = ON:on
off = OFF:off
uh = UH:uh
```

Clocks can also be defined as infinite lists of `bit` values. The definition provided for the OR gate above cannot be used in sequential circuits. A new function has to be defined by lifting the existing function to work over lists of bits.

```
bitwise op b1 b2   =    map2 b1 b2
                        where
                        map2 (a:x) (b:y) = (a $op b):(map2 x y)
or_gate = bitwise b_or
```

Again, Hill uses too poor a representation for signals, since he has had to define a special function `map2`. In our system, this is not necessary, because we use a tuple of streams over which a normal map can be applied.

Using the above definitions, Hill has built some modest circuit descriptions of synchronous circuits. However, this approach is hampered by the fact that circuit descriptions are simply Miranda functions. Miranda's limitations e.g. lack of a parallel combining form, are reflected in Hill's descriptions. This prevents structural information from being captured elegantly— Hill's definitions only give *connectivity*.

Hill has suggested that by using a different semantics his system could be used to produce circuit layouts and test vectors. This is quite easy to accomplish in his system since for most alternative interpretations, only the definitions of the base components need to be altered. Hill also suggests the use of a more detailed representation for `bit` e.g. voltage levels.

The author has implemented a much improved version of Hill's system which uses a richer data type to capture structured logic types. In this system, every function is unary, and Miranda's built in function composition can be used to give elegant and readable circuit descriptions. There is no need to define a special function for serial composition.

4.6.2 System Semantics

Boute has extended the denotational semantics of programming languages to a semantics suitable for describing arbitrary systems [Boute 88]. A system is a collection of physical objects (subsystems) which interact with each other through physically identifiable interfaces. Thus, systems can comprise of objects which are not computations in any sense. However, useful computations can still be performed over these objects. One major advantage of Boute's system is that different meaning functions can be used with the same formal description to calculate different system properties such as component cost and performance characteristics.

Boute presents a *systems semantics* for describing system properties by means of semantic functions. This is different from denotational semantics which defines exactly one interpretation using an abstract mathematical domain [Stoy 77]. System semantics defines various interpretations corresponding to different characteristics of a physical system. The generalization of denotational semantics by the use of abstract domains (abstract interpretation) is mirrored in systems semantics by an extension in the opposite direction (adding information rather than removing it) by using concrete interpretations. This is done by injecting extra information into domains rather than abstracting information.

Boute defines semantics using a model which consists of a meaning function m which maps elements of a set S into elements of a domain of interpretation D. The domain D is the domain of possible meanings ($m \in S \rightarrow D$). A model M is a pair $M = (D, m)$. Boute uses total functions to define meanings, so corresponding models are completely defined.

Boute employs a hardware description language called Functional Description of Systems (FUNDS). We shall not present the entire syntax and semantics of this language: the examples should be sufficient to demonstrate the principles under consideration. The syntax of FUNDS is left flexible and resembles usual functional language syntax (like SASL [Turner 79]). The combinational subset contains the following constant entities:

$$\text{constant} \quad = \quad \textit{zero} \mid \textit{one} \mid \textit{not} \mid \textit{and} \mid \textit{or}$$

where *not* is of 1 place *and* and *or* are multi-place (i.e. any number of arguments).

To avoid the proliferation of semantic definitions, common parts are factored out. At the semantic level, a generic definition is introduced for models to which others models are said to conform. Models which do not have factorizable parts are said to be *singular*. At the syntactic level when semantic functions are defined over an abstract syntax, the

concrete syntax is defined in terms of the abstract syntax without reference to the meaning functions.

The following generic model is defined for the meaning of functions (*mfun*) and expressions (*mexp*). Here, D is a zeroth-order domain of interpretation. The interpretation for the constants is of the general form $k \in C \to D_C$, where $D_C = D^* \to D$.

$mexp \in E \to I \to D$

$mexp\ v\ i = i\ v$ variables

$mfun \in F \to I \to D^* \to D$

$mfun\ c\ i = k\ c$ for constants

$mfun\ [\![\lambda(v_0,\ldots,v_{n-1}).e]\!]\ i \in D^n \to D$

$mfun\ [\![\lambda(v_0,\ldots,v_{n-1}).e]\!]\ i\ (d_0,\ldots,d_{n-1}) = mexp\ e\ i\ [d_0/v_0] \ldots [d_{n-1}/v_{n-1}]$

$m \in S \to D^* \to D$

$m\ s = m\ \text{fun}\ s\ i$ i can be any interpretation

Using this generic model, a simplex behavioural model $Smplx = (\{0, 1\}, smplx)$ for the combinational part of FUNDS can be constructed. The constants have the following definitions:

$k\ zero \in D^0 \to D$ with $k\ zero = 0$

$k\ one \in D^0 \to D$ with $k\ one = 1$

$k\ not \in D \to D$ with $k\ not\ d = \neg d$

$k\ and \in D^* \to D$ with $k\ and\ (d_0,\ldots,d_{n-1}) = d_0 \wedge \ldots \wedge d_{n-1}$

$k\ or \in D^* \to D$ with $k\ and\ (d_0,\ldots,d_{n-1}) = d_0 \vee \ldots \vee d_{n-1}$

Since the model *Smplx* conforms to the generic model, *mexpr* and *mfun* do not need to be defined again. The meaning function for sentences is then *smplx*.

Boute also defines several others models including a structural model for describing loop-free single-output combinational circuits built from elementary gates. He chooses a model that makes clear the distinction between fanout and replication by using an appropriate naming convention. A multiplex behavioural model is also presented for sequential circuits.

To contrast Boute's method with what we have presented, we give the essential part of a worst-case timing model with $D = \mathbb{R}_{\geq 0}$. For an n-place constant c, $k\ c$ in $D^n \to D$ with:

$$k\ c\ (d_0,\ldots,d_{n-1}) = \max\ [d_0,\ldots,d_{n-1}] + delay\ c$$

where *delay* in $C \to D$ is an auxiliary function specifying the delay for each constant. This is very similar to the maximum combinational depth calculation presented in section 4.5.

The main differences are in syntax.

4.6.3 Other Work

O'Donnell has used alternative interpretations to produce drawings of tree-network circuits [O'Donnell 88]. He presents a language called Hydra which offers the designer several specification styles, including the ability to capture geometric information in the same way as Ruby does. Hydra also allows path depth and netlist analyses to be performed. Using combining forms similar to those found in TeX for drawing pictures, he has drawn complex circuits by exploiting the technique of functional geometry [Henderson 82]. Similar analyses have been used to extract layout information from FP [Schlag 84].

Meshkinpour presents a functional hardware description language called FHDL. Using this notation, he has reorganized a given system in a pipelined fashion in order to improve its throughput. To help partition digital systems, a symbolic interpreter is adapted to compute timing information. This is done in an ad hoc manner by associating attributes with various gates. This is similar to the abstract interpretation we have presented for timing analysis.

4.7 Discussion

Abstract interpretation can be used to analyse hardware descriptions, giving information which is related to the circuit's behaviour. The advantage of this technique is that once a simulator is available for a language it only takes a small amount of extra effort to produce other analysis tools. This is because we usually only have to redefine the meaning of the processing nodes like AND and NOT. The definitions for wiring circuits tend to be the same in many analyses, so the standard definition can be re-used.

Analysing circuits for testability involves finding information which is not so directly related to the behaviour of the circuit. To do this, the behaviour of the basic components has to be altered in a less disciplined manner which destroys the safety principle. However, there is no way round this, since the standard semantics does not contain enough information to abstract from and to yield the analyses we are interested in.

Boute has used a concrete domain to obtain more detailed information about circuits. However, to perform testability analyses we will need completely different domains. These cannot be made by simply injecting extra elements into the standard domain. We

need an interpretation which is even more general than that offered by concrete domains. For example, testability measure uses a domain of vectors that are unrelated to the logic values in the standard domain.

To describe such analyses requires the use of a non-standard semantics over a non-standard domain of values. This is the topic of the next chapter, which deals with such non-standard interpretations in detail.

Chapter 5

Non-Standard Interpretation

5.1 Introduction

Abstract interpretation does not possess sufficient power to capture all circuit analyses of interest. In this chapter, we use the non-traditional discipline of *non-standard interpretation.* We allow the standard semantics to be replaced by any other semantic definition. Normally, there will be no formal connection between the standard and non-standard semantics. Similarly, there need not be an abstraction between standard and non-standard values.

Our approach is similar to that of Boute [Boute 88] outlined in chapter 4. Boute uses a generic model to capture common aspects of circuit analyses. We provide a more powerful generic mechanism. The technique we adopt operates over a richer language than that used by Boute because we can deal (in a limited fashion) with inverse. Boute argues that the choice of good composing forms is an essential part of his system semantics technique, especially for alternative interpretations. We provide a richer collection of combining forms and we also allow all of these combining forms to be overridden. Boute does not permit the single combining form that his system supports (i.e. functional composition) to be redefined. Thus, in one sense at least, our work can be viewed as an extension of Boute's to relational style descriptions with more powerful combining forms and alternative interpretations.

Our aim is to make non-standard interpreters that we can slot into a circuit analysis tool, rather like one can slot expansion cards into the backplane of computers to increase their power and functionality. We have developed such a backplane for analysing Ruby circuit descriptions. It should be easy for the user to specify and add new interpretations. However, some user interface code might have to be written for interpretations which require their results to be output in a special manner e.g. bar graphs.

Non-standard interpretation has been used by Luk [Luk 90] for analysing parameterised designs. Luk uses various metrics which are employed to characterise the performance trade-offs for generic designs. Akella and Gopalakrishnan [Akella 90] have performed test pattern analysis at a higher level of abstraction by associating testing directly with the specification of the design. Faults are injected into a structural specification, and the behavioural consequences are inferred by process composition.

Various techniques have been used by the author to implement non-standard interpretation, with differing degrees of success. Two promising techniques for non-standard interpretation are presented. We show some initial attempts as motivation for the final approaches. One of the final approaches is used when we want to observe the values at the primary inputs and outputs of circuits. The other technique returns a graph of the circuit under analysis with all the internal nets annotated with their non-standard values.

Throughout this chapter, a *node* is a processing element like an AND or NOT gate. It is not a wire. Wires are grouped into *nets*. Every wire on a net has the same value.

5.2 Techniques for Expressing NSI

There are many ways to represent a non-standard interpretation of Ruby hardware descriptions. We have talked about 'changing the semantics' of the elementary gates in a rather informal manner. Before a system can be built for performing non-standard interpretation, we have to be much more precise about what we mean by 'interpretation'. This is done by discussing various models for interpretations that have been implemented.

To aid the description of various interpretations, we assume that we have available an algebraic object that represents the abstract syntax of Ruby expressions. The semantics of Ruby is then described by giving the semantic denotations for the abstract syntax. The specification of this rather large object would look like the following algebraic type declaration. The vertical bars separate *constructors* which correspond the elements of the Ruby language. Thus, the abstract syntax is represented in terms of *constructors*.

RUBY := *And'* | *Or'* | *Not'* | *App'* | *Ser'* [*RUBY*] | *Par'* [*RUBY*] |
Block' string [*RUBY*] | *Id'* | *Inv' RUBY* | *Fork' num...*

A prime is written after the name of each constructor to avoid confusion with the corresponding syntactic entity. We can define the standard semantics for Ruby by giving

a semantic function for each constructor (abstract syntax) in *RUBY*. We shall use the term 'Ruby construct' to mean one of the constructors in *RUBY*. Note that this specification is recursive, and represents Ruby circuit descriptions as a tree. We shall talk about graphs to allow for the possibility of feedback loops in sequential circuits, or the sharing of nodes. The leaf nodes are constructors of arity 0. For example, the processing nodes *And'*, *Or'* and *Not'* and basic wiring forms like *App'* do not operate over circuit descriptions. The internal nodes correspond to the 'higher-order' constructors like serial (*Ser'*) and parallel (*Par'*) composition which themselves take other Ruby constructions and combine them to make a new construction.

A Ruby expression can refer to the name of another Ruby definition available through the current environment. The constructor *Block'* describes such a reference. The first argument is the name of the definition which is being referenced and the second argument is a list of Ruby expressions which are higher order arguments (parameters). This mechanism is implemented just like function calls in traditional functional languages. The reference is replaced by the defining body of the referenced definition, with the appropriate parameter and argument substitutions (call-by-value). For example, if fst R is defined in the environment to be *Par'* [R, ι] then the expression *Block'* "fst" [*And'*] is expanded as follows:

Block' "fst" [*And'*] → *Par'* [*And'*, ι]

Note that the formal higher order formal parameter R matches with the actual parameter (argument) *And'*. The transformation above assumes the existance of a global constant environment.

The semantic functions, including non-standard interpretations, can be considered to be mappings between the abstract syntax of Ruby (call this *A*) and an abstract domain of interpretation for expressing the semantics (call this *D*). This is demonstrated in figure 5.1 for a mapping *E*. The mapping also needs to take account of environment information which is omitted from the diagram.

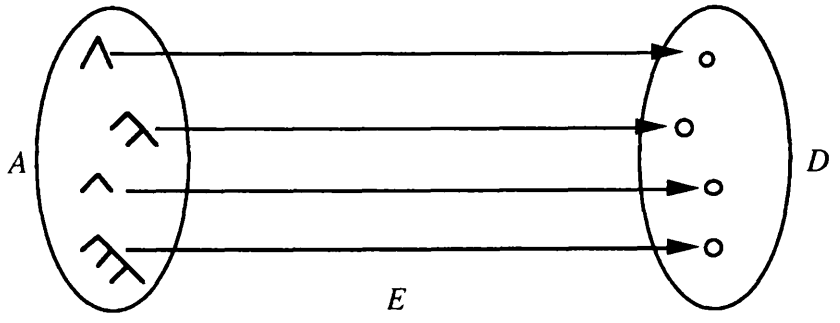


Figure 5.1: A semantic mapping between *A* and *D*.

An important restriction is placed on how non-standard interpretations are to be constructed. The most general approach is to replace the relation which specifies the behaviour of each element of Ruby by a non-standard relation. The non-standard semantics could then be implemented by mapping it onto a relational language. Although this is the most straightforward way to proceed, we choose a different technique. We want our interpretations to be realised efficiently, to capture the information flow precisely and to be easy to implement. Relational implementations make data dependencies implicit and usually result in backtracking implementations that are not very efficient. After considering a large number of circuit analysis algorithms, we have come to the conclusion that many complex algorithms can be decomposed into a series of unidirectional analyses. Each unidirectional stage can be implemented by *functions* rather than relations. This leads to a much more efficient implementation, while retaining ease of coding.

We choose a very simple scheme for trying to capture relational analyses by composing unidirectional analyses. Only two kinds of unidirectional analyses are used: *forwards* analysis and *backwards* analysis. In forwards analysis, information flows only from the domain to the range. In backwards analysis, information flows from the range to the domain. Complex relational analyses are described by using combinators that operate over unidirectional analyses. For example, one useful combinator applies a forwards analysis and then overlays the result of this onto a backwards analysis. We cannot capture all relational analyses using this scheme, but we have been able to express many complex backtracking circuit analyses using this technique.

One of the most obvious ways to make a non-standard interpretation is to completely respecify the semantics of Ruby, as shown in figure 5.2 for a non-standard interpretation *E'*. Note that the semantic domain will in general be different for each interpretation. This is how the first non-standard interpretation system was built by the author. However, providing an alternative semantics for all of Ruby is a rather unsatisfactory approach. One of the most appealing aspects of non-standard interpretation is the ability to change the meaning of only a small subset of the language (e.g. the three logic gates) to get a completely new interpretation. The other elements of the language should have the same semantics as before, but should operate over non-standard values.

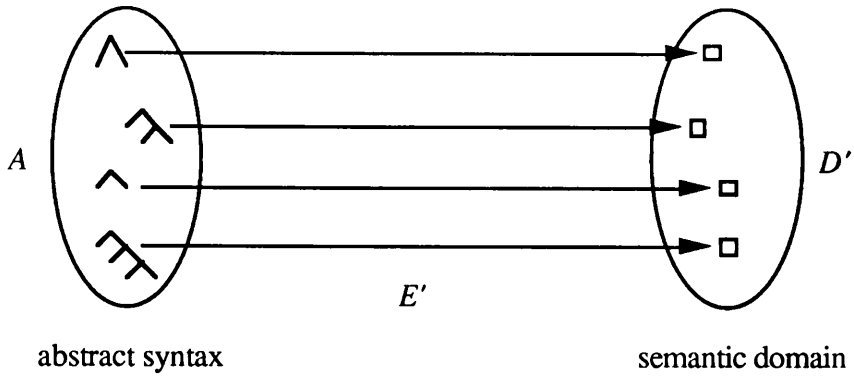


Figure 5.2: Completely respecified non-standard semantics

The difference between figure 5.1 and figure 5.2 is that the range is different (non-standard semantic domain) and the semantic function is different (a non-standard semantics). However, the domain is the same in both schemes (since we want to analyse the same description under many interpretations).

In most interpretations, the meanings of the wiring primitives and combining forms remain unaltered. Wires carry information without examining it. They may lose information by not connecting (or *relating*) a wire in the domain to the range. Wires can also duplicate information as well as re-arrange the order of information in a tuple. However, the information content does not affect the behaviour of wiring circuits. Such circuits behave then rather like polymorphic functions.

In most interpretations, the wiring primitives and combining forms will just be plumbing that transmits the values of interest that are computed at combinational gate nodes. A large area of most circuit designs is spent on communication rather than processing. Ruby provides a rich collection of operations for describing and laying out various wiring forms. It would be tedious to have to respecify them for each new analysis.

This leads to an alternative technique for making non-standard interpretations. We can parameterise the standard semantics on the 'processing' nodes i.e. *And'*, *Or'* and *Not'*. This is done by parameterising the semantics on the language constructs which require different interpretations. At first, the basic gates i.e. the processing nodes were selected for parameterisation. This method corresponds directly to the generic instantiation mechanism used in the Ada language for generic packages.

This approach effectively divides the syntactic domain *A* shown in figure 5.1 into two parts. One part is invariant between different interpretations and is used to describe those syntactic entities which have fixed interpretations. The other part contains the syntactic entities that change meaning under different interpretations. This is demonstrated in figure

5.3 for a non-standard interpretation *E''*.

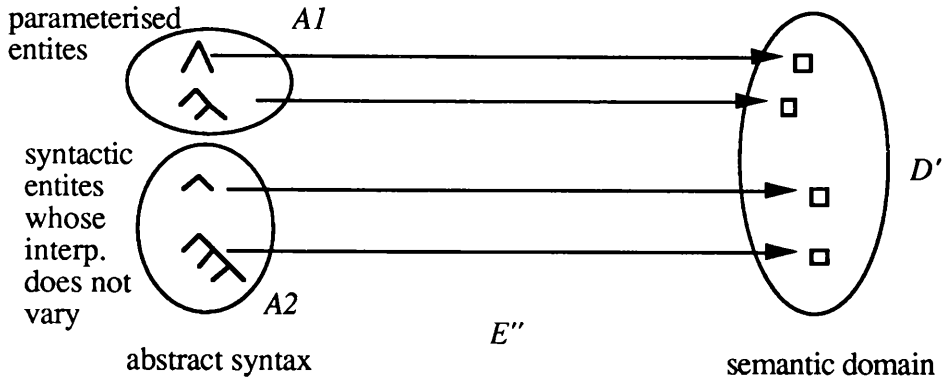


Figure 5.3: Parameterised interpretations.

Note that the syntactic domain *A* has been split into two domains *A*1 and *A*2. It should be the case that $A1 \cup A2 = A1$ and $A1 \cap A2 = \varnothing$. This ensures that there is exactly one mapping for every well formed syntactic entity.

This method works well for analyses that only need to provide alternative semantics for the basic gates. This covers a large class of interpretations. For example, symbolic simulation and deductive fault simulation can both be represented by this model. This technique can be implemented in Miranda simply as a function which can be partially applied. However, it is inflexible because it is difficult to cope with changing the status of a syntactic entity from non-parameterised to parameterised. For example, we might also want our interpretation to be parameterised on *Fork'*. This involves changing *A1* and *A2* and re-coding our implementation function, although the change is very minor.

This was indeed done, and then several other language features were also added to the list of paramerterised language constructs. The situation degenerated to the point that the standard interpretation was a hollow shell providing no default semantics for any language feature because everything was paramerterised. This takes us back to where we started i.e. having to re-specify the semantics of the entire language for each interpretation. Clearly, another method was required that allowed certain Ruby constructs to have their semantics *redefined* while leaving the others alone.

A variant of the above technique involves making the standard interpretation the 'behaviour' or 'simulation' interpretation where every Ruby construct including the processing nodes had a default (simulation) semantics. A mechanism is then provided for over-riding the semantics of *any* Ruby construct.

A semantic definition is now provided for the forwards and backwards standard semantics. The semantic definition *S* is called an *interpretation* and takes as its parameters the direction of the analysis, a Ruby expression to evaluate and domain or range values to

be used during the evaluation. The environment ρ is always constant during evaluation, so it is not an explicit parameter. The direction is denoted by *f* for forwards and *b* for backwards.

$$direction := f \mid b$$

The type of an interpretation can be given as follows where *V* 1 and *V*2 denote the range and domain of interpretation:

$$interpretation: \quad direction \rightarrow \text{RUBY} \rightarrow V\,1 \rightarrow V2$$

We shall use partial application to simplify our semantic definitions. Let *f*, *g* and *h* be functions and *x* be a parameter. Under partial application the following equivalences hold. They extend in a natural manner to other combining forms.

The standard semantics for the serial and parallel combining forms are defined as follows where the third parameter is omitted by partial application. Pattern matching is used and the definitions are scanned in a top down manner.

$$S\,f\,[\![Ser'\ [P, Q]]\!] = S\,f\,[\![P]\!]\ ;\ S\,f\,[\![Q]\!] \tag{5.1}$$

$$S\,b\,[\![Ser'\ [P, Q]]\!] = S\,b\,[\![Q]\!]\ ;\ S\,b\,[\![P]\!] \tag{5.2}$$

$$S\,dir\,[Ser'\ [x]]\!] = S\,dir\,[\![x]\!] \quad \text{(singleton serial composition list)}$$

$$S\,dir\,[\![Par'\ [P, Q]]\!] = [S\,dir\,[\![P]\!], S\,dir\,[\![Q]\!]] \tag{5.3}$$

In the above definitions, semicolon (;) refers to the usual forward function composition: $(f\,;\,g)\,x = g\,(f\,x)$. A definition for parallel composition specialised to functions is also required:

$$[F, G]\,\langle a, b\rangle = \langle F\,a, G\,b\rangle \tag{5.4}$$

Inverse is defined by:

$$S\,f\,[\![Inv'\ B]\!] = S\,b\,[\![B]\!] \tag{5.5}$$

$$S\,b\,[\![Inv'\ B]\!] = S\,f\,[\![B]\!] \tag{5.6}$$

A named Ruby definition is elaborated by looking up the name in the environment and then performing a textual substitution of parameters by arguments using the function *subst*. The environment function has type ρ :: *string* → *RUBY*

$$S\,dir\,[\![Block'\ name\ args]\!] = S\,dir\,[\![subst\ (\rho\ name)\ args]\!] \tag{5.7}$$

The behaviour of the basic gates are described by set-valued functions, using one function for each direction. For example, the forward behaviour of AND is given by Andf and the backward behaviour by Andb defined as:

$$\text{Andf } \{(L, L)\} = \{L\} \tag{5.9}$$
$$\text{Andf } \{(L, H)\} = \{L\} \tag{5.10}$$
$$\text{Andf } \{(H, L)\} = \{L\} \tag{5.11}$$
$$\text{Andf } \{(H, H)\} = \{H\} \tag{5.12}$$
$$\text{Andb } \{L\} = \{(L, L), (L, H), (H, L)\} \tag{5.13}$$
$$\text{Andb } \{H\} = \{(H, H)\} \tag{5.14}$$

The corresponding functions for OR (Orf, Orb) and NOT (Notf, Notb) are defined similarly. This gives the following standard semantics for the basic gates:

$$\mathcal{S}f[\![And']\!] = \text{Andf} \tag{5.15}$$
$$\mathcal{S}b[\![And']\!] = \text{Andb} \tag{5.16}$$
$$\mathcal{S}f[\![Or']\!] = \text{Orf} \tag{5.17}$$
$$\mathcal{S}b[\![Or']\!] = \text{Orb} \tag{5.18}$$
$$\mathcal{S}f[\![Not']\!] = \text{Notf} \tag{5.19}$$
$$\mathcal{S}b[\![Not']\!] = \text{Notb} \tag{5.20}$$

Higher order combining forms like **map** are instantiated into thier fixed size equivalents at run time. Thus, the analysis of a circuit containing a **map** degenerates into the analysis of a fixed size parallel composition. The standard definitions of the higher order combining forms are used to simply unfold them from descriptions.

A non-standard interpretation is made by overriding some or all of the standard interpretation by another semantic definition over the same language (or abstract syntax). The usual definition for over-riding is used, employing the infix operator $\oplus$:

$$\begin{aligned}(P \oplus Q)\, a &= P\, a, \text{ if } a \in \text{dom } P \\ &= Q\, a, \text{ otherwise}\end{aligned} \tag{5.30}$$

If I is a new interpretation for some of Ruby, then a non-standard interpretation is given by:

$$I \oplus S$$

From the definition of $\oplus$ it is clear that the following identity holds:

$$S = S \oplus S$$

We now review our decision to use the above mechanism for non-standard

interpretation and compare it with the most general non-standard interpretation scheme. For a full relational implementation, a set could be used to represent the required relation. However, since we are only interested in running our circuits either forwards or backwards, it seems natural to represent the non-standard semantics by two functions. Another reason for separating the forward and backwards semantics is that many analyses only make sense in one direction. For these, the semantic function for the other direction can be left undefined. Of course, separating a relation into two functions allows for much more efficient implementation. This allows the relation to be implemented by a pair of functions without explicit backtracking.

5.3 An Example: Symbolic Simulation

A symbolic simulator is constructed using the interpretation scheme presented above. The non-standard values are now symbolic expressions which represent the value at a given node in terms of input variables. Let the non-standard value be called *symbol* and define it as:

symbol := *variable* | *AndExpr symbol symbol* | *OrExpr symbol symbol* |
NotExpr symbol

The input to the simulator is the name of primary inputs. We define symbolic simulation only for forward interpretations: the backward case is left undefined.

An exclusive-or gate can be defined in Ruby by:

```
exor = split ; [[ι, not], [not, ι]] ; [and, and] ; or
```

The abstract syntax tree in terms of *RUBY* for this definition is shown in figure 5.4.

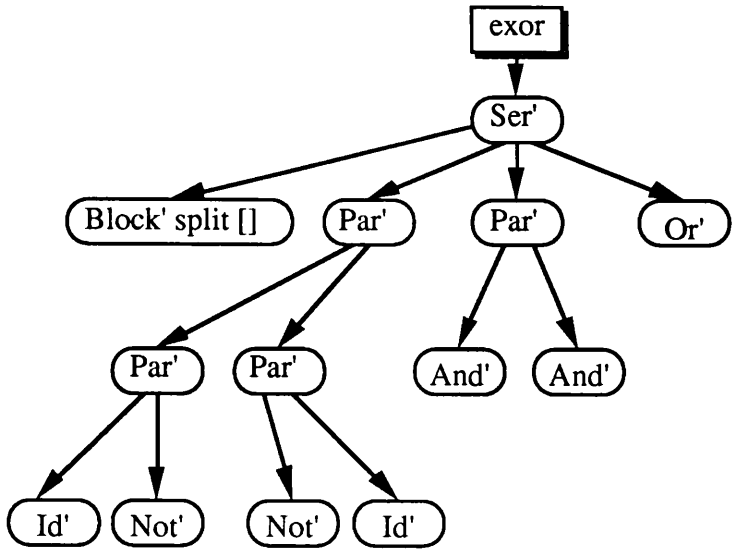


Figure 5.4: Abstract syntax for exclusive-or definition.

It is instructive to note that non-standard interpretations that use alternative semantics for just the processing constructs (*And'*, *Or'*, *Not'*) change the interpretation of some leaf nodes, as is the case in figure 5.4. Other leaf nodes like *Id'* remain unaltered.

A symbolic simulation of the definition with the non-standard value $\langle x, y \rangle$ (x and y are *symbolic* variables) should give the following result:

$$OrExpr\ (AndExpr\ (NotExpr\ y)\ x)\ (AndExpr\ (NotExpr\ x)\ y)$$

The expression above is a LISP-like representation of the boolean expression $x.\neg y \mid \neg x.y$ which realises the exclusive-or operation.To implement symbolic simulation as a non-standard interpretation, only the processing nodes need to be redefined. A suitable interpretation is *Sym*, defined as:

$$\begin{aligned} &and\_sym\ \langle x, y \rangle = AndExpr\ x\ y \\ &or\_sym\ \langle x, y \rangle = OrExpr\ x\ y \\ &not\_sym\ x = NotExpr\ x \end{aligned}$$

$$\begin{aligned} Sym\ f\ [\![AND]\!] &= and\_sym \\ Sym\ f\ [\![OR]\!] &= or\_sym \\ Sym\ f\ [\![NOT]\!] &= not\_sym \end{aligned}$$

Note that the backward interpretation is left undefined since symbolic simulation is usually only carried out in the forward direction. However, it is interesting to consider what a backward symbolic simulator should do. The forwards case finds an answer to the question "If I give the following inputs, what expression appears at the output?". The answer to this question is easily derived from the structure of the circuit. The backwards case can be thought of as asking two slightly different questions. In one case, constant

values are assigned to outputs of internal nodes and the question is "What inputs produce these outputs?". Answering this question essentially involves performing the task of a test pattern generation program. In the other case, we associate expressions (in terms of input variables) to the primary outputs. This asks the question "what input assignments produce these values at the output, if any?". This performs the task of a program that checks to see if the realisation of the circuit meets its specification. There is a strange link between backwards symbolic simulation and test pattern generation and circuit validation. Roth [Roth 80] has shown that his test pattern generation technique (the D-algorithm) could be adapted to validate certain kinds of circuits. This exploits the behavioural information present in test patterns.

A symbolic simulator interpreter in the forward direction called *SS* can now be built:

$$SS = (Sym \oplus S) f$$

This symbolic simulator is now used to simulate the exclusive-or circuit:

SS ⟦exor⟧ ⟨*x*, *y*⟩
≡ {Use standard definition of function-call elaboration and serial composition.}
SS ⟦split ; [[ι, not], [not, ι]] ; [and, and] ; or⟧ ⟨*x*, *y*⟩
≡ {Use standard definition for elaborating wiring circuit split and serial composition.}
SS ⟦[[ι, not], [not, ι]] ; [and, and] ; or⟧ ⟨⟨*x*, *y*⟩, ⟨*x*, *y*⟩⟩
≡ {Standard interpretation used for identity and parallel composition. Over-riding interpretation *Sym* used for not.}
SS ⟦[and, and] ; or⟧ ⟨⟨*x*, *NotSym* y⟩, ⟨*NotSym x*, *y*⟩⟩
≡ {Standard interpretation used for parallel composition. Over-riding interpretation *Sym* used for and.}
SS ⟦or⟧ ⟨*AndSym x* (*NotSym y*), *AndSym* (*NotSym x*) *y*⟩⟩
≡ {Over-riding interpretation *Sym* used for or.}
OrSym (*AndSym x* (*NotSym y*)) (*AndSym* (*NotSym x*) *y*))

which is the expected result. Note that we have used syntactic entities inside the ⟦...⟧ meta brackets for clarity. We should have written *And'* instead of and.

A symbolic simulator is a very obvious candidate for implementation as a non-standard interpretation. It is clear from the outset that we only have to re-define the processing nodes: the semantics of everything else stays the same. The example works well: the definition given above is natural looking. For a modest outlay, we have reused a large amount of the standard interpretation to build a completely new tool.

In the standard interpretation the values that flow along the wires are standard boolean values. The nodes are represented by functions that manipulate this boolean data. Figure

5.5 show a graph of the exor gate under simulation with the input ⟨L,H⟩. The standard values on each arc are shown.

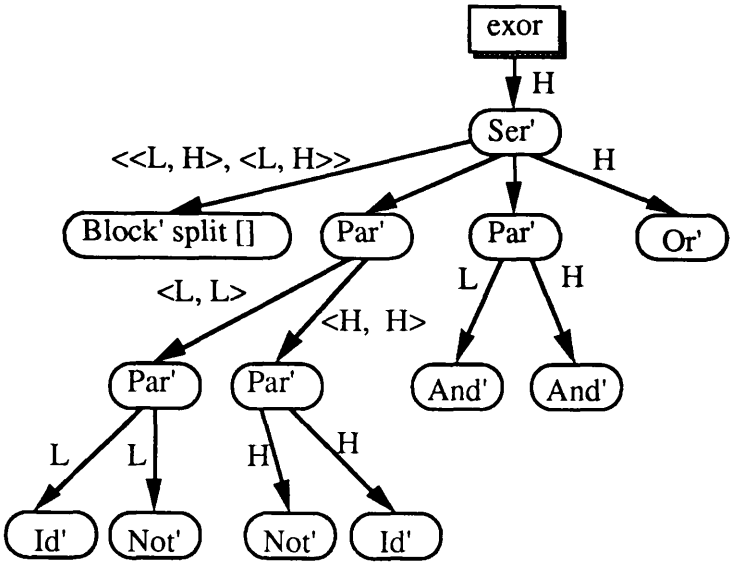


Figure 5.5: A standard interpretation of exor.

Contrast this with the graph that corresponds to the symbolic simulation non-standard interpretation shown in figure 5.6.

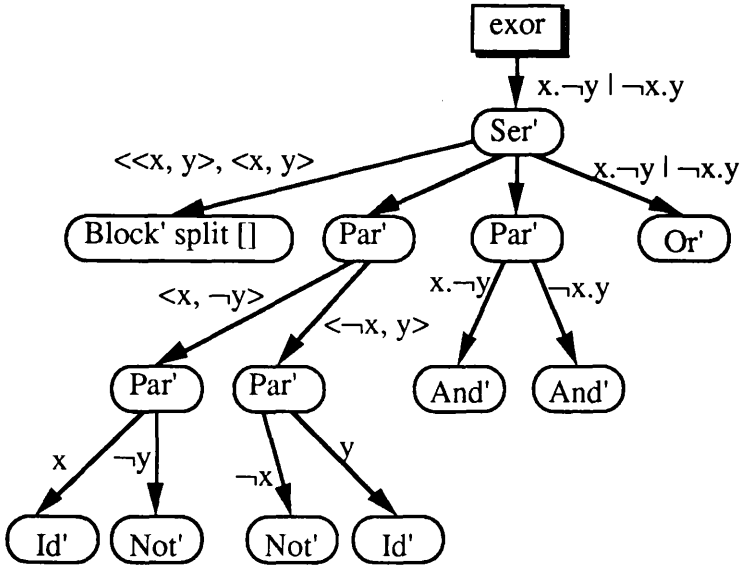


Figure 5.6: Symbolic Simulation NSI of exor.

For clarity, *NotSym* is represented by ¬ (prefix), *AndSym* by . (infix) and *OrSym* by | (infix).

These two graphs are isomorphic: we are computing over the same structure. The same Ruby description is analysed by two different interpretations. This gives rise to automatic consistency between the circuit specification used for description (behaviour) and the circuit specification used for other analyses.

One weakness of this technique as it stands is the inability to examine the value of internal nets. This is a crucially important requirement, since all testability analyses are concerned with these internal connections.

5.4 Labelling Nets

Many analyses require the circuit's nets to be uniquely labelled. For example, in deductive fault simulation, we talk about node *n* stuck at some value. Should Ruby descriptions be annotated by the designer? This would require *every* node to be given a unique name.

There are several reasons why this is a bad idea. Firstly, this would make Ruby descriptions ugly by littering them with distracting information. A large part of Ruby's elegance arises from its carefully designed syntax which makes apparent certain characteristics of designs. Hence the mathematical-style notation rather than an intractable verbose VHDL style.

Many hardware description languages require nets to be explicitly labelled, and most net representations like EDIF [EDIF Comimittee] rely on all nodes having names. However, using a simple labelling scheme, it is not possible to label every Ruby expression. Consider the circuit map and. It is not possible to tell how many nets there are until this circuit is given some input, like ⟨⟨L,H⟩, ⟨H,H⟩⟩ or is just as part of some other circuit which fixes the size of the map e.g. map and ; and. Recursive descriptions often describe circuits whose dimensions depend on the size of the data.

Instead of trying to cope with circuits which have this kind of data dependency, we shall automatically label internal nets of only fixed-size Ruby circuits. By 'fixed-size' we mean a circuit for which the number of internal nodes can be determined even if it contains generic combining forms like map. The size can be fixed by constraining the generics by using them with fixed size circuit builders (e.g. the combinational gates) or by applying an input of known size. The input itself is not important.

Whatever labelling scheme is chosen, it must be easily understood by the designer, because he or she will have to be able to identify internal nets from the label assigned. Later, we shall see how to produce a graph-like representation with internal arcs labelled. Rather than examining a circuit diagram that corresponds to a Ruby description with a view to finding a suitable labelling we shall use the Ruby description itself for labelling.

Each net (which corresponds to an arc in the abstract syntax tree) is to be labelled by the processing node that drives it. We constrain ourselves to circuits which have at the most one output of a processing node connected to a net.

The following labelling scheme is used. We assume that we have an infinite supply of numerical labels (whole numbers) starting with 1. Let the *current label* be the next free label which has not been used. To label a combinational node we assign it the current label and then increment the current label. Wiring circuits do not consume labels: they just carry labels between combinational nodes. To label a composite (or higher-order) constructor like serial or parallel composition, we label the constituent circuits from the left to the right.

The type *RUBY* presented earlier is redefined to allow the basic gate constructors *And'*, *Or'* and *Not'* to hold values by making their arity one. This is done by making *RUBY* a polymorphic type. In the labelling interpretation, we specialise this polymorphic type to integer values to allow us to attach label values to the basic gates. This extra value is written as a subscript to *And'*, *Or'* and *Not'*. Where it is omitted, its value is not needed and is assumed to be undefined.

The labelling interpretation is then defined using the following definition of $\mathcal{L}$:

$$
\begin{array}{lcl}
\mathcal{L}{:}f\,[\![\,And'\,]\!]\,c & = & And'_c \\
\mathcal{L}{:}f\,[\![\,Or'\,]\!]\,c & = & Or'_c \\
\mathcal{L}{:}f\,[\![\,Not'\,]\!]\,c & = & Not'_c \\
\mathcal{L}{:}f\,[\![\,Id'\,]\!]\,c & = & Id' \\
\mathcal{L}{:}f\,[\![\,Fork'\ n\,]\!] & = & Fork'\ n \\
\mathcal{L}{:}f\,[\![\,Ser'\ (x{:}xs)\,]\!]\,c & = & Ser'\ (x_c : (\mathcal{L}{:}f\,[\![\,Ser'\ xs\,]\!]\ (c + \#x))) \\
\mathcal{L}{:}f\,[\![\,Par'\ (x{:}xs)\,]\!]\,c & = & Par'\ (x_c : (\mathcal{L}{:}f\,[\![\,Par'\ xs\,]\!]\ (c + \#x)))
\end{array}
$$

The definition of L over other combining forms follows in a similar manner. This definition used a function # which operates over abstract syntax descriptions. This returns the number of labels consumed by a fragment of abstract syntax, and corresponds directly to how many processing nodes are found. A partial definition is:

$$
\begin{array}{lcl}
\#\,And' & = & 1 \\
\#\,Or' & = & 1 \\
\#\,Not' & = & 1 \\
\#\,Ser'\ [\,] & = & 0 \\
\#\,Ser'\ (x{:}xs) & = & \#x + \#\,Ser'\ xs
\end{array}
$$

For example, the labelling of the exor circuit as defined above is:

$$exor = split\ ;\ [[\iota, not_1], [not_2, \iota]]\ ;\ [and_3, and_4]\ ;\ or_5$$

where each combinational node is subscripted by its label. Notice that by giving a different Ruby description, we can get a different labelling:

$$exor2 = split\ ;\ [[\iota, not_1]\ ;\ and_2, [not_3, \iota]\ ;\ and_4]\ ;\ or_5$$

Thus, it is not possible to simply look at the circuit diagram and label the internal nets. We must label Ruby descriptions themselves.

The labelling scheme outlined above is easy to implement and is also straightforward for a human to perform. This labelling scheme can also be implemented as a non-standard interpretation, as shall be shown later. The exclusive-or graph is labelled using this scheme in figure 5.7.

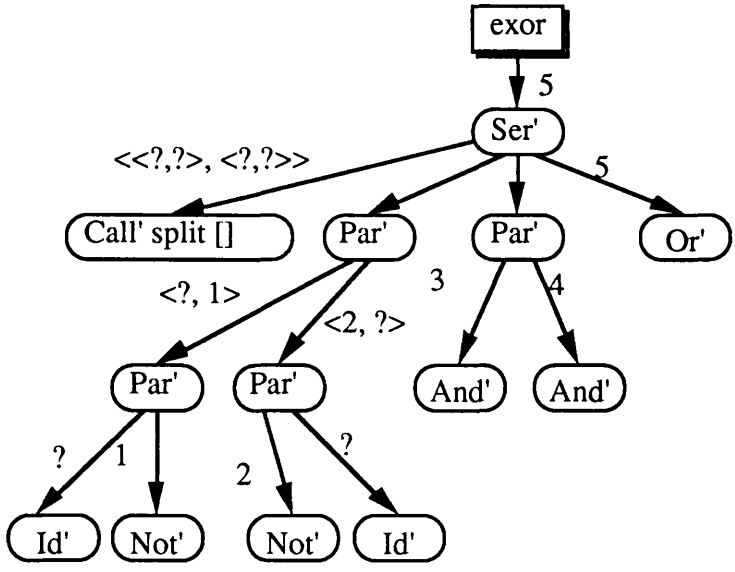


Figure 5.7: Labelling of exor.

Notice that certain arcs which are connected to primary inputs have undefined labels because they are not driven explicitly by a combinational gate. The algorithm for labelling could be amended to deal with input wires as a special case, but a better solution seems to be to provide a special component, say inpad, that represents an 'input node' (rather like an input pad). This node is treated like a combinational node when labelling i.e. it increments the current label, but it behaves like a wire i.e. does not modify the incoming information. A third description of an exclusive-or gate can then be given as:

$$exor3 = [inpad_1, inpad_2]\ ;\ split\ ;\ [[\iota, not_3], [not_4, \iota]]\ ;\ [and_5, and_6]\ ;\ or_7$$

which properly labels the primary nets. However, we are often not interested in the values at the primary nets, so we shall often not bother to use inpad.

5.5 Internal Connections

It would be useful to have as output from an interpretation a graph (like figure 5.5) which gives not only the value at the output, but also the values on all the internal arcs. This information is especially useful in symbolic simulation when the behaviour returned at the output does not match the expected behaviour. The graph could be analysed to discover where the behaviour of the implementation departs from the specification, thus reducing the size of the implementation that has to be debugged.

There are more pressing reasons for being able to observe internal nodes. Many testability analyses compute valuable data about internal nets. Using the scheme described above, this data is locked 'inside' the circuit since we are only able to observe the primary outputs.

There are various ways to get at the information locked in the internal nets. The first method adopted was to change the non-standard values to be tuples. One element of the tuple contained the 'result' from the previous combinational node i.e. the same value as before, and the second element contained a set of node assignments. A node assignment is itself a pair of node numbers and values at that node. To get the values at the internal nets, we gather together all the node assignments appearing at the outputs (by taking their set union) and tabulate the results on node numbers.

Running such an interpretation on the exclusive-or gate example with input ⟨L, H⟩ using the standard interpretation would give output like:

Node	Value
1	L
2	H
3	L
4	H
5	H

The node column could also be annotated with the kind of gate the label refers to by a slight modification to the non-standard value. However, output of this type is difficult to analyse. It would be preferable to have output which resembles the decorated graph shown in figure 5.5. For example:

exor <L, H> = split ; [[ι, not_1 L], [not_2 H, ι]] ; [and_3 L, and_4 H]; or_5 H

Then next section considers one way of achieving this by composing interpretations.

5.6 Composing Interpretations

Since application of an interpretation can be considered to be a transformation (or function) from one graph to another, a natural extension is to allow two transformations over isomorphic graphs to be composed.

Consider the labelling example. The interpretation $\mathcal{L}$ takes a Ruby description (a graph) and some data and returns a graph as a result. Because of the way higher order combining forms are elaborated, the graph returned may not be isomorphic to the circuit description graph. For example, every instance of map is replaced by the corresponding parallel composition. We apply interpretations only to circuit descriptions of fixed-size so we are sure that the graphs will be isomorphic between composed interpretations.

The graph returned by the standard interpretation forms a Ruby description which can then be analysed by another interpretation using its own non-standard values. In the labelling example, we want to apply the labelling interpretation to the graph annotated with standard values, to return a graph annotated with label and logic value pairs.

To allow such a combination to be expressed, another combining form is introduced over interpretations: serial composition. We shall denote this closed operator by ;. When two circuits are composed, we have to provide a pair as 'input' data. The first element of the pair is the input to the first interpretation and the second element of the pair is the input to the second interpretation. The meaning of serial composition over interpretations is defined by using interpretation that have their direction of analysis partially applied:

$$\langle a, b\rangle\ (\mathcal{X}\,;\mathcal{Y})cir \ \Leftrightarrow\ \mathcal{Y}(\mathcal{X}\,cir\ a)\ b$$

Informally, interpretation $\mathcal{X}$ analyses *cir* with input a and returns a new annotated graph as its result. This graph is analysed by interpretation $\mathcal{Y}$ with input b to return a third annotated graph which is the result of the serial composition. Both interpretations use the same environment. Note that composition is defined not over interpretations (which are parameterised on a direction and of type $direction \rightarrow RUBY' \rightarrow V1 \rightarrow V2$) but on interpretations in a given direction (i.e. the direction is partially applied giving a function of type $RUBY' \rightarrow V1 \rightarrow V2$).

The new labeling interpretation *L2* can now be defined in terms of $\mathcal{S}$ and $\mathcal{L}$:

$$L_f = \mathcal{L}\, F$$
$$S_f = S\, F$$
$$L2 = S_f;\ L_f$$

This composite interpretation expects as input a pair $\langle a, b \rangle$, where a is a standard value (e.g. tuple of logic values) and b is the number to start labelling from. Figure 5.8 shows the graphs constructed in the *L2* labelling of the circuit and ; not.

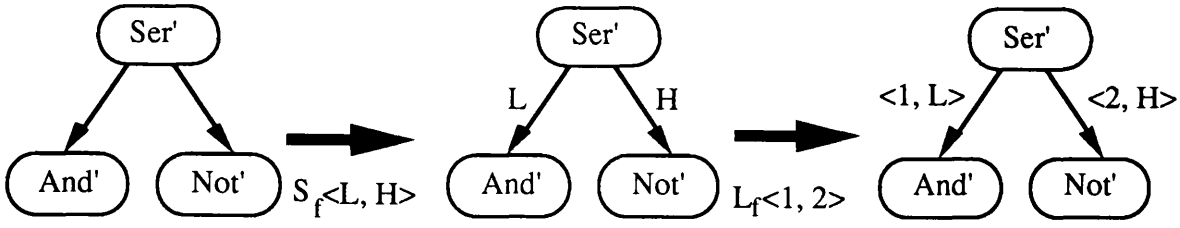


Figure 5.8: *L2* interpretation for a NAND gate.

5.7 Conclusions

Starting from the standard semantics, various adaptions have been explored in an attempt to find a good method for non-standard interpretation. The simplest way to give an alternative semantics is to define one from scratch, but this is unsatisfactory because much work is duplicated. Intuitively we might think of non-standard interpretation as providing alternative semantics for the processing nodes, so the semantics could be parameterised on the definitions of these nodes. However, we adopt a more powerful system that allows any Ruby language feature to be redefined.

Various ways of combining interpretations to produce new interpretations have been presented. Interpretation overriding provides the mechanism for making a non-standard interpretation by adapting an existing interpretation. Interpretation composition combines interpretations to produce new interpretations and is useful for developing complex analyses in a modular fashion. A very useful interpretation built in this way is the labeling interpretation.

The non-standard interpretations presented all analyse isomorphic circuit descriptions,

so we avoid the problem of inconsistency between the standard circuit representation and different representations used for other analyses.

Chapter 6

Applications of NSI

6.1 Introduction

In the last chapter, a technique for implementing non-standard interpretation was proposed. This technique is evaluated by using it to build various analysis tools. The tools implemented as non-standard interpretations are deductive fault simulation and SCOAP testability measure. An alternative circuit representation is also considered for anlayses where are net based rather than node based. We also discuss the technique of partial evaluation for circuit analysis and how this method can be easiy cast as a non-standard interpretation.

We show that one of the weakness of non-standard interpretation with respect to abstract interpretation is that we lose the ease with which safety properties can be proved. A method for attempting to recover safety properties by combining interpretations is presented.

6.2 Deductive Fault Simulation Interpretation

Now that internal nets can be labelled and combinational gates can have their semantics changed, there is enough machinery available to preform deductive fault simulation by non-standard interpretation. The output from a deductive fault simulation program is a set of stuck-at faults that can be detected by a given test pattern. These stuck-at faults give information about internal nets, so the circuit must be labelled. This can be done using the L interpretation. However, recall that deductive fault simulation works by 'deducing' which faults can be detected by analysing the correct behaviour of the circuit. This suggests that

deductive fault simulation encapsulates the standard interpretation. Instead of re-specifying the standard interpretation, we can use the *L2* interpretation which produces a labelled graph annotated with standard values.

All that remains now is to make an interpretation that takes an *L2* labelled description and (empty) fault sets as input and produces as output the faults that can be detected by the given test pattern (embedded in the *L2* annotated graph).

The non-standard semantics to be attributed to the combinational gates by deductive fault simulation (section 3.5) is given by:

$$\begin{aligned} &and\_ded\ \langle n, v\rangle\ \langle\langle X, x\rangle, \langle Y, y\rangle\rangle = \langle \not\subset_n\ (\not\subset_x X \cap \not\subset_y Y) \cup \{n/\neg v\}, n\rangle \\ &or\_ded\ \langle n, v\rangle\ \langle\langle X, x\rangle, \langle Y, y\rangle\rangle = \langle \not\subset_n\ (\not\subset_x X \cup \not\subset_y Y) \cup \{n/\neg v\}, n\rangle \\ &not\_ded\ \langle n, v\rangle\ \langle X, x\rangle = \langle \not\subset_v\ (\not\subset_x X \cup \{n/\neg v\}, n\rangle \end{aligned}$$

Note the extra first parameter refers to the value deposited at the node from the previous (*L2*) interpretation, and the second parameter is the data coming into this node through 'wires'. The non-standard values like $\langle X, x\rangle$ are pairs: the first element is a set of faults and the second element is the standard value for the net which is transmitting the fault set.

The symbol $\not\subset$ is used to describe a conditional set complementation operation. When applied to a set X with a subscripted boolean (or logic) value, $\not\subset_x X$ means complement set X (with respect to $X \cup Y$) if x is true. If x is false then X is unaltered. When $\neg$ is used in a fault set and applied to a boolean value, it simply denotes boolean negation. For example, $\{n/\neg v\}$ refers to the stuck-at fault for node n which is of opposite polarity to the correct simulation value at the node n i.e. the fault at the output of a gate.

The result of each of the definitions is a pair: the first element gives the set of faults that is propagated past this gate, along with the fault detectable at the output of this gate. The second element is the correct logic value at the output of the gate. Note that these 'correct' logic values are not computed here: they have already been worked out by the *L2* interpretation. Here, they are just passed to successor nodes to allow the fault sets to be properly complemented.

This interpretation, called *DS*, can now be given as:

$$DS\,f\,[\![\,And'\,]\!] = and\_ded \tag{6.1}$$

$$DS\,f\,[\![\,Or'\,]\!] = or\_ded \tag{6.2}$$

$$DS\,f\,[\![\,Not'\,]\!] = not\_ded \tag{6.3}$$

Note that the backward case has been left undefined. Deductive fault simulation makes sense only for the forward case. The backwards analysis asks the question "what input(s) do I need to test for the following set of faults?" which is an extension of the test pattern generation problem. We choose to tackle test pattern generation as a separate problem (shown later). It is interesting to note that our organization of interpretations into forwards and backwards analyses has pointed out a fundamental relationship between deductive fault simulation and test pattern generation.

The *DS* interpretation can be used to build a specification for a deductive fault simulator. This interpretation produces at the output sets of faults covered by a given test pattern. The union of all the fault sets at the outputs give the final result of the deductive fault simulation.

The *DS* interpretation takes as input pairs. For example, to analyse the exor description for the input ⟨L,H⟩ we would give ⟨⟨L, H⟩, ⟨⟨{ },L⟩,⟨{ },H⟩⟩⟩ as input. The first element contains logic values which are used to produced a fixed circuit circuit for labelling. The second element of the tuple contains a tuple itself of pairs of fault sets and logic values which is used by the *DS* interpretation. Such a fault simulator is called $\mathcal{DEDSIM}$ and is defined as:

$$\mathcal{DEDSIM} = L2 \; ; \; (DS_f \oplus S_f)$$

where $DS_f = DS\,f$ and $S_f = S\,f$.

One pleasing aspect of having formulated the deductive fault simulation problem in this manner is that it has been decomposed into several sub-problems which have been solved independently. The structure of the division corresponds directly to the different interpretations used. This agrees with good software engineering practice: the problem is divided into smaller problems whose solutions are then composed. The methods for composition used here are overriding and serial composition of interpretations.

6.3 SCOAP TM Interpretation

Like deductive fault simulation, the SCOAP testability measure algorithm requires the internal nets to be uniquely labelled. It is essential that the values at internal arcs can be observed. The SCOAP algorithm, as presented in chapter 3, has non-trivial data dependency. Note that to compute the observability of the output net of a node *N*, we must have already computed all the controllabilities from the primary inputs to node *N*, and we must have computed all the

observabilities from the output of node N to the primary outputs. This seems to require simultaneous flow of information forwards and backwards, and perhaps suggests an implementation in a logic language like PROLOG.

However, note that the SCOAP problem can be split into two stages. The controllability values can be computed without knowing any of the observability values. This can be done using a simple left to right interpretation. Once a circuit has been annotated with controllability values, we are in a position to compute observability values by working backwards from the primary outputs to the primary inputs. This can be accomplished by using a backward interpretation. Composing these two interpretations with a labelling interpretation will give us a SCOAP testability measure interpretation.

Let the forward controllability measure interpretation be *CONT*. This can be defined immediately from the definition in chapter 3 as:

$$\textit{and\_cont}\ n\ \langle v, \langle cx, cy\rangle\rangle = \langle\langle cx, cy\rangle, \langle \min[\textit{set0}\ cx, \textit{set0}\ cy] + 1, \textit{set1}\ cx + \textit{set1}\ cy + 1\rangle\rangle$$

$$\textit{or\_cont}\ n\ \langle v, \langle cx, cy\rangle\rangle = \langle\langle cx, cy\rangle, \textit{set0}\ cx + \textit{set0}\ cy + 1, \min[\textit{set1}\ cx, \textit{set1}\ cy] + 1\rangle$$

$$\textit{not\_cont}\ n\ \langle v, cx\rangle = \langle cx, \langle \textit{set0}\ cx + 1, \textit{set0}\ cx + 1\rangle\rangle$$

$$\textit{set0}\ \langle x, y\rangle = x$$
$$\textit{set1}\ \langle x, y\rangle = y$$

$$\textit{CONT}\ f[\![\textit{And'}]\!] = \textit{and\_cont} \tag{6.4}$$
$$\textit{CONT}\ f[\![\textit{Or'}]\!] = \textit{or\_cont} \tag{6.5}$$
$$\textit{CONT}\ f[\![\textit{Not'}]\!] = \textit{not\_cont} \tag{6.6}$$

This interpretation does not use any node annotations. Each node should hold information about the controllabilities of the input and output nets to that node. For this reason, the non-standard values are not just a pair of controllability measures. Instead, we use a pair whose first element is the controllabilities of the input nets of the *previous* node, and the second element containing a pair of controllability measures. This slight contortion arises from the fact that SCOAP is really a net-based analysis which is being cast in a node-based framework.

Assuming a graph has been annotated with controllability measure, a backward observability measure *OBSV* can be defined as:

$$\begin{aligned} &and\_obsv\ \langle\langle cx, cy\rangle, v\rangle\ obsv \\ &\quad = \langle\langle cx, set1\ y + obsv + 1\rangle, \langle cy, set1\ x + obsv + 1\rangle\rangle \end{aligned}$$

$$\begin{aligned} &or\_obsv\ \langle\langle cx, cy\rangle, v\rangle\ obsv \\ &\quad = \langle\langle cx, set0\ y + obsv + 1\rangle, \langle cy, set0\ x + obsv + 1\rangle\rangle \end{aligned}$$

$$\begin{aligned} &not\_obsv\ \langle\langle cx, cy\rangle, v\rangle\ obsv \\ &\quad = \langle cx, obsv + 1\rangle \end{aligned}$$

$$OBSV\ b\ [\![And']\!] = and\_obsv \tag{6.7}$$

$$OBSV\ b\ [\![Or']\!] = or\_obsv \tag{6.8}$$

$$OBSV\ b\ [\![Not']\!] = not\_obsv \tag{6.9}$$

This interpretation uses the controllability annotations at the combinational nodes to compute the observability values. The graph return is annotated with a pair: the first element contains controllability information and the second element is an observability measure. Notice that this interpretation is only defined for the backwards case.

SCOAP can now be described by composing these two interpretations:

$$SCOAP = L2\ ;\ (CONT\ f)\ ;\ (OBSV\ b)$$

Once again, the problem has been divided into sub-problems which have been solved independently. The SCOAP analysis was expressed as the composition of three sub-analyses: (i) labelling, (ii) controllability measure and (iii) observability measure. The observability interpretation is a backwards analysis: we have simplified the task posed by the apparently bi-directional nature of the problem by chosing interpretations that are uni-direction. It is easy in the case of SCOAP to find such a division.

6.4 Inverting Nodes and Arcs

The interpretation models presented so far have been 'node centred'. By this, we mean that they are concerned with analysing characteristics of nodes like AND gates. Many analyses are certainly node based e.g. counting the number of gates in a circuit.

However, many other analyses seem to be more 'net based' rather than node based. This is especially true of testability analyses which compute information about internal nets. The SCOAP testability measure involved a slight contortion with node values which allowed the testability information of surrounding nets to be held.

This suggests that if we are really performing net based analyses, we should represent nodes in the graph by nets and arcs by 'components' (which are, confusingly, called nodes in the circuit!). This gives us a netlist view of a circuit, rather like an EDIF description. How would non-standard interpretation proceed in such a representation? Instead of re-defining nodes, we re-define arcs with non-standard semantics. Nodes contain nets which hold non-standard values.

We choose not to use this method for non-standard interpretation for two reasons. First, it is not too difficult to pose a net based analysis as a node based analysis. This has been done for both deductive fault simulation and SCOAP. Secondly, the spirit of non-standard interpretation seems to suggest that we analyse the *same* (isomorphic) description with different semantics. If we flipped the nodes and arcs of a Ruby abstract syntax tree, we would be analysing a slightly different description. This different description is strongly related to the original description because there is a homomorphisim that relates the two representations (the homomorphism that flips nodes and arcs). We guess that most analyses are node based rather than net based. However, in the field of testability, many important analyses are naturally net-based, but these can be dealt with by our system.

6.5 Partial Evaluation

Partial evaluation is the evaluation of expressions in the source code of some language at compile time. This is often possible if enough information is available at compile time. One advantage of partial evaluation is that some expressions can be replaced by their values at compile time. This leads to savings at run time. A novel application of partial evaluation has been found in the field of automatic compiler generation [Peyton Jones 85, Launchbury 90].

We have implemented a simple partial evaluation system as a non-standard interpretation. It uses some of the usual laws of switching algebra to simplify boolean expressions. The representation of expressions is similar to that used by the symbolic simulation interpretation

presented in chapter 5. A listing of the code for the non-standard interpretation can be found in appendix A.3. Some of the simplifications implemented are:

```
> simplify (NotSymbol (NotSymbol x)) = x                    ¬¬x = x
> simplify (NotSymbol (AndSymbol x y))                      ¬(x ∧ y) = ¬x ∨ ¬y
> = OrSymbol (simplify (NotSymbol x)) (simplify (NotSymbol y))
> simplify (AndSymbol SymbolTrue x) = x                     x ∧ false = false
```

The simplification function tries to make use of as much information as possible to perform calculations at compile time. Product of sum expressions are transformed into sum of products expressions since this is the canonic representation used by many analysis tools, e.g., Quine and McCluskey tabular minimization tools.

By applying this partial evaluation before other analyses, we can reduce the total amount of work that has to be done by simplifying the original description. This saving is especially worthwhile when several interpretation stages are composed together.

It may not always be desirable to apply such a partial evaluation. Some interpretations may want to analyse the original formal description without any alterations. One such example is hazard detection, which requires finding redundant circuits. These redundancies may be inadvertently removed by the partial evaluation system.

A partial evaluation analysis returns as its result a symbolic expression. This has to be converted into a Ruby description before the results of the analysis can be used by other interpretations. This is a difficult task to perform automatically, even when the original formal description is available. However, if a circuit description can be reconstructed from the output of this interpretation, then we have found a new type of non-standard interpretation. Here, we have an example of an interpretation which allows us to perform *transformations* on the formal description analysed.

This represents an increase in the power of non-standard interpretation, since many more circuit analyses could be represented if transformed circuit descriptions were returned rather than isomorphic circuit descriptions. Although we have done some work on reconstructing circuits from symbolic expressions, at the moment there seems to be no satisfactory method for doing this. Our method involves analysing boolean expressions containing no state variables and producing a network of basic gates combined with serial and parallel composition. The resulting Ruby expression is very unreadable. For example, it is not too difficult to transform $(x \wedge y) \vee \neg z$ to [AND, ι] ; [ι, NOT] ; OR which can be simplified to [AND, NOT] ; OR.

Future work could look for methods of transforming a formal description given the source and target symbolic expressions. The key might be to seek a good algorithm for finding the difference between two symbolic expressions and then identifying what portion of the formal description this difference corresponds to. This will probably require the use of existing automatic analysis and verification tools.

In several areas of hardware description analysis and program analysis we have found strong analogies and techniques which are applicable to both types of descriptions. Non-standard interpretation is an obvious example. We believe the meaning of partial evaluation is similar in these two types of descriptions. In programming languages, partial evaluation analyses the source code to produce a more efficient, but semantically equivalent, program. This is then compiled to better object code. In hardware, partial evaluation analyses hardware descriptions to produce behaviourally equivalent descriptions. These descriptions correspond to hardware which operates more quickly and uses fewer components. Thus, in both cases, the quality of the realisation (object code for programs, hardware for HDLs) is improved.

6.6 Combining Interpretations

One very useful feature of abstract interpretations is that we can prove that they conform to some safety criterion w.r.t. the standard interpretation. For example, the behaviour of a strictness analysis can be checked against the standard interpretation to make sure that only correct approximations are found. This is possible because the abstract behaviour is just part of the standard behaviour: a cut down version or an approximation. The abstract values are just approximations of the standard values. This fundamental relationship between standard and abstract interpretations makes many properties of the abstract model easier to establish and verify (including correctness and safety).

Unfortunately, non-standard interpretation does not share this property. This is because the relationship between the standard and non-standard interpretations can be completely arbitrary. Also, non-standard values need not be linked to standard values in any way. For example, there is little similarity between running an exor circuit with input ⟨L, H⟩ and computing its SCOAP testability measures. Indeed, these are two different kinds of analysis. One analysis is *dynamic* and the other is *static*. Simulation is dynamic because it needs some input and a circuit description before it can give a result. However, SCOAP attempts to

compute an approximation to the testability of a circuit by considering only the structure of the circuit. Since SCOAP does not need any 'input', it is a static analysis.

The standard interpretation with logic values is not rich enough to perform useful abstract interpretations. This is because the basic data entity is a boolean variable, which does not give us much to abstract from! Tuples of booleans are the most complex data values that occur in the standard interpretation, but even these do not contain enough information.

What is ideally required is a 'super-interpretation' from which we can abstract enough information to allow us to make abstract interpretations for simulation, testability measure, deductive fault simulation, labelling etc. This interpretation would propagate along its 'wires' complex data object which contain sets of faults, testability measures, logic values etc.

Such a super-interpretation is a very contrived object, and of course does not exist in any useful form. The analysis of circuits is performed by a set of tools which are detached in their operation rather than being rolled into one gigantic analysis tool.

We could artificially build a super-interpretation by combining existing interpretations. This poses several questions. Firstly, how are interpretations combined? Can they be blended together, factoring out common functionality? This is a difficult analysis to perform. Let us just represent the combination of n interpretations by placing them into an n-tuple. Then, abstract interpretation involves uses a *projection* function to extract the required sub-behaviour.

The next question to consider is how to represent data values in a super-interpretation. Again, attempting to merge data by factoring seems like a difficult task, so we shall just tuple the data values in a similar manner to how the interpretations are tupled. For example, the i^{th} element of a data tuple is manipulated by the i^{th} interpretation of the tuple that holds the super-interpretation.

Parallel composition over interpretations (i.e. functions) as defined in chapter 5 can be used to combine interpretations to produce a super-interpretation:

$$SUPER = [L, L2, SCOAP, DEDSIM, \ldots]$$

A new interpretation could then be made by projecting out one or more of the components from *SUPER*. Unfortunately, this method has not gained us anything. We still need to specify interpretations of interest beforehand. Any property we prove about an abstract object from *SUPER* holds only if the component interpretations of *SUPER* have been validated.

If we could construct a super-interpretation for a finite set of non-standard interpretations, then there is a possiblity for some factorisation. Many interpretations will perform the same calculations. A realisation for *SUPER* could economise on space by factoring out the common parts of such interpretations. One problem with this suggestion is that these calculations are expressed as functions over which equality tests are not usually allowed. This makes the implementation of such a scheme problamatic when using a programming language like Miranda. The main use of super-interpretations at the moment seems to be as a conceptual device.

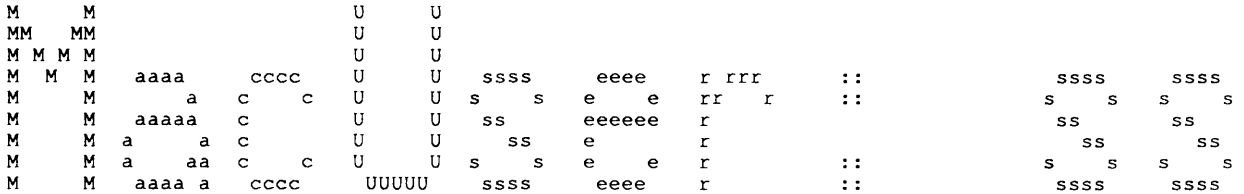
A good area for future research is to see if there is some other way of proving properties of alternative analyses which use isomorphic structures (w.r.t. standard analysis). As shown above, the traditional methods used in abstract interpretation don't seem to be too helpful, but other techniques in algebra may be readily applicable. Intuitively, one would think that there is much to inherit from the standard interpretation when analysing isomorphic descriptions.

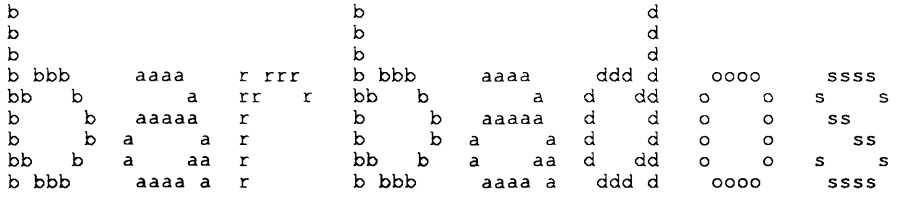
6.7 Conclusions

Non-standard interpretation has been used to specify two non-trivial circuit analyses. This was accomplished by providing a specification which is very similar to the algorithms given in chapter 2. A large part of the standard semantics has been re-used, which allows a concise descriptions of the non-standard behaviour by concentrating attention on the nodes where the non-standard processing takes place.

Not only does non-standard interpretation allow circuit analyses to be constructed very quickly and in a natural manner, but it also is a good paradigm for implementing circuit analyses. This is because many circuit analyses have a similar structure to the circuit under analysis. Combining forms have been introduced to combine small interpretations into bigger ones. This allows problems to be sub-divided into smaller problems and solved independently. This is good software engineering practise and allows circuits with complex bi-directional data flow to be modelled by a series of uni-directional interpretations. These uni-directional interpretations can be implemented efficiently using functions, rather than relations which would be required to implement a bi-directional interpretation.

In the next chapter, an implementation of a non-standard interpretation system in the style presented in the last chapter is shown.





Job: MacUser: ss
Date: Fri Nov 15 18:02:10 1991

In real life: Mr Satnam Singh
Quota: NONE
Consumed: 767 (after this job)

Fortune is brought to you by Nick to make banner pages more interesting:
Razors pain you;
Rivers are damp;
Acids stain you;
And drugs cause cramp.
Guns aren't lawful;
Nooses give;
Gas smells awful;
You might as well live.
Dorothy Parker

Chapter 7

Implementation

7.1 Introduction

The implementation of our non-standard interpretation system is presented. The implementation is in Miranda, and is machine independent. There are many reasons for using a lazy functional language for the implementation. Functional languages are much more expressive than traditional languages. They provide powerful features for combining existing programs to make larger programs, thus encouraging modularity. Richer data types are offered as standard e.g. lists. Polymorphism is a very useful feature, which is employed to describe the operation of wiring circuits. Currying is useful for making specialised interpretations from a general interpretation. Functional languages employ a terse notation which is suitable for algebraic manipulation, thus making the task of verification easier. Lazy evaluation allows allows complex data dependencies to be specified elegantly [Wadler 85].

One of our aims is to show that non-standard interpretation allows analyses to be quickly prototyped. New analyses are added by writing a small Miranda module which is incorporated into the system.

We start by giving an overview of the system software. Section 7.2 presents example scripts which the user submits to the system for execution. Details of the standard and symbolic interpretations are given in section 7.3. Section 7.4 outlines the graphical user interface and section 7.5 presents a simple attempt at implementing a non-standard interpretation system. This apporach is shown to have shortcomings which are rectified in a more complex implemention presented in section 7.6. Section 7.7 considers the impact of using Ada as an implementation language instead of Miranda, and finally, section 7.8 concludes the chapter.

7.2 System Overview

The architecture of the system is shown in figure 7.1. The entire system is written in Miranda, with the exception of some Pascal programs that were used to convert between Miranda picture data types and Macintosh PICT/MacDraw II format. A direct manipulation graphical interface also exists. There is a bolt-on graphical interface, which works under the SunView or X11 window systems. This is also implemented in Miranda, along with a SunView and X11 drivers written in Lex, Yacc and C.

From a user's point of view, the system just provides a collection of programs that can be used to analyse Ruby descriptions which have been compiled. The compiler is for a large subset of Ruby and is also part of the system. For example, if the user wanted to analyse a NAND gate circuit then the first step would be to compile the following description:

```
> nand = and ; not ;;
```

This is compiled into an abstract syntax tree form which is used by interpretations for analysis. If the above description is in a file called 'alpha.ruby' then the compiled version is deposited in the file 'alpha.env'.

The user then decides what analyses are required. This is done by preparing a script which is executed by the system. Let us prepare a simple script called 'alpha.run' to analyse the NAND gate shown above. First, we have to IMPORT from file 'alpha.env' the definition of our NAND gate. The we have to decide which analyses we want to perform. The keyword STANDARD will execute the standard interpretation taking as parameters the circuit to the analysed and the domain values to be used. Similarly, SYMBOLIC performs a symolic simulation and TRUTH produces a truth table. The behaviour of these interpretations is defined in chapters 5 and 6. Let the file 'alpha.run' be:

```
IMPORT alpha ;;

STANDARD nand {<L, H>} ;;
TRUTH nand ;;
SYMBOLIC nand {<a, b>} ;;
```

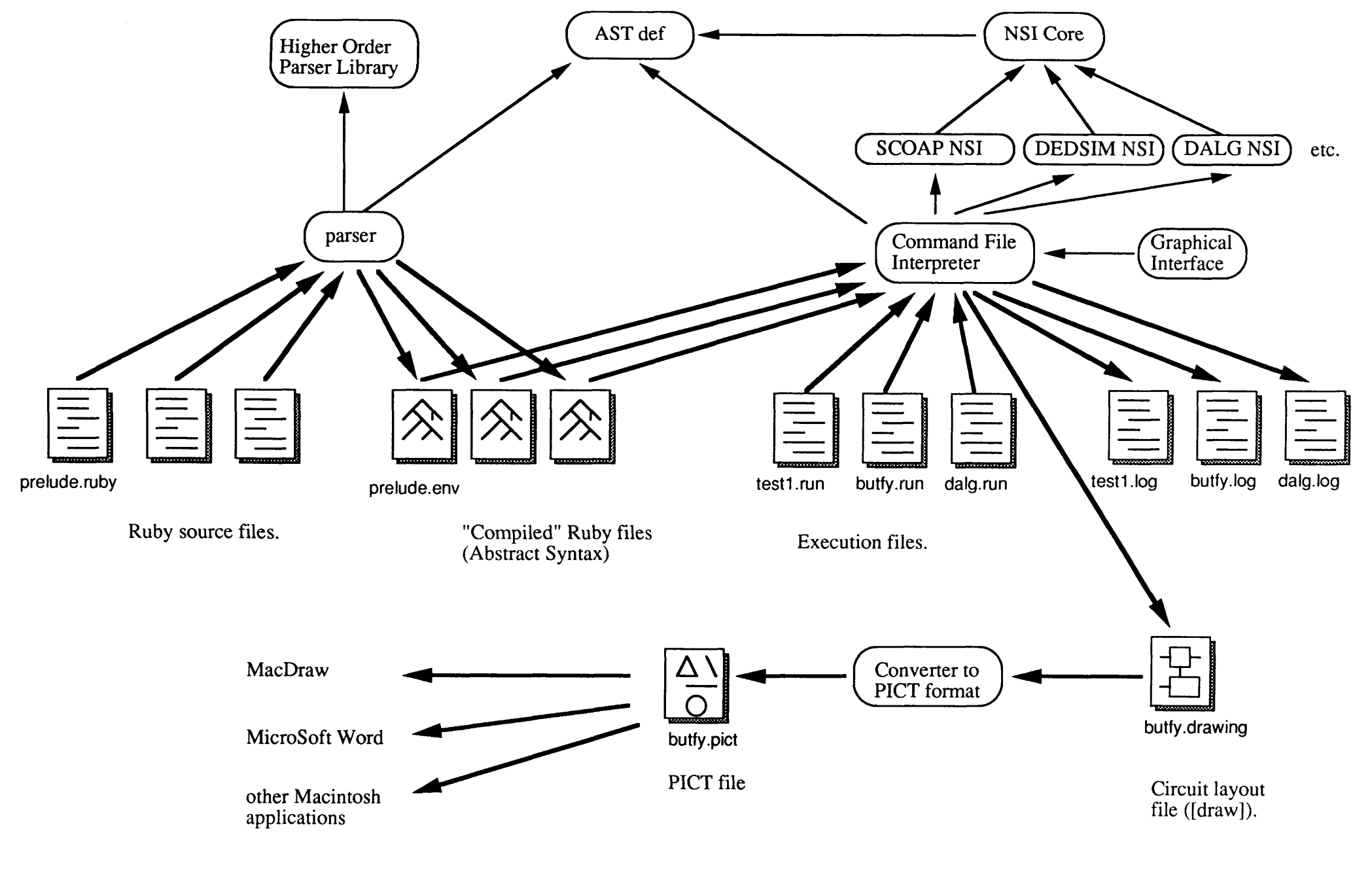
After importing the definitions from the file 'alpha.env' (the .env is omitted) the nand circuit (defined in 'alpha.env') is simulated with input {<L, H>}. Next, a truth table is requested for nand and finally a symbolic interpretation with input variables {<a, b>} is

performed. The output is placed in the file 'alpha.log'. Executing the script 'alpha.run' produces:

```
Ruby NSI System V10.12.90
-------------------------
Tue Dec 18 01:58:38 GMT 1990
Parsing file alpha.run
   1) Standard: {<L,H>} nand  {H}
   2) Truth table: nand
        <L,L> -> {H}
        <L,H> -> {H}
        <H,L> -> {H}
        <H,H> -> {L}
   3) Symbolic: {<a,b>} nand  {~(a & b)}
```

The first few lines of the output announce the name and version of the program along with the date and time the execution started. Each output is numbered and gives the name of the interpretation being performed and the circuit under examination along with domain and range values.

Output 1 shows a NAND gate which has only one value at its domain so the range also contains only one value since NAND is a function in the forward direction. Output 2 demonstrates the truth table interpretation which runs the NAND gate for every possible input. Set brackets are omitted on the domain values. Finally, output 3 shows the boolean expression that represents the behaviour of the NAND gate. Logical negation is represented by ~, conjunction by & and disjunction by |. Variable names may be of arbitary length.


Higher Order
Parser Library
AST def
NSI Core
SCOAP NSI
DEDSIM NSI
DALG NSI
etc.
parser
Command File
Interpreter
Graphical
Interface
prelude.ruby
prelude.env
test1.run
butfy.run
dalg.run
test1.log
butfy.log
dalg.log
Ruby source files.
"Compiled" Ruby files
(Abstract Syntax)
Execution files.
MacDraw
MicroSoft Word
other Macintosh
applications
butfy.pict
PICT file
Converter to
PICT format
butfy.drawing
Circuit layout
file ([draw]).

7.3 The Standard and Symbolic Interpretations

This section presents many example scripts and the output expected of our non-standard interpretation system. The following sections consider how to write such a system. The output presented in this section was produced by an actual interpretation system which is presented in a later section and has been checked manually for correctness.

A parser for the Ruby language has been written using a library of higher order parsers. Using higher order functions, we can give a YACC like specification for a grammar. This specification is executable, and we can easily associate actions with productions. This is a much more satisfactory approach to that taken in YACC, which associates C code with YACC productions.

The syntax accepted by the parser is slightly different from that used in chapter 2. This is to allow Ruby descriptions to be given in plain text files which are constrained to use the ASCII character set. The major differences are that inverse is denoted by a percent symbol (%), beside is written as <-> and below as \V/.

Each line is assumed to be a comment, unless it contains the > character as its first symbol. This is the same commenting convention that is used in Miranda and Orwell. Unlike Miranda, our parser does not use the offside rule to determine when a definition finishes. Instead, indentation has no special meaning, and each definition must be terminated by a double semi-colon (;;).

A powerful set of core operations is implemented as primitives in the non-standard interpretation system. These include serial composition, parallel composition, inverse and append. A general forking primitive is provided which takes as a parameter the number of forks to perform. Split is defined as a special case. The projections π_1 (written pi1) and π_2 (written pi2) are also defined in terms of a more general projection relation. The other common Ruby circuits are described in a prelude file. This is just a normal file which is parsed and compiled. The standard prelude (in the file '`prelude.ruby`') is shown on the next page.

Most of the definitions shown have already been explained in chapter 2. The relations `lsh` and `rsh` are used to re-organize tuples. It is surprising that these powerful relations can be expressed in terms of the small core of Ruby that has been implemented directly.

These kinds of operations have to be provided as primitives in other hardware description languages.

```
Prelude for Singh-Ruby Nov. 90


> fst R = [R, id] ;;
> snd R = [id, R] ;;

First and second.

> apl = fst [-] ; app ;;
> apr = snd [-] ; app ;;

Append left and append right.


> pi1 = project 2 1 ;;
> pi2 = project 2 2 ;;

Projection relations over pairs.


> split = fork 2 ;;

A two-way fork.

> swap = rev \ 2 ;;

Swap for 2-tuples.


> lsh = id \V/ id ;;
> rsh = id <-> id ;;

Left and right shifts e.g..
        <<a, b>, c> lsh <a, <b, c>>
        <a, <b, c>> rsh <<a, b>,c >


> rdl R = row (R ; pi2%) ; pi2 ;;
> rdr R = col (R ; pi1%) ; pi1 ;;

Reduce left and reduce right.

> irt R = (tri R) \ rev ;;

An upside-down triangle.


Some more useful wiring circuits.

> distl = row (fst split ; lsh ; swap) ; pi1 ;;
> distr = swap ; distl ; map swap
```

The usual logic gates can be defined in terms of the elementary gates. The following definitions appear in the file 'gates.ruby'.

```
> nand = and ; not ;;
> nor = or ; not ;;

> exor = split ; [[id, not], [not, id]] ; [and, and] ; or ;;
```

Variants of a full-adder appear in the file 'arith.ruby'.

```
Standard Arithmetic circuits.

> half_adder = split ; [and, exor] ;;

This circuit takes two inputs a and b and
delivers the carry and sum. half_adder <a, b> = <carry, sum>


> full_adder = snd half_adder ; rsh ; fst swap ; lsh ;
>              snd half_adder ; rsh ; fst or ; swap ;;

A full adder: full_adder <carry_in, <a, b>> = <sum, carry_out>

> horiz_adder = row full_adder ;;
> horiz_adder2 = snd zip ; horiz_adder ;;

This implements a 'flat' adder:
    <carry_in, <<x0,y0>..<xn..yn>> flat_adder <s0..sn, carry_out>
where
  xi, yi are corresponding bits to be added
  s0..sn are the sum outputs

> vert_adder = fst zip ; col (full_adder \ swap) ;;

The vertical version of the horiztonal full adder.
```

The results obtained by running a test script to exercise the standard and symbolic evaluation interpretations are shown overleaf.

```
Ruby NSI System V10.12.90
-------------------------
Tue Dec 18 01:23:38 GMT 1990
Parsing file test1.run
   1) Standard: {<L,H>} and  {L}
   2) Standard: {<H,H>} and  {H}
   3) Standard: {H} and % {<H,H>}
   4) Standard: {L} and % {<L,H>,<H,L>,<L,L>}
   5) Standard: {<L,L>} or  {L}
   6) Standard: {<L,H>} or  {H}
   7) Standard: {L} or % {<L,L>}
   8) Standard: {H} or % {<H,H>,<H,L>,<L,H>}
   9) Standard: {L} not  {H}
  10) Standard: {H} not  {L}
  11) Standard: {H} not % {L}
  12) Standard: {<L,H>} nand  {H}
  13) Standard: {H} nand% {<L,H>,<H,L>,<L,L>}
  14) Standard: {H} nor% {<L,L>}
  15) Standard: {<L,H>} exor  {H}
  16) Standard: {<H,H>} exor  {L}
  17) Standard: {H} exor% {<H,L>,<L,H>}
  18) Symbolic: {<a,b>} exor  {a & ~b \/ ~a & b}
  19) Truth table: nor
       <L,L> -> {H}
       <L,H> -> {L}
       <H,L> -> {L}
       <H,H> -> {L}

  20) Truth table: nor%
       <L> -> {<H,H>,<H,L>,<L,H>}
       <H> -> {<L,L>}

  21) Symbolic: {<a,b>} half_adder  {<a & b,a & ~b \/ ~a & b>}
  22) Truth table: half_adder
       <L,L> -> {<L,L>}
       <L,H> -> {<L,H>}
       <H,L> -> {<L,H>}
       <H,H> -> {<H,L>}
```

If set brackets are omitted on the domain, then they are added automatically. For example, when the half adder is given the input `<L, L>` in the truth table interpretation, it is automatically coerced to the singleton set `{<L, L>}`.

Output 2 shows an AND gate run forward under the standard interpretation with one value in its domain giving a singleton set at its range. Output 4 shows the result of running the gate backwards by applying {L} to the inverse of AND. This gives three possible values at the range. Running the inverse of AND with `{<L, H>}` produces an empty set.

The entire source code for the symbolic non-standard interpretation is given below. The first few lines contain include directives which make available definitions from other Miranda modules. Next, `symbolic_interp` is defined by giving a list of three triples. Each triple contains a reference to a part of Ruby abstract syntax which is to be overloaded and its forward and backward overloaded semantics. A recursive algebraic type `symbolic_rep` is used to represent syntax graphs. The functions associated with each of the basic gates are defined in a straightforward manner. Finally, the interpretation is defined using the function `standard` which takes as input a list of triples (as defined above) and returns an overriden interpretation.

```
> || Symbolic Interpretation

> %include "ruby"
> %include "standard"
> %include "~/miranda/general.lit"

> symbolic_interp
>  =         [(And', and_sym, undef),
>             (Or',  or_sym, undef),
>             (Not', not_sym, undef)]

> symbolic_rep ::= Symbol string
>                  | AndSymbol symbolic_rep symbolic_rep
>                  | OrSymbol symbolic_rep symbolic_rep
>                  | NotSymbol symbolic_rep


> and_sym n (Tuple [Symbolic x, Symbolic y]) = Symbolic (AndSymbol x y)
> or_sym n (Tuple [Symbolic x, Symbolic y]) = Symbolic (OrSymbol x y)
> not_sym n (Symbolic x) = Symbolic (NotSymbol x)


> symbolic_nsi = standard symbolic_interp
```

The interpretation is named `symbolic_interp` and consists of a list of triples. Each triple specifies some feature of Ruby to be overriden and gives the forward and backward overriding functions. The backwards case for symbolic simulation is left undefined. The data type `symbolic_rep` describes the non-standard values used in this interpretation. Here, symbolic values are represented by syntax tress in terms of variables (from primary inputs) and the three elementary logical operations.

The source code for the deductive fault simulation and SCOAP testability measure interpretations is shown in Appendix A. Examples of executing these interpretations are given below. First, we show some deductive fault simulations.

```
  1) LabelSyn: <L,L,L> apl%; full_adder
           [[-], id]%; app%; [id, fork 2; [and#1, fork 2; [[id, not#2],
[not#3, id]]; [and#4, and#5]; or#6]]; id <-> id; [rev \ 2, id]; id \V/ id;
[id, fork 2; [and#7, fork 2; [[id, not#8], [not#9, id]]; [and#10, and#11];
or#12]]; id <-> id; [or#13, id]; rev \ 2
```

```
      2) Annotate: <L,H> exor
           fork 2; [[id, not#1 L], [not#2 H, id]]; [and#3 L, and#4 H]; or#5
H
      3) Deductive Fault Simulation: <<{},L>,<{},H>> exor  <L,H>
           fork 2; [[id, not <{1/1},L>], [not <{2/0},H>, id]]; [and
<{3/1},L>, and
   <{4/0, 2/0},H>]; or <{5/0, 4/0, 2/0},H>
      4) Deductive Simulation: exor
          <L,L> -> {3/1, 4/1, 5/1}
          <L,H> -> {2/0, 4/0, 5/0}
          <H,L> -> {1/0, 3/0, 5/0}
          <H,H> -> {1/1, 2/1, 3/1, 4/1, 5/1}

      5) Deductive Simulation: half_adder
          <L,L> -> {1/1, 4/1, 5/1, 6/1}
          <L,H> -> {1/1, 3/0, 5/0, 6/0}
          <H,L> -> {1/1, 2/0, 4/0, 6/0}
          <H,H> -> {1/0, 2/1, 3/1, 4/1, 5/1, 6/1}
```

As an example, consider output 2. This shows how the exlusive-or design is labelled. Output 3 demonstrates the result of performing deductive fault simulation with input <L,H>: the output produced is <{5/0, 4/0, 2/0},H>. The states that the faults detected by the pattern <L, H> are {5/0, 4/0, 2/0} and the output value for a correctly functioning circuit should be H. Output 4 shows a deductive fault simulation of the exclusive-or circuit with every 2-tuple test pattern. This output can be used by another anlysis to determine further testability information like fault dominance.

Output 5 shows a deductive fault simulation of a half adder for every 2-tuple input. Every element of the set produced by test pattern <L, L> also occurs in some other test pattern. Thus, <L,L> can be left out of an exhaustive test program is <L, H>, <H,L> and <H,H> are included.

The following output demonstrates some SCOAP testability measures. The first five outputs show SCOAP combinational controllability and observability values. Outputs 6, 7 and 8 show only the controllability values.

```
   1) Scoap: <<?,(1,1)>,<?,(1,1)>> <<?,0>,<?,0>> half_adder
           fork 2%; [and <(1,1,2),(1,1,2)>, fork 2%; [[id%, not (1,1,6)],
[not (1,1,6), id%]]; [and <(1,1,6),(2,2,5)>, and <(2,2,5),(1,1,6)>]; or
<(2,4,3),(2,4,3)>]
      2) Scoap: <<?,(1,1)>,<<?,(1,1)>,<?,(1,1)>>> <<?,0>,<?,0>> full_adder
           [id%, fork 2%; [and <(1,1,5),(1,1,5)>, fork 2%; [[id%, not
(1,1,11)], [not (1,1,11), id%]]; [and <(1,1,11),(2,2,10)>, and
<(2,2,10),(1,1,11)>]; or <(2,
  4,8),(2,4,8)>]]; (id <-> id)%; [rev \ 2%, id%]; (id \V/ id)%; [id%, fork
2%; [and <(1,1,9),(5,5,5)>, fork 2%; [[id%, not (5,5,7)], [not (1,1,10),
id%]]; [and <(1,1,11),(6,6,6)>, and <(2,2,9),(5,5,6)>]; or
<(2,8,4),(3,8,3)>]]; (id <-> id)%;[or <(2,3,3),(2,7,3)>, id%]; rev \ 2%
```

Output 1 presents a SCOAP analysis of a half adder. Each of the inputs to the final OR gate of the have adder have the same SCOAP vectors. For these nodes the SCOAP set0

controllability is 2, the set1 controllability is 4 and the observability is 3. The SCOAP values for the output nodes are not shown because they are not of interest. Similarly, the observabilities of the primary input nodes are not of interest. The second output shows an analysis of a full adder design which makes use of the half adder design. From examining the unfolded half adders, it is clear than a hierarchical approach to SCOAP test pattern generation could speed up analysis significantly. The could be done by expressing the testability measures as relative offsets rather than absolute values.

The following outputs show SCOAP applied to smaller circuits. Note that the exlcusive-or circuit has symmetrical measures. This is what one would expect of a circuit which as 50% of its outputs as 1 and 50% as 0.

```
      3) Scoap: <?,(1,1)> <?,0> not
            not (1,1,1)
      4) Scoap: <<?,(1,1)>,<?,(1,1)>> <?,0> and ; not
            and <(1,1,3),(1,1,3)>; not (2,3,1)
      5) Scoap: <<?,(1,1)>,<?,(1,1)>> <?,0> exor
            fork 2%; [[id%, not (1,1,6)], [not (1,1,6), id%]]; [and
<(1,1,6),(2,2,5)>, and <(2,2,5),(1,1,6)>]; or <(2,4,3),(2,4,3)>
```

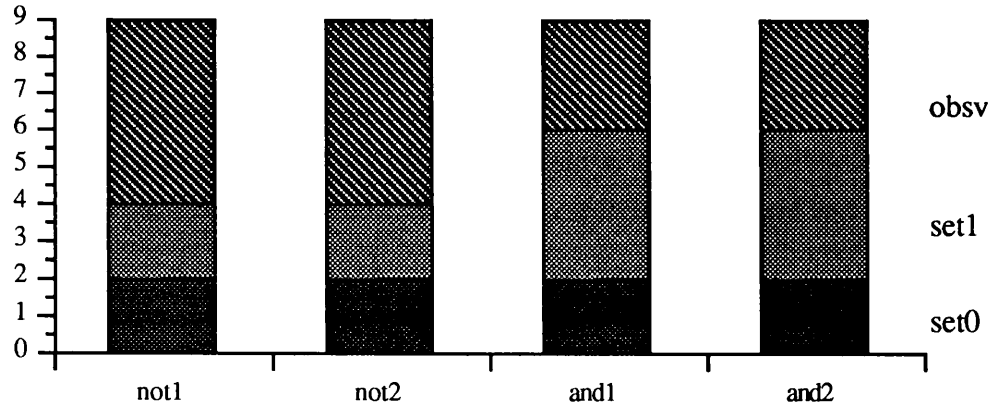
The following output shows that is also possible to run a constituent interpretation. In this case, we show the controllability interpretation. Consequently, each net is annotated with a pair rather than a triple. The outputs 6 and 7 are abstractions of the output from 5 and 2 because they can be obtained by removing information about observability values.

```
      6) Controllability: <<?,(1,1)>,<?,(1,1)>,<?,(1,1)>> fras
            [id, not (1,1), id]; [apl , id]; [id, [-]]; app; halve%; [and ,
or ]; and
      7) Controllability: <<?,(1,1)>,<?,(1,1)>> exor
            fork 2; [[id, not (1,1)], [not (1,1), id]]; [and <(1,1),(2,2)>,
and <(2,2),(1,1)>]; or <(2,4),(2,4)>
      8) Controllability: <<?,(1,1)>,<<?,(1,1)>,<?,(1,1)>>> full_adder
            [id, fork 2; [and <(1,1),(1,1)>, fork 2; [[id, not (1,1)], [not
(1,1),id]]; [and <(1,1),(2,2)>, and <(2,2),(1,1)>]; or <(2,4),(2,4)>]]; id
<-> id; [rev \ 2, id]; id \V/ id; [id, fork 2; [and <(1,1),(5,5)>, fork 2;
[[id, not (5,5)], [not (1,1), id]]; [and <(1,1),(6,6)>, and <(2,2),(5,5)>];
or <(2,8),(3,8)>]]; id <-> id; [or <(2,3),(2,7)>, id]; rev \ 2
```

The triples associated with each node give the set 0, set1 controllability and the observability for that node. For example, the last OR gate of output 1 has one input with set0 = 2, set1 = 4 and obsv = 3 (the other input has the same values).

This output can be put though a filter to produce bar charts that highlight nodes of poor testability. This makes the output much easier to digest. We have implemented a filter which transforms the output above into a format suitable for the Jazz package on Macintosh computers. The data was read into the Jazz spreadsheet from which graphs were automatically produced. This is how the SCOAP tables in chapter 2 were

constructed. The diagram below was produced by taking output 5 and filtering it through to Jazz. It was then manually touched up in MacDraw II to add the labels on the right.



7.4 A Graphical Interface

The interpretation system has been given a direct manipulation graphical interface which works under the SunView windowing system on Sun computers. This makes the system much easier to use and the output much easier to understand. The graphical system presents menus and button so that the user does not have to remember a large number of housekeeping commands. The graphical display can produce bar charts and output to other specialised programs like spreadsheets for further processing.

A special driver was written by the author to allow Miranda programs to control the SunView windowing system from a remote host over a UNIX network. This was a large undertaking, but the final result shows that functional languages can be used to write programs with better interfaces.

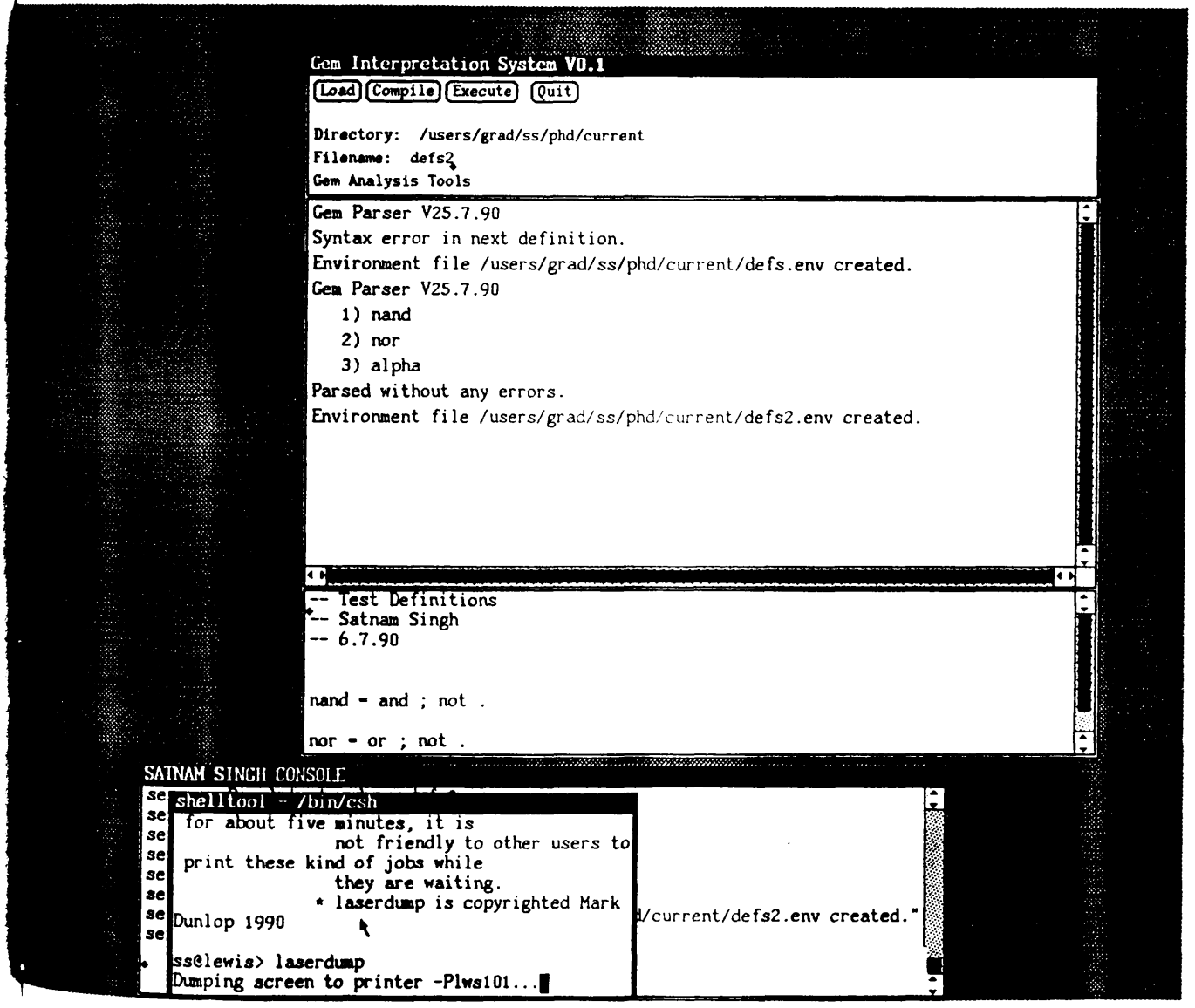
The details of the implementation of the graphical interface are outside of the scope of this thesis. A screen snapshot of one of the interfaces is shown on the next page. The interface is currently rather simplistic. There is much scope for improvement. For example, the output from the SCOAP testability measure could be automatically filtered through a function to produce histograms. At the moment, a separate tool in the system has to be manually run by the user.

Another problem with the interface is that it is hardwired to work with the SunView windowing system. It would be beneficial to devise a generic intermediate representation

for windowing operation. Unfortunately, no widely accepted representation is available and the features offered by most windowing systems tend to differ greatly.

The need to write a small SunView server in C blemishes the otherwise entirely functional language code. We require current functional languages to be extended to provide better system interfacing facilities before this problem can be resolved. However, under SunView, it is difficult to write windowing applications in *any* language other than C. This is due to the large number of C data types used to represent SunView objects and the difficulty of cross-linking between C and other languages like Pascal and Ada.

reen dump for ss on Tue Dec 18, 1990 at 02:16.



7.5 A Simple NSI System

This section presents a simple non-standard interpretation system for a subset of Ruby. We try to implement as directly as possible the semantic equations presented in chapter 5. Such a system should provide the core of the program used to implement the system outlined in section 7.2. The problems with this particular implementation provide some motivation for the more complex implementation actually produced.

The Ruby subset we use has the three basic gates, serial and parallel composition, inverse, identity and fork. The basic gates can 'store' information at their nodes: the type of the information is a polymorphic type parameter to `ruby`. The direction of flow is represented by the algebraic type `direction`.

```
> || Demonstration Toy NSI System

  Shows a naive system that won't work. Motivates the more exotic
interpretation
  system actually used in real implementation.

> ruby * ::= And * | Or * | Not * | Ser [ruby *] | Par [ruby *] |
>            Inv (ruby *) | Id | Fork num

> direction ::= F | B
```

It would be nice to make the data domain polymorphic, but this prevents a natural definition for overriding when using the Hindley-Milner [Milner 78] polymorphic type system. This is because overriding needs to have a type like $\tau\to\tau\to\tau$, where τ is the type of an interpretation. This requires all interpretations to be of the same type. Consider f_1 and f_2 and an overriding expression $f_2 \oplus f_1$. In the above scheme f_1 and f_2 will be monotypes because they are specialised to a particular task. In a strict polymorphic language like Miranda, this would require circumventing the type security rules. However, even if we use a polymorphic base type for data objects, particular interpretations will have monotypes which will in general not be compatible. For example, one interpretation might work over logic values while another works over testability vectors. Such interpretations could not be unified by overriding.

The alleviate this problem, we have adopted the 'universal type' approach. All the types we need are subtypes of a larger type. Now, all data values have monotypes and overriding does not present a problem.

```
> data ::= Tuple [data] | H | L | Cont (num, num) | Sym [char]
>          | AndSym data data | OrSym data data | NotSym data | Undefined
```

Some useful functions are defined over logic values. Pattern matching is top down:

```
> lognot :: logic -> logic
> lognot L = H
> lognot H = L

> logand :: logic -> logic -> logic
> logand H H = H
> logand a b = L

> logor :: logic -> logic -> logic
> logor L L = L
> logor a b = H
```

The function `lift` is defined to make the definition of set valued functions easier. It lifts a function that operates over one data value to a function that operates over a set of data values by repeatedly applying the base function. The results of the individual computations are combined by set union. The standard Miranda function for transforming a list into a set by removing duplicates is called `mkset` .

```
> lift f d c [] = []
> lift f d c (x:xs)= mkset ((f d c x)++(lift f d c xs))
```

When performing parallel composition of several circuits, the results of the constituent circuits have to be combined carefully so that all the correct set-values appear at the domain or range. For example, if the output of some arithmetic circuits is {1,2}, {3}, {4,5} then their parallel composition produces as output the set {⟨1,3,4⟩, ⟨1,3,5⟩,⟨2,3,4⟩,⟨2,3,5⟩}. The following function `combs` performs this computation.

```
> combs ([xs]) = [[x] | x <- xs]
> combs (xs:ys) = [x:y | x <- xs ; y <- combs ys]
```

The standard interpretation is now defined. The basic gates are defined in a straight forward manner. The top level function is `standard` which takes as its input a direction, a circuit and set of input values at the domain or range. It returns the set values at the domain or range depending on the direction of the interpretation. To make the definition simpler, we use an auxiliary function `standard'` which takes similar input as standard expect a single input value is used rather than a set of values. The function `standard'` is then lifted over a set of values by using the auxilluary function `lift`.

```
> standard = lift standard'

> standard' F (Inv r) = standard' B r
> standard' B (Inv r) = standard' F r

> standard' F (Not v) x = [lognot x]
> standard' B (Not v) x = [lognot x]
```

```
> standard' F (And v) (Tuple [a, b]) = [a $logand b]
> standard' B (And v) H = [Tuple [H, H]]
> standard' B (And v) L
>  = [Tuple [L, L], Tuple [L, H], Tuple [H, L]]

> standard' F (Or v) (Tuple [a, b]) = [a $logor b]
> standard' B (Or v) L = [Tuple [L, L]]
> standard' B (Or v) H
>  = [Tuple [L, H], Tuple [H, L], Tuple [H, H]]
```

Serial and parallel composition are given the following definitions. The first line states that the serial composition of a circuit with nothing is simply the same as computing the circuit by itself. This provides the base case for the recursive unfolding of serial composition shown on the following line. Parallel composition is defined by decomposing the input tuple and independently applying these parts to the constituent circuits of the parallel composition. We have to take every possible combination of outputs: this is computed by the function `comb`.

```
> standard' d (Ser [c]) x = standard' d c x
> standard' F (Ser (c:cs)) x = standard F (Ser cs) (standard' F c x)
> standard' B (Ser cs) x = standard B (Ser (init cs)) (standard' B (last
cs) x)
> standard' d (Par xs) (Tuple vs)
>  = map (Tuple) (combs [standard' d x v | (x, v) <- zip2 xs vs])
```

The wiring primitives are defined as:

```
> standard' d Id x = [x]
> standard' F (Fork n) x = [Tuple (rep n x)]
> standard' B (Fork n) (Tuple xs)
>  = [hd xs], if #(mkset xs) = 1 \/ #xs ~= n
>  = [], otherwise
```

We define a new symbolic interpretation by overriding it with the standard interpretation.

```
> symbolic' F (Not v) x = [NotSym x]
> symbolic' F (And v) (Tuple [a, b]) = [AndSym a b]
> symbolic' F (Or v) (Tuple [a, b]) = [OrSym a b]
> symbolic' d r v = [Undefined]
> symbolicl = lift symbolic'

> symbolic d c = (symbolicl d c) $override (standard d c)
```

Overriding is given the most natural definition:

```
> override f1 f2 v
>   = r1, if r1 ~= [Undefined]
>   = f2 v, otherwise
>     where
>     r1 = f1 v
```

However, the following examples show that there is a problem with this interpretation:

```
> nand = Ser [And 1, Not 2]


    Miranda  symbolic F (And 1)  [(Tuple [Sym "a", Sym "b"])]
    [AndSym (Sym "a") (Sym "b")]

    Miranda symbolic F nand [(Tuple [Sym "a", Sym "b"])]
    [H]
```

The standard interpretation works correctly, and the symbolic interpretation gives the correct answers when used with the basic gates or wiring primitives. However, when a higher order combining form is used, the computation locks into the standard interpretation.

We need to somehow remember what the base interpretation is. This has to be done through a parameter, making our interpretations higher order functions. We also have to extend our system to deal with block definitions and arithmetic operations as well as the remaining part of Ruby.

The next section introduces a more complex implementation for a non-standard interpretation system. Overriding is modelled by lists and list concatenation. The system also contains extensions to allow internal node values to be examined.

7.6 The Core of the Interpretation System

The core of the interpretation system is implemented by two modules called `ruby` and `standard`. These modules are refered to in the include directives shown in the listing of the symbolic simulation interpretation shown in section 7.3.

The module ruby defines the abstract syntax tree for the core subset of Ruby that we have implemented. The module contains auxiliary functions for pretty printing and environment support. It also implements a polymorphic tuple type that represents the values that flow along wires.

The definition of the abstract syntax tree used for ruby is shown below.

```
> ruby ::=  Block [char] [ruby]   || Block eloboration.
>         | Ser [ruby]            || Serial composition.
>         | Par [ruby]            || Parallel composition.
>         | Repeat num ruby       || Repeated composition.
>         | Power ruby ruby ruby    || Repeated application.
>         | Inv ruby              || Inverse.
>         | Conj ruby ruby        || Conjugate.
>         | Id                    || Identity.
>         | App                   || Tuple append.
>         | Singleton             || Singleton.
```

```
>            | Project num num     || Projection from tuple.
>            | Map ruby            || Map.
>            | Rev                 || Reverse.
>            | Swp                 || Vertical reflection.
>            | Tri ruby            || Triangle
>            | Beside ruby ruby    || Beside. 2 -> 2.
>            | Below ruby ruby     || Below. 2 -> 2.
>            | Row ruby            || Row. 2 -> 2.
>            | Col ruby            || Col. 2 -> 2.
>            | Zip                 || Zip.
>            | Trn                 || Transpose.
>            | Halve               || Halve. 2n -> [n, n]
>            | Pair                || Pair. 2n -> [2,..2]
>            | Fork num            || Multi-way fork.
>            | Restrict ruby num   || Type restriction.
>            | Loop ruby           || Feedback
>            | Delay               || Delay component
>            | Numeric num         || Numeric value
>            | Plus ruby ruby      || Addition.
>            | Minus ruby ruby     || Subtraction.
>            | Multiply ruby ruby  || Multiplication.
>            | Divide ruby ruby    || Division.
>            | Parameter string    || Parameter name.
>            | And  tuple
>            | Or tuple | Not tuple | Vdd | Vcc | Comp2 tuple|| Node.
```

Another type `ruby'` is used to identify constructors of `ruby` for overriding.

```
> ruby' ::= Ser' | Par' | Repeat' | Power' | Inv' | Conj' | Id' | App' |
>           Singleton' | Project' | Map' | Rev' | Swp' | Tri' | Beside' |
>           Below' | Row' | Col' | Zip' | Trn' | Halve' | Pair' | Fork' |
>            Restrict' | Numeric' | Loop' | Delay' |
>            Plus' | Minus' | Divide' | And' | Or' | Not' | Comp2'
```

The type `ruby'` should not be confused with *RUBY* as defined in chapter 5. *RUBY* corresponds to the Miranda type `ruby`. Rather confusingly, `Id'` in `ruby'` is not the same as *Id'* in *RUBY*. The system keeps a list of language features which can be overriden. The type `ruby'` simply ennumerates references to the language features that can be changed.

The standard interpretation is defined in the module `standard`. This also contains definitions for implementing overriding. The main export from this module is the function `standard` which is used (often by currying) to build non-standard interpretations. The function `standard` is the implementation of the model presented in chapter 5 and corresponds to a non-standard interpretation builder. An example of its use can be seen in the listing of the symbolic simulation interpretation. The first argument of this function is an interpretation. The remaining arguments are the environment, the circuit description to be evaluated and the values on the domain of the circuit (or range when the circuit is interpreted backwards).

An interpretation is implemented by a list of triples defined as follows:

```
> interp_type
>  == [(ruby', tuple -> tuple -> tuple,
>       tuple -> tuple -> tuple)]
```

Over-riding is implemented by list concatenation. When a meaning is sought for a constructor of `ruby'`, the list is searched from the beginning until a match is found. We have experimented with more elaborate representations for interpretation overriding (e.g. by using a data type that does not merge interpretations). However, these representations were discarded for the simple list concatenation method which provides us with all the power we need.

To find a meaning for some Ruby construct of type `ruby'`, the following function is used. It is called `arity_one` because it is used to attribute meanings to those constructors of Ruby that make use of values at nodes. The first parameter is the environment and the second, `tok`, is the element of `ruby'` to be over-ridden. The next parameter, `n`, gives a unique label to the node. This function is used for the forward case only: the next parameter `ff` gives the standard forward defining function. The last parameter is the value on the domain, which will always be a set in our implementation.

```
> arity_one i tok n ff (Set xs)
>  = set_union [setify (f' n x) | x <- xs]
>    where
>    matches = [f | (r, f, b) <- i ; r = tok]
>    f = hd matches
>    f' = ff, if matches = []
>       = f, otherwise
> arity_one i tok n f v
>  = error ("arity_one case: " ++ showtok tok ++ " with " ++ show_tuple
v)
```

The list of all possible matches is constructed by the comprehension on the fourth line. However, lazy evaluation ensures that only enough of the list is actually evaluated to give the first match. If the environment does not contain a meaning for the Ruby component tok, then the standard forward function `ff` is used.

The backwards case is similar.

```
> arity_one_inv i tok n bf (Set xs)
>  = set_union [setify (f' n x) | x <- xs]
>    where
>    matches = [b | (r, f, b) <- i ; r = tok]
>    f = hd matches
>    f' = bf, if matches = []
>       = f, otherwise
```

Many Ruby constructs do not need to examine the values at nodes. These are defined

using specialised versions of `arity_one` and `arity_one_inv`. These functions are useful because they automatically raise the arity of the forward and backward defining functions. This makes their definitions simpler since they don't have to involve an unused parameter.

```
> arity_zero i tok ff (Set xs)
    = arity_one i tok Undefined (raise ff) (Set xs)

> arity_zero_inv i tok bf (Set xs)
>  = arity_one_inv i tok Undefined (raise bf) (Set xs)

> raise f n x =  f x
```

An example of some functions that use the above functions are shown below.

```
> standard i env Id v = arity_zero i Id' id v
> standard i env (Inv Id) v = arity_zero_inv i Id' id v
> standard i env (And n) v = arity_one i And' n (raise logand) v
> standard i env (Inv (And n)) v
    = arity_one_inv i And' n (raise inv_logand) v
```

Identity is defined to use the standard Miranda identity function. This works at arity zero because the identity function does not need to examine the value at nodes. The AND gate is defined by using as the standard function the logical conjunction function `logand` which has its arity raised. The inverse is defined using the function `inv_logand` which is defined to be the inverse of `logand`.

These kinds of definitions are repeated for every construct of Ruby i.e. every element of `ruby'`. The entire source definition for `standard` takes up a several pages. However, this provides the basic framework for our non-standard interpretation system which pays off great dividends when defining new analyses.

The module `standard` also contains definitions for set manipulation and a variation on the function `standard` that returns the values of internal nodes. It also contains a function for automatically labelling nodes and provides many test functions. The interpretation combining functions are also defined here.

One important interpretation combining function is `cross` which is used to combine the results of two interpretations. Some of the source code for this function is shown below:

```
> cross (And a) (And b) = And (Tuple [a, b])
> cross (Or a) (Or b) = Or (Tuple [a, b])
> cross (Not a) (Not b) = Not (Tuple [a, b])
> cross (Ser xs) (Ser ys) = Ser [cross x y | (x, y) <- zip2 xs ys]
> cross (Par xs) (Par ys) = Par [cross x y | (x, y) <- zip2 xs ys]
> cross (Beside a b) (Beside c d) = Beside (cross a c) (cross b d)
```

When two interpretations are combined, the results at corresponding nodes are held in a pair (a tuple of 2 elements). This can be visualised by superimposing the results of two interpretations.

7.7 Comparison with Ada

We believe the choice of a lazy functional implementation language is an important reason why our system has been built quickly and why new interpretations are easy to add. In this section, we discuss the impact of using a modern imperative language like Ada for the implementation. The author has implemented a small subset of the system in Ada, as well as a test pattern generation program (the D-algorithm presented in chapter 8).

One of the key aspects of non-standard interpretation is the ability to produce a generic scheme that captures hardware analyses at a useful level of abstraction. In our system, this is done by overloading the standard interpretation. The use of a powerful and carefully chosen representation for interpretations has greatly simplified our implementation. One important aspect of our system is that it contains functions as first class objects. Non-standard interpretation is then implemented by overriding, which can be represented naturally by functions that manipulate data structures that also contain functions.

Ada does not treat functions and procedures as first class objects. Procedures and functions can be passed as parameters but they can not be held in data structures. Thus, whenever a new interpretation has to be added, the core of the interpretation system has to be modified for the extra functions to be introduced. Our system does not require the interpretation core files to be altered in this way. We simply write a new module and link it to the existing system.

Another important feature of functional languages is polymorphism. Non-standard interpretations are defined by using Miranda functions that operate over non-standard values. These non-standard values can be of any type. Our interpretation is constructed so that wires can carry any type of information and so that the defining functions for nodes can operate over tuples of any type. This means that a universal type encompassing all likely data types does not need to be constructed.

Ada does not have a polymorphic type system. It does support another system for generalising declarative units like procedures and packages called the *generic* mechanism which works by instantiation. However, even using a generic system, which is less

powerful than polymorphism, we still have to alter the core code of the interpretation system to explicitly declare all new non-standard values. There is no need to do this in our interpretation system.

Although the analysis of sequential circuits has not been presented in this thesis, our system can simulate sequential circuits that contain feedback loops and delay elements. This is done by changing the standard values to be streams of logic values. The definitions of the basic gates are then lifted to operate over streams. Lazy evaluation allows the stream model to be implemented in a straightforward manner.

Ada does not allow for the expression of infinite data objects, and consequently does not support lazy evaluation. Thus, streams and stream operations have to be implemented in a less direct manner compared with lazy lists in Miranda. One could construct a stream data type using Ada's concurrent tasking primitives, but this would be overkill and inelegant as well as being difficult to program.

Of course, well designed functional languages like Miranda have many other advantages over imperative languages like Ada. The points raised above emphasise three of the most important reasons why it was wise to use a lazy functional language for our implementation.

7.8 Summary

An actual implementation of a non-standard interpretation system has been shown. This is a large subset of Ruby which supports a rich set of combining forms. As the source code for the symbolic simulator shows, non-standard interpretation allows new analysers to be built by expending very little extra effort once the standard interpretation exists.

The system has been extensively tested using small to medium sized circuits. We believe that the task of formally proving the correctness of the system is greatly eased by the non-standard interpretation discipline. However, this is still a long and tedious task and is outside the scope of this thesis.

A graphical interface has been built for a large part of the system. This greatly improves the useability of the system. We note that more lines of code in the system are used to implement the user interface than anything else. Perhaps this suggests that we should be trying to build rapid prototyping tools for interfaces too.

The choice of a lazy functional language for the implementation has been a good one.

Higher order functions, polymorphism and lazy evaluation are important features of Miranda which have allowed our system to be represented elegantly. These features also allow new interpretations to be added easily. These features are not found in modern imperative languages like Ada. Such languages would make our source code more more cumbersome and new interpretations would be more difficult to add.

Chapter 8

Test Pattern Generation

8.1 Introduction

In this chapter a non-standard interpretation implementation of the D-Algorithm for test pattern generation is presented. First, the notion of a test pattern is formally defined. The D-Algorithm is then introduced using one of the many notations employed by the originator [Roth 66]. The description is then simplified by defining a partial order and using a clearer mathematical notation.

Although the D-Algorithm has much more complicated information flow than any previous interpretation implemented it is shown that it may be easily implemented without explicitly specifying backtracking. The algorithm can be decomposed into three phases which are applied in sequence. In each phase, the algorithm seeks a path in the circuit satisfying a given property. There are often many possible paths– the original implementation uses backtracking to find a suitable path. The non-standard interpretation implementation of the D-Algorithm realises each of the three phases with a unidirectional interpretation with simple data flow i.e. left to right in forward case and right to left in the backward case.

Information about all possible paths is propagated through each unidirectional interpretation. This produces a list of all possible paths to the output of each phase. However, just one path is required, so only the head of the list needs to be evaluated. Lazy evaluation ensures that the tail of the list is not constructed so this implementation does no more work than the backtracking version.

A modification of the backward phase of the D-Algorithm is used to check that two circuits exhibit the same behaviour. The technique is quite efficient when the two circuits under comparison are nearly identical. This is often the case when one

design is modified to incorporate some "engineering change" . An example of such a change is a post-hoc measure to increase testability.

To help describe the D-Algorithm, some terminology is introduced to describe faults and values on wires.

8.2 Formal Description of a Test Pattern

8.2.1 Terminology

Wires and faults

The wires or lines in circuits will be denoted by lower case letters e.g. a, b, and c. It is convenient to assume that we have available a set containing all the lines available. Let $LINES$ be such a type, so if a is a line then $a \in LINES$.

A line may be stuck-at-0 or stuck-at-1. The shorthand $a/0$ meaning a stuck-at-0 will be used in this chapter (similarly $a/1$ for a stuck-at-1). The value on a line is denoted by the set *logic* which contains $\{0,1\}$ i.e. $\{0,1\} \subset logic$.

Tuples

We are only interested in analysing combinational circuits with the D-Algorithm. To simplify the presentation, we model a circuit as a relation between tuples rather than using sequences. In particular, a circuit is modeled by a relation from an n-tuple $(a_1, a_2, ..., a_n)$ to a *logic* value. Each element in the tuple is of type *logic*. Multiple output circuits may be modeled by many single output circuits. If p is a tuple, then p_i is the i^{th} element of the tuple p. For example, if $p = (x, y, z)$ then $p_2 = y$.

Let $ALL(n)$ be the set of all tuples of size n. For example,

$$ALL(2) = \{(0,0),(0,1),(1,0),(1,1)\}$$

The set $ALL(n)$ contains 2^n elements.

All tuples in this chapter are homogenous i.e. all the elements are of the same type *logic*.

Gates

As before we model logic gates as being relations between wires [Sheeran 90]. For an n input circuit, the behavioural relation is between the sets $ALL(n)$ and *logic*.

A behavioural relation essentially represents the truth table of the gate behaviour. In the case of the AND gate this relation may be represented by a set of pairs. The first element of the pair denotes the domain (a 2-tuple in the case of an AND gate) and the second element of the pair denotes the range (a *logic* value). Thus, AND is considered as being the following relation:

$$AND : (logic, logic) \leftrightarrow logic$$
$$AND = \{((0,0),1),((0,1),0),((1,0),0),((1,1,),1)\}$$

Composite circuits may be constructed by performing relational composition with existing relations.

It is convenient to describe the operation of an AND gate by a function from its inputs to its output. It is also useful to be able to describe the function which takes a value for the output of the AND gate and returns the set of all possible input values that produce this output. We shall place an arrow over the name of a relation to extract the forward and backward functions. A left to right arrow extracts the forward function, whilst a right to left arrow extracts the backward function. For a relation R, these two operations are defined as:

$$\vec{R} = \{y : xRy\}$$
$$\overleftarrow{R} = \{x : xRy\}$$

Now $\vec{AND}$ represents the forward function for simulating the AND gate and let $\overleftarrow{AND}$ represent the behaviour for running the circuit backwards. These two functions have the following types and behaviour:

$$\vec{AND}: (logic, logic) \rightarrow \{logic\}$$
$$\vec{AND}\ (x,y) = \{z : (x,y)ANDz\}$$
$$\overleftarrow{AND}: logic \rightarrow \{(logic, logic)\}$$
$$\overleftarrow{AND}\ z = \{(x,y) : (x,y)ANDz\}$$

It is useful to have some way of obtaining the number of inputs a gate or circuit has. Let $INPUTS(G)$ be the size of the tuple that gate G is defined over. For example,

$$INPUTS(AND) = 2$$

8.2.2 Specification of a test pattern

If G is the behaviour of the working circuit, then let G_f describe the behaviour of G under the influence of fault f. To expose the fault f we have to find some input

for which G produces a different output from G_f. To be precise, we require some p such that $\vec{G}\ p \neq \vec{G_f}\ p$ (alternatively, $\vec{G}\ p \oplus \vec{G_f}\ p$, where $\oplus$ is the usual exclusive-or operator).

One way of finding such a p is to build the truth tables for the functions $\vec{G}$ and $\vec{G_f}$ and compare the output columns. Selecting only the rows that differ in their outputs gives suitable values for p. Let $TEST(G, f)$ be the set of all values p such that $\vec{G}\ p \neq \vec{G_f}\ p$:

$$TESTS(G, f) = \{p : p \in ALL(INPUTS(G)), \vec{G}\ p \neq \vec{G_f}\ p\} \tag{8.1}$$

A tuple p is a *test pattern* for circuit G under fault f if $p \in TESTS(G, f)$. If $TEST(G, f) = \{\}$ then the fault f is not detectable because $G = G_f$. This is usually because the fault occurs in a redundant part of the circuit G. Redundancy is often deliberately built into circuits to overcome problems like hazards, but in turn it causes some faults to be undetectable.

A literal transcription of the above comprehension requires building $ALL(n)$ for both functions yielding an algorithm of complexity $O(2^{n+1})$, where $n = INPUTS(G)$. This is an impractical implementation for all but the smallest circuits. A much more efficient method is required for finding a member of this set.

8.3 The D-Algorithm

8.3.1 Introduction to the D-Algorithm

The D-Algorithm [Roth 66] is a test pattern generation algorithm which takes as input a combinational circuit description and a fault and produces as output a test pattern to expose the fault. This section introduces the D-Algorithm and its associated calculus.

The single stuck-at fault model is employed in this paper to describe the D-Algorithm, although this technique also works with other models. This model represents failure by assuming that *exactly* one wire in the circuit maintains the same logic value, irrespective of what pattern is applied to the primary inputs.

We have shown that a straightforward implementation of the specification of a test pattern requires an algorithm of exponential complexity. The D-Algorithm is introduced with an explanation of how it seeks a test pattern.

The original paper on the D-Algorithm presented many tables involving logic gates that were hand constructed. We show how to compute these tables from first

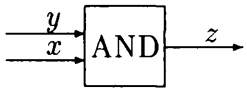


Figure 8.1: An *AND* gate.

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

Table 8.1: *AND* gate behaviour

principles, thus easily extending them to any logic gate.

8.3.2 Singular Cover

It is necessary to describe what input patterns to a gate will give a particular output pattern. A set containing this information is called a *singular cover*. The singular cover for an *AND* gate with inputs x and y and with output z is shown in Table 8.2.

This table contains the following information. To set the output z of the *AND* gate to 1, both the inputs have to be set to 1. The output of this gate may be set to 0 by setting x to 0 and y to any value, or by setting y to 0 and x to any value. The letter X denotes an arbitrary "don't care" value.

This singular cover is really an abbreviated form of the truth table. It is designed to be used in the opposite direction from the truth table i.e. given an output, a set of inputs that produces this output is required. The singular cover is in two parts: one for the patterns that produce a 0 output and the other for the patterns that produce a 1 output (hence the horizontal dividing line in the table).

The expanded form of the singular cover for a gate G which takes as input an

x	y	z
1	1	1
0	X	0
X	0	0

Table 8.2: *AND* gate singular cover.

n-tuple may be represented as the following pair of sets of tuples:

$$COVER(G) = (\{p : p \in ALL(n); \vec{G}\, p = 0\}, \{p : p \in ALL(n); \vec{G}\, p = 1\})$$

The first element is the set of all input patterns that produce output 0 and the second element is the set of all input patterns that produce output 1. To build this cover would require an operation of complexity $O(2^n)$ which is too expensive to compute and store for realistic circuits.

In order to construct the smaller and more convenient singular cover from the cover, the X value must be added to the domain of *logic* values. Later, two other values will be required. Let *logic* now be the set $\{0, 1, X, D, \overline{D}\}$.

The singular cover may be built by the following technique. For each set C in $COVER(G)$ perform the following operation. If two tuples $x, y \in C$ can be found in which there is some position where they differ i.e. $x_i = 0$ and $y_i = 1$ then remove these two tuples from the set and replace them with an element like x or y but with X as its i^{th} element. This procedure is repeated until there are no such differences. This will occur when there is no cubes which differ in only one position to an other cube. Note that an X does not combine with just any member of *logic* to return another X. Only differences between 0 and 1 are considered. At the end of this procedure the set will contain the condensed singular cover. This allows tables of sigular cover to be computed automatically for any combinational component.

8.3.3 D-cubes

Assume that the AND gate in Table 8.2 is suspected of having a fault at line x. This requires the observation of line x which needs fault information to be propagated to an observable output. The first step is to make the output z sensitive to the value on line x. For the fault information on line x to be propagated to line z requires input line y to be set to 1 so that $x \wedge 1 = x = z$ i.e. z assumes the value at x.

By setting $y = 1$ we are ensuring that lines x and z have the same logic value. If the faulty gate was a $NAND$ gate, then by setting $y = 1$, we are ensuring the output z always has the opposite value from from the input x. This relationship between lines is represented symbolically by D and $\overline{D}$. The symbols D and $\overline{D}$ may assume the values 0 or 1. All the Ds in a given pattern have the same value and similarly all the $\overline{D}$s. If both Ds and $\overline{D}$s occur in the pattern, then the elements containing a D have a logic value opposite to those elements containing a $\overline{D}$. The D notation is used to relate wires which always have opposite logic values.

A *D-cube* is a vector or string of values of type *logic*. D-cubes are used to represent fault propagation information by relating the values at various nodes in the circuit. These cubes are called *propagation cubes* because they show how to propagate fault information from one input line to an output line. In circuits with reconvergent fanout, it is possible to have more than one input carrying fault information. This case is dealt with later.

Table 8.3(a) shows 2 D-cubes for an AND gate. The first line explains how to make the output sensitive to the value on line x i.e. set $y = 1$ so $z = x$. The second line explains how to make the output sensitive to the value on line y i.e. set $x = 1$ so $z = y$. Table 8.4 contains the same information for a $NAND$ gate. Here, the value being propagated from an input line to the output line is negated: this relationship is expressed by assigning a D to the input line and a $\overline{D}$ to the output line. Alternatively, $\overline{D}$ could be assigned to the input line and D to the output line: the important fact is that these lines have opposite logic values in a working circuit

Table 8.3(b) shows the propagation D-cubes for an OR gate. If all the fault sensitive values in a singular cover were "flipped" i.e. Ds changed to $\overline{D}$s and vice versa, then the cover would still represent the same partial function. The choice of D in Table 8.3 is arbitrary: $\overline{D}$ could have been used instead.

For a circuit containing n wires, the D-Algorithm uses an n tuple to describe the assignments to these wires. Instead of writing the tuple in the usual $(v_1, \ldots, v_n)$ notation, where each coordinate i represents the assignment at wire 1, an abbreviated form is used. The abbreviation involves omitting some or all of the information about coordinates that have an X value– instead a list of assignments to coordinates is constructed. For example,

$$D^x 1^y D^z$$

represents one of the propagation D-cubes for the AND gate in Figure 8.1. The assignments are superscripted with the coordinate. If a is a cube then a_i is the value at coordinate i. If this is not explicitly mentioned in the cube then the value at wire i is understood to be X. Cubes are essentially an abbreviated form of tuples.

For this AND gate, the cube $D^x 1^y D^z$ means that if $y = 1$ then the value of the output z will always be the same as the value on the input x. For the $NAND$ gate, the cube $D^x 1^y \overline{D}^z$ means that if $y = 1$ then the value of the output z will always be different from the value on the input x.

The D symbols differ from the X symbol because all the Ds in a given cube must have the same value, whereas the Xs are independent of each other. If two

x	y	z
D	1	D
1	D	D

(a)

x	y	z
D	0	D
D	D	D
D	D	D

(b)

Table 8.3: Propagation D-cubes for (a) AND and (b) OR gates.

x	y	z
D	1	$\overline{D}$
1	D	$\overline{D}$

Table 8.4: Primitive D-cubes for a $NAND$ gate.

coordinates i and j of some cube p have been assigned the same fault sensitive value then in a working circuit the logic value present at these two coordinates is always the same. If the circuit has input lines $p_1 \ldots p_n$, then no matter what inputs are applied $p_i = p_j$ is always true in working circuit. A similar situation holds for when two coordinates are both $\overline{D}$.

This implication does not follow if $p_i = p_j = X$. For example, the cube $1^a X^b D^c \overline{D} d$ may represent a member of either $\{1^a 0^b 0^c 1^d, 1^a 1^b 0^c 1^d\}$ (varying X with $D = 0$) or $\{1^a 0^b 1^c 0^d, 1^a 1^b 1^c 0^d\}$ (varying X with $D = 1$). Notice that in each set that $\overline{D}$ always has the opposite value of D.

8.3.4 Embryonic Tests

A primitive cube for a gate contains *embryonic* tests. These are tests only for the lines on the input and output of the gate concerned and not necessarily test patterns to be applied to the primary inputs. The primitive cube $D^x 1^y D^z$ for the AND gate from Table 8.3 contains two embryonic tests. For the first test, let $D = 0$ i.e. the output should be 0 in a working circuit. This is not the case if $x/1$ or $z/1$ so the signal 010 is a test for $x/1$ and $z/1$. The other embryonic test from this cube may be obtained by setting $D = 1$. This yields the test 111 for the faults $x/0$ and $z/0$ by a similar argument. The other primitive cube for the AND gate $1^x D^y D^z$ also yields two tests: if $D = 0$ this gives the signal 100 which tests for $y/1$ and if $D = 1$ then this yields 111 which tests for $y/0$. Thus to test for all stuck it faults, it is necessary to apply the signals 010, 100 and 111.

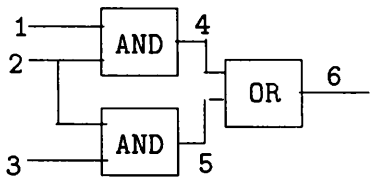


Figure 8.2: Composite circuit

8.3.5 Combining Embryonic Tests

Figure 8.2 on page 150 shows a simple three gate circuit with the singular cover and D-cubes shown in Table 8.5. This table is now used to find a test which will make the output line 6 sensitive to the value at input line 1.

First, input line 1 is assigned the value D represented by D^1. Let this information be stored in a *testcube* i.e. $tc^0 = D^1$. Required is a cube which has a D or $\overline{D}$ value in coordinates 1 and 4 i.e. line 4 is to be made sensitive to line 1. If line 4 can be sensitised then the process can be repeated to find a sensitive path past the gates that have line 4 as an input. If line 4 cannot be sensitised then there are no other possible paths to the output and test pattern generation fails.

Examining the D-cubes for an AND only one cube satisfies these requirements: $D^1 1^2 D^4$. Alternatively, using the table of D-cubes (table 8.5), we see only one column which has a D value in rows 1 and 4. This cube contains a test for line 1 in terms of line 4. If line 1 is stuck-at-0 then setting $D = 1$ gives the test 111. If line 1 is stuck-at-1 then setting $D = 0$ gives the test 010. Now, tc^0 has been combined with $D^1 1^2 D^4$ to yield another test cube $tc^1 = D^1 1^2 D^4$.

Line 4 now carries fault information which has to be propagated to the output line 6. In order to do this, a D-cube which has D values in coordinates 4 and 6 is required. There are 2 such values: $D^4 0^5 D^6$ and $D^4 D^5 D^6$. Choosing the first cube and combining it with tc^1 yields another test cube $tc^2 = D^1 1^2 D^4 0^5 D^6$.

The above process may be represented symbolically as follows. When two cubes are "combined" what is actually happening is an intersection process which checks that no coordinate in the two cubes has an inconsistent value. For example, if one cube contained the value 1^4 and another contained the value 0^4 then these two

cubes may not be combined or intersected. This is because the first *requires* line 4 to be set to 0 and the second also *requires* line 4 to be set 1 in order to propagate fault information. Since this condition cannot be meet, no fault information passes through the gate concerned.

Here is a symbolic representation of the example given above:

$$tc^0 = D^1$$
$$tc^1 = tc^0 \cap D^1 1^2 D^4 = D^1 \cap D^1 1^2 D^4 = D^1 1^2 D^4$$
$$tc^2 = tc^1 \cap D^4 0^5 D^6 = D^1 1^2 D^4 \cap D^4 0^5 D^6 = D^1 1^2 D^4 0^5 D^6$$

The cubes tc^0, tc^1, tc^2 form a connected chain from a primary input to a primary output.

The cube tc^2 contains possible tests for lines 1 *and* 4 in terms of line 6. This cube has the following interpretation. If line 2 has signal 1 and line 5 has signal 0 then lines 1, 4 and 6 will have the same value. This cube does not say anything about how (or if) lines 2 and 5 may be set to these values.

Most cubes require a constant value to be applied to the input line of the gate propagating fault information. The intersections above have not taken account of how (or even if) these nodes can be set appropriately. This requires another step called the *consistency operation* or *justification* to be applied to each possible test pattern produced by the intersection process. Working from a sensitive output, the singular cover is used to justify the fixed logic node assignments made by the forward intersection process. If a primary output is reached, then the current possible test is a proper test which detects the fault concerned when it along with any other input assignments required by the justification process are applied to the appropriate primary inputs. Otherwise, this possible test is discarded.

8.3.6 Intersecting Cubes

The intersection of cubes in general is now considered. Let a and b be cubes and define the intersection of these cubes as follows. If for some coordinate i, $a_i = 1$ and $b_i = 0$ or $a_i = 0$ and $b_i = 1$ then define the result of the intersection to be the *empty cube* ϕ. Such an intersection is called a ϕ-intersection. This happens when the cubes being combined are trying to assign opposite fixed logic values to line i i.e. contain inconsistent information. If there are no ϕ-intersections then use the

1	2	3	4	5	6	Notes
1	1		1			singular cover for AND_1
0			0			
	0		0			
	0			0		singular cover for AND_2
	1	1		1		
		0		0		
			1		1	singular cover for OR_1
				1	1	
			0	0	0	
D	1		D			D-cubes for AND_1
1	D		D			
	D	1		D		D-cubes for AND_2
	1	D		D		
D			D	0	D	D-cubes for OR_1
			0	D	D	
			D	D	D	

Table 8.5: Singular Cover and primitive D-Cube

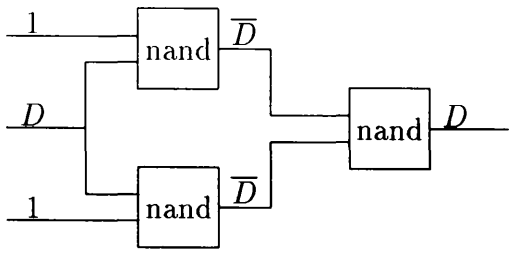


Figure 8.3: Reconvergent Fanout

following rules:

$$0 \cap 0 = 0 \cap X = X \cap 0 = 0$$
$$1 \cap 1 = 1 = 1 \cap X = X \cap 1 = 1$$
$$X \cap X = X$$
$$X \cap D = D \cap X = D$$
$$X \cap \overline{D} = \overline{D} \cap X = \overline{D}$$
$$D \cap D = \mu$$
$$\overline{D} \cap \overline{D} = \mu$$
$$D \cap \overline{D} = \overline{D} \cap D = \lambda$$

The first two lines deal with the case where a fixed value is combined with an uncommited value X. The fixed value simply dominates the uncommited value. The third line deals with the trivial case where two uncomitted values are combined to give an uncomitted value. The fourth and fifth lines show that D and $\overline{D}$ values also dominate X values.

Intersections between fault sensitive values (D and $\overline{D}$) occur in circuits with reconvergent fanout. An example of such a circuit is shown in Figure 8.3. An intersection between two fault sensitive values of the same polarity (i.e. both D or both $\overline{D}$) is called a μ-intersection. Intersections between fault sensitive values of opposite polarity are called λ-intersections.

If μ and λ intersections are generated between the components of the cubes, then the intersection of these cubes is not defined. This corresponds to one cube saying that lines i and j must always have the same value whilst the other cube is saying that i and j must always have opposite values. It is impossible to satisfy both of these assertions simultaneously. An example of such an intersection is:

$$D^1 1^2 \overline{D}^3 \cap D^1 1^2 D^3 = \lambda^1 1^2 \mu^3$$

Since both λ and μ appear in the result, this intersection is not defined. If no λ intersections take place then it is safe to use the following rules for intersecting cubes

a and b:

$$D \cap D = D$$
$$\overline{D} \cap \overline{D} = \overline{D}$$
$$\nexists i.a_i \cap b_i = \lambda$$

The third line ensures that there are no λ terms arising from the combination of the two cubes. This corresponds to the case where for each coordinate i in one cube which is D or $\overline{D}$, the corresponding coordinate in the other cube is either X or the same. An example of such an intersection is:

$$D^1 1^2 \overline{D}^3 \cap D^1 X^2 \overline{D}^3 = D^1 1^2 \overline{D}^3$$

If no λ intersections take place then the μ intersections may be resolved by flipping all the D and $\overline{D}$ values in the second cube and then reintersecting the result with the first cube. This will effectively transform the μ intersections to λ intersections. An example of this case is:

$$D^1 1^2 \overline{D}^3 \cap \overline{D}^1 1^2 D^3 = D^1 1^2 \overline{D}^3 \cap D^1 1^2 \overline{D}^3 = D^1 1^2 \overline{D}^3$$

This above intersections are sensible because both cubes are conveying consistent information: the first says that line 1 and line 3 always have the same value by assigning the same value to both lines (D in this case). The second also states that lines 1 and 3 always have the same value (this time by using the symbol $\overline{D}$). By flipping the D and $\overline{D}$ values in the second cube, we do not alter its information content, but it does now becomes consistent with the first cube. For this method to work it must be established that no λ intersections take place, since flipping a λ result at b_i makes it immediately inconsistent with the value at a_i.

Intersections of cubes are formed until a primary output is reached or the intersection yields an empty cube. If a primary output has been reached then a possible sensitive path has been found. The forward propagation phase (called *D-drive*) makes assumptions about the input values of gates that propagate the fault information. For example, tc^2 above assumes that line 5 can be set to 0 in order to propagate the fault sensitive value at line 4 past the OR gate to line 6. All of these assumptions have to be justified by backward simulation. This part of the D-Algorithm is called the *consistency* phase. It may not be possible to set a particular line to the desired value, or the assumptions made may be mutually inconsistent.

If the result of intersecting two cubes fails, then the algorithm backtracks to the last fork and tries to find a different sensitive path. If there are no remaining paths then the fault under consideration is not detectable.

8.3.7 The Consistency Phase

The assumptions made in the D-drive phase of the algorithm are justified in the consistency phase of the algorithm. The singular covers are used to find a connected chain from a fault sensitive output to the primary inputs. This is done by a backward simulation of the circuit. Again, there may be several possible paths from a sensitive primary output to the primary inputs, and only some (or none) will be paths that are consistent with the assignments made in the D-drive phase. The algorithm uses a backtracking procedure to find a possible path.

In the example circuit there is only one possible sensitive path found by the D-drive phase, namely: $tc^2 = D^1 1^2 D^4 0^5 D^6$. This makes the assumption that line 5 can be set to 0. This assumption is checked by the consistency phase by using the singular cover shown in Table 8.5. Required is a cube that gives information about what the inputs to the bottom AND gate must be to secure a value of 0 at the output (line 5) of the gate.

There are three cubes that have line 5 set to zero:

$$X^3 0^2 0^5$$
$$X^2 0^3 0^5$$
$$0^4 0^5 0^6$$

The last one contains no information about the input lines of the bottom AND gate i.e. lines 2 and 3, so it is discarded. The first cube is checked to see if it contains information consistent with the assignments made during the D-drive process by intersecting it with tc^2:

$$X^3 0^2 0^5 \cap D^1 1^2 D^4 0^5 D^6 = D1\phi 2 D^4 0^5 D^6 = \phi$$

This intersection is empty i.e. inconsistent, so the cube is $X^1 0^2 0^5$ discarded. Intersecting the second cube yields:

$$X^2 0^3 0^5 \cap D^1 1^2 D^4 0^5 D^6 = D^1 1^2 0^3 D^4 0^5 D^6$$

Thus, $D^1 1^2 0^3 D^4 0^5 D^6$ is a complete test cube for stuck-at-0 and stuck-at-1 faults at line 1. To test for line 1 stuck at 0, the pattern $1^1 1^2 0^3$ is used: in a working circuit the output should be 1 (the cube states that in a working circuit lines 1 and 6 always have the same value if line 2 is 1 and line 3 is 0). Similarly, to test for line 1 stuck at 1, the pattern $0^1 1^2 0^3$ is used: if the output is 0 the circuit is working.

8.3.8 Possible Adaptions to the D-Algorithm

The D-Algorithm stops when it finds one possible connected D-chain from the site of the fault to the primary outputs and one possible path that justifies the assignments made in the D-drive phase. There may be more than one possible path or set of assignments that sensitise the site of the fault.

The algorithm has already been modified by others to take into account several heuristics. For example, during the backwards phase when there is more than one possible route to the primary inputs, the algorithm tries the shortest paths first.

One a test pattern has been generated, it may be run through the deductive fault simulator interpretation to discover all the other faults that this pattern exposes.

Often, more than one pattern will expose a given fault. However, some of these patterns will expose more additional faults than others. It is desirable to find the pattern which exposes the most faults. This can be accomplished by adapting the D-Algorithm to try all possible paths in order to construct all the possible test patterns for a given fault. Then, each of these patterns is passed through a deductive fault simulator to discover which is the "best" pattern. This will usually lead to a smaller set of test patterns for a complete circuit, but is a very expensive operation to perform. However, for a production design, testing time is a very important factor in the cost of each unit, so the extra one-off analysis time may be worthwhile if the testing time is reduced by applying a smaller set of test patterns.

8.4 Re-expressing D-Intersection

The intersection process is the most important part of the D-Algorithm. Various notations have been employed by Roth to describe D-intersection ($\cap$ in the original paper [Roth 66] and I since then [Roth 80]).

The notation $a \cap b$ was used to appeal to the notion that the result of this operation somehow contains all of the information in cubes a and b and nothing else, or is undefined if the information in a is inconsistent with the information in b. Furthermore, some cubes contain more information than others. Roth says that cube a D-contains another cube b if it is possible to obtain cube b from a by replacing the X values in a by suitable fixed logic values. This relationship is also expressed in set notation as $a \supset c$ in the original literature. This notation was later replaced by the notion of *faces* and *cofaces*.

It is possible to define a partial order over cubes a and b using the above con-

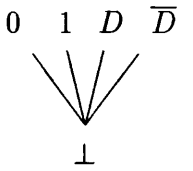


Figure 8.4: Partial order for assignment values.

tainment property. By constructing a suitable domain, the intersection of two cubes corresponds to finding the least upper bound of the two cubes $a \sqcup b$.

This notation is much more natural than that used by Roth. Firstly, the analogy with the use of $\sqcup$ and $\sqsubseteq$ in domain theory is strong.

In domain theory, $a \sqcup b$ denotes the value, if it exists, which contains all the information in a and all the information in b *and nothing else.* Such a value does not always exists if a and b are inconsistent with each other. However, least upper bounds of inconsistent objects do not exist, whereas the union of a set of two objects returns a set with inconsistent elements. The use of least upper bound seems to be a much more apt notation which captures more accurately what happens when two cubes are intersected.

In domain theory, the expression $a \sqsubseteq b$ means that b contains at least as much information as a. This is a similar concept to one cube "containing" another. The symbol X will now be replaced by $\perp$ to signify that there is no information about what value is assigned to a wire. For example, $0^2D^3\perp^5 \sqsubseteq 0^2D^31^5$ may be constructed by defining a partial order over the individual elements of the cubes.

Consider the following partial order for the assignment values:

$$\perp \sqsubseteq 0$$
$$\perp \sqsubseteq 1$$
$$\perp \sqsubseteq D$$
$$\perp \sqsubseteq \overline{D}$$

This is like the **Bool** domain with two fault sensitive elements added. This partial order is shown pictorially in the lattice in Figure 8.4.

The partial order over cubes may be naturally defined as follows. For any two n input cubes a and b:

$$a \sqsubseteq b \;=_{def}\; c \text{ where } \forall i : 0 \leq i \leq n : \; c_i = a_i \sqcup b_i$$

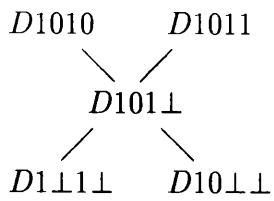


Figure 8.5: Part of a partial order for cubes.

If the least upper bound of any of the elements does not exist, then the least upper bound of the two cubes does not exist either.

Part of the lattice containing the cubes $D1\perp1\perp$ and $D10\perp\perp$ is shown in Figure 8.5. The coordinates for each assignment have been omitted: the cubes are to be combined pairwise. All the cubes above $D1\perp1\perp$ and $D10\perp\perp$ are consistent. It is possible to obtain $D1010$ from $D1\perp1\perp$ be replacing both $\perp$ values by 0. However, only one cube ,$D101\perp$, contains the information in the cubes $D1\perp1\perp$ and $D10\perp\perp$. The other two cubes contain information about the value of the fifth element which was not present in the original cubes. In the lattice, these two cubes are upper bounds for the bottom cubes, but $D101\perp$ is the *least* upper bound.

$$D1\perp1\perp \sqcup D10\perp\perp = D101\perp$$

If two cubes are not consistent, then the least upper bound does not exist.

Unfortunately, there is a case where there is no least upper bound of two cubes but there are bounds which contain useful values. This occurs when λ intersections take place but no μ intersections occur. Consider the result of combining the following cubes:

$$D^1 1^2 \overline{D}^3 \sqcup \overline{D}^1 \perp^2 D^3$$

The least upper bound does not exist, because $D \sqcup \overline{D}$ does not exist. However, $D^1 1^2 \overline{D}^3$ and $\overline{D}^1 1^2 D^3$ are two sensible results. They are two upper bounds, but neither of them is a least upper bound. This situation is shown in Figure 8.6.

The cubes $D^1 1^2 \overline{D}^3$ and $\overline{D}^1 1^2 D^3$ really represent the same information. By flipping the fault sensitive values of one cube we obtain the other. We are unable to detect this because the symbols D and $\overline{D}$ appear in different contexts in the two cubes. One solution is to generate different names for fault sensitive values for each cube. If $a, b, \ldots$ are names for fault sensitive values, then our domain is still flat i.e. $\perp \sqsubseteq \alpha$ where α is a fixed logic value or a sensitive value and there is no relationship between the other values.

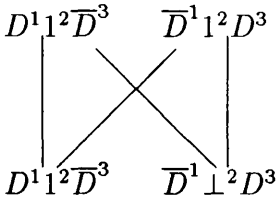


Figure 8.6: No least upper bound.

To express the fact that one sensitive value is always the opposite of another, we introduce a function over this domain for "negation", denoted by an overbar. This function is defined as:

$$\overline{\perp} = \perp$$
$$\overline{0} = 1$$
$$\overline{1} = 0$$

The result of applying this function to a fault sensitive value a is $\overline{a}$. Thus, the two cubes above could be represented by $a^1 1^2 \overline{a}^3$ and $\overline{b}^1 1^2 b^3$. To compute the least upper bound of these cubes, we have to find solutions to $a \sqcup \overline{b}$ *and* $\overline{a} \sqcup b$. The possible values for a and b are drawn from the set 0, 1. Possible substitutions for a and b can be found by solving the equations:

$$a = \overline{b}$$
$$b = \overline{a}$$

The solution set is $\{(a = 1, b = 0), (a = 0, b = 1)\}$. Using the first solution in the set, one upper bound may now be found:

$$a^1 1^2 \overline{a}^3 \sqcup \overline{b}^1 1^2 b^3 = (a \sqcup \overline{b})^1 1^2 (b \sqcup \overline{a})^3$$
$$a \sqcup \overline{b} = 1 \sqcup 1 = a$$
$$b \sqcup \overline{a} = 0 \sqcup 0 = \overline{a}$$
$$a^1 1^2 \overline{a}^3 \sqcup \overline{b}^1 1^2 b^3 = a^1 1^2 \overline{a}^3$$

The other solutions may be constructed in a similar manner.

This section has described a structure that can be used to descirbe fault sensistive information. The structure allows the rules for combining fault sensistive information to be cast using well known mathematical notation. This results in a better understanding of the rules and gives a clearer idea ofhow these operations can be implemented.

8.5 Implementing the D-Algorithm

8.5.1 Introduction

The D-Algorithm is implemented in three stages. Each stage is coded as a non-standard interpretation and the stages are combined to produce the complete D-Algorithm. The first stage computes a list of possible test patterns that might expose the given fault. The next stage checks if each of these patterns is capable of driving fault information to an observable output, making certain assumptions on the way. For each successful pattern, the consistency phase checks to see if the assumptions made during the previous stage can be satisfied, yielding true test patterns.

To demonstrate the implementation, we shall use the circuit shown in figure 8.2. The Ruby description for this is submitted to our system in the file '**ch8.ruby**':

```
Chapter 8 test circuit.

> testcir = [id, [split, id]] ; reorg ; [and, and] ; or ;;
> reorg = [id, lsh] ; rsh ;;
```

The output below shows how the circuit is simulated and annotated. The test file simulates the circuit for input $< L, < H, L >>$. It then labels the circuits, followed by an annotated circuit graph.

```
> IMPORT prelude ;;
> IMPORT ch8 ;;

> STANDARD <L, <H, L>> testcir ;;
> LABEL <L, <L, H>> testcir ;;
> ANNOTATE <L, <H, L>> testcir ;;
```

The output produced is:

```
   1) Standard: {<L,<H,L>>} testcir  {L}
   2) LabelSyn: {<L,<L,H>>} testcir
         [id, [fork 2, id]]; [id, id \V/ id]; id <-> id; [and#1, and#2]; or#3
   3) Annotate: {<L,<H,L>>} testcir
         [id, [fork 2, id]]; [id, id \V/ id]; id <-> id; [and#1 <{L}>,
and#2 <{L}>]; or#3 <{L}>
```

```
Parse OK.
Results logged in ch8.log
```

8.5.2 Sensitising the Faulty Node

The first step of the D-Algorithm involves finding an input combination to set the faulty node to a sensitive value. For example, if node i is stuck-at-0, then an input combination that sets this node to 1 is sought. This has been implemented as a simple backward interpretation. We start off with a circuit which has every internal node set to X except the faulty node, which is set to the opposite value to what it is stuck-at. For example, to test for node 6 stuck-at-0, node 6 is assigned the value 1.

Setting the value of a node to a particular value is an operation which can be implemented simply as a non-standard interpretation in either direction. For simplicity, we consider a circuit graph which has already been decorated by labels which we will parse in the forward direction. The method involves propagating a pair. The first element contains a function which applied to a node number will return true or false depending on whether this node has to be updated or not. The second element contains the update value. To initialise all the nodes to X we supply the constant function which takes any integer and returns true.

$$labelall : int \rightarrow bool$$
$$labelalln = true$$

To label the entire circuit graph with X values, we propagate the tuple $< labelall, X >$. The interpretation which does this is called SETALLX. To label a particular node, we use the following function:

$$labelnode : int \rightarrow int \rightarrow bool$$
$$labelnodenm = n = m$$

So, to label node 13 with the value $\overline{D}$ we would propagate the tuple $< labelnode13, \overline{D} >$. The definition of each processing node simply applies the function in the first element of the pair to the node number and then updates the value at its node when neccessary. The interpretation that performs this task is called $\mathcal{SETNODE}$, of which $\mathcal{SETALLX}$ is a special case.

The next step is to perform a backward simulation to discover what input patterns will sensitise the fault site. We cannot simply re-use the standard interpretation, since that is only defined over two-valued logic. A new interpretation is needed

for three-valued logic (X, 1 and 0). All output nodes are assigned X: the backward simulation will sweep towards the inputs until it encounters the faulty node. At this node, the output net will have the value X, but the value at the node will be 1 or 0. The backward simulation continues from this node, but this time the output of the node is taken from the value *at* the node, rather than the X at the output net. Shown below is the backward definition for an AND gate. This constructor `Set` is used to build a set from a list. The third line considers the case where the output of the AND gate is `H` and there are no previous assignments to this node (i.e. the set at the node is `Set [X]`). In this case, there is only one possible assignment to the inputs to produce a `H` at the output. This is the singleton set $\{< H, H >\}$ which is returned as the result of the function in this case. The next line considers the case where there is a `L` at the output and returns the three possible input patterns that can produce this value. The other lines generalise these definitions over X values.

```
> and_sen (Set [H]) v = and_sen (Set [X]) H
> and_sen (Set [L]) v = and_sen (Set [X]) L
> and_sen (Set [X]) H = Set [Tuple [H, H]]
> and_sen (Set [X]) L = Set [Tuple [L, L], Tuple [L, H], Tuple [H, L]]
> and_sen (Set [X]) X = Set [Tuple [H, H]]
```

This function is lifted to operate over sets of any size: the definition shown is defined just for singleton sets. The interpretation which implements the sensitisation operation for the whole circuit description is called $\mathcal{SEN}$. This relies on the circuit being appropriately decorated: we name the interpretation that performs the decorations and the backwards analysis $\mathcal{SENCIR}$ and define it as:

$$\mathcal{SENCIR} = (\mathcal{SETALLX}\{\}); (\mathcal{SETNODE}\{\}); (\mathcal{SEN}\lfloor)$$

To execute this interpretation, abritary inputs are given for the first stage when the entire circuit graph is labelled with X values. For the second stage, we supply the appropriate pair containing the node to be sensitised and the value that node is to be set to. The final stage is run backwards with X values at each output node.

8.5.3 The D-Drive Phase

The D-drive phase may be implemented as a simple forward interpretation. The information flow appears to be complicated by backtracking, which seems to call for a forward interpretation that oscillates. However, instead of propagating forward information about only one path, information about *all* possible paths is propagated.

We shall consider forks with a fan-out of 2. The analysis extends easily to forks of a greater size. At each fork in the circuit which has a fault sensitive value at its input, the two outputs are defined as follows. Instead of using the standard behaviour of fork, which would make both outputs equal to the input, each output is a list of two values. One prong of the fork will propagate the list $[D, \perp]$ whilst the other prong will propagate the list $[\perp, D]$. This has the effect of trying three possible paths:

1. Try to propagate the fault sensitive value only through the first fork, making no assignments to the second.

2. Try to propagate the fault sensitive value through the second fork, making no assignments to the first.

3. Try to propagate the fault sensitive value though both forks simultaneously.

Sometimes it is necessary to simultaneously form a double D-chain (corresponding to *two* sensitive paths) to cope with reconvergent fanout (hence the rules for $D \sqcup D$ etc.).

Nodes are defined naturally to propagate a list of possible propagation cubes. However, lazy evaluation will ensure that only those elements required to produce a test vector are actually evaluated [Wadler 85]. Each two input gate takes along each input a list of possible assignments. The output is the cartesian product of the two lists, with ϕ intersections removed.

For an AND gate with input lines x and y, the D-drive rule for propagating a D value along the x input is expressed as follows in the non-standard interpretation system. Angle brackets are used to form tuples. an asterisk denotes a non-standard operation and the lower case identifiers describe fault sensitive information. The non-standard values are D-cubes.

$$< tc_a, tc_b > AND^* \, tc_c \Leftrightarrow tc_c = tc_a \sqcup tc_b 1^y \quad \text{if} \quad D^x \sqsubseteq tc_a \wedge \perp^y \sqsubseteq tc_b$$

The propagation requires line y to be 1 in order to propagate the D at line x to the output of the or gate: this is specified by adding the cube 1^y to tc_b. The other rules are expressed in a similar manner. The above definition is extended to allow sets of cubes to be the non-standard values, and the cartesian product between the sets is formed using the above rule.

$$< S_1, S_2 > AND \, S_3 \Leftrightarrow S_3 = \{tc_3 : \, tc_1 \in S_1, tc_2 \in S_2, < tc_1, tc_2 > AND^* \, tc_3\}$$

Here, S_i are the sets of possible test cubes. This this coded in a stright forward manner by defining the ordering relation and then using the usual definitions of least upper bounds and set comprehensions. The interpretation is named $\mathcal{DRIVE}$.

8.5.4 Implementation of the Consistency Operation

The consistency operation involves checking that the assignments produced by the D-drive phase can actually be made by manipulating primary inputs. It is very similar to backward simulation. A backward logic simulator has already been implemented by using a backward non-standard interpretation. The output of the circuit is specified (either completely or partly) and all the possible input patterns that produce the given output are returned.

For each possible test cube tc, a path is sought back to the primary inputs using the singular cover. The algorithm steps back a gate at a time: for each gate the least upper bound for every cube c in that gates singular cover $SCOVER$ is found, if it exists. For example, if the output of the gate described by $SCOVER$ is required to be 1, then:

$$tc' = \{c : c \in snd\, SCOVER,\ \text{if}\, c \sqcup tc\, \text{exists}\}$$

The second element of $SCOVER$, which contains the set of cubes that assign 1 to the output is accessed by the *snd* function which returns the second element of a tuple. This gives a possible set of test cubes, each of which are extended backwards using the above process until a primary input is reached. If the final set is empty, then it is not possible to justify any of the possible test cubes generated by the D-drive process so no test pattern is generated for the given fault.

The backwards interpreter is implemented by performing a 5-valued backward simulation, rather like the sensitisation interpretation. However, at each stage, the set of values stored at each node, which represent assignments made to that node, are checked against propagated values. If inconsistent values are encountered i.e. when a node has been previously required to hold a 0 but an attempt is made to set its output net to 1, then an empty set is propagated towards the inputs.

If the result of this phase gives an empty set for a particular assignment, then this assignment is not propagated any further by the D-drive process. Let the interpretation for this phase be called $\mathcal{CONSIS}$. The definition for the AND gate in the consistency interpretation is given below, along with the function used to check for conflicts. The `conflict` functions just checks to see if any incompatible assignments have been made to the same node e.g. L and H, or D and $\overline{D}$.

```
> and_consis (Set assignments) v
>  = Set [], if conflicts assignments v
>  = and_sen consis v, otherwise

 > conflicts a L = True, if member a H
> conflicts a H = True, if member a L
> conflicts a v = True, if (member a L) & (member a H)
> conflicts a v = True, if (member (v:a) D) & (member (v:a) Dbar)
```

8.5.5 The complete algorithm

The complete D-Algorithm is then described by:

$$DALG = SENNODE; DRIVE\ f; CONSIS\ b$$

The result is a graph, annotated at each node with sets of D-cubes. Not all these D-cubes correspond to useful tests: only those cubes that make it to an observable output can be used to determine test patterns.

To perform a D-Algorithm test pattern generation on our test circuit, we have to place some dummy components at the inputs to allow us to talk about the primary inputs. This is becuase net values are derived from the node that drives the net. Primary inputs of circuits in isolation are not driven by any nodes. We use the special component `inpad` for this purpose. It is like the identity relation over single wires, except that it consumes a label during the labelling interpretation. The test script for our circuit is shown below. Instead of setting a node to a fixed value we have chosen to use D which should return a list of test cubes at the result. In this case, we expect only a singleton set since we know in advance that there is only one satisfactory test cube.

```
IMPORT prelude ;
IMPORT ch8 ;
IMPORT pads ;

DALG <3, D> [inpad, [inpad, inpad]] ; testcir ;;
```

The output produced is shown below. Unfortnately, the nodes are labelled differently from the figure, but the cube produced is correct.

```
1) DALG:  <3, D>  [inpad, [inpad, inpad]] ; testcir  {D3H2D5L4D6}
```

The implementation is unfortunately slow and wasteful of space. An imperative implementation can use a smaller number of variables to represent cubes and use explicit backtracking to update these cubes as the analysis proceeds.

The current implementation could be improved by adding an extra parameter to the interpretation function to hold environment information, thus avoiding the expensive task of labelling *every* node with a set of cubes. For a large circuit, an environment look-up could be an expensive operation.

However, it is still useful that such a complex algorithm can be broken down into modular chunks which can be implemented independently.

8.6 Verification using the D-Algorithm

The D-Algorithm may be modified to compare two designs for equivalent behaviour. First, an m output design is broken down into m single output designs.

Let A and B be single output circuits which are to be tested for equivalence. The output of A is assigned value 1 and the output of B is assigned the value 0. Backward simulation in the style of the consistency operation is used to find what input values produce the given output value. If there is any intersection between the results of the two backward simulation (i.e. if the least upper bound of the cubes returned by them exists) then a counterexample has been found that states A and B cannot be equivalent.

This counterexample is a pattern which sets A to 1 and B to 0, but for A and B to be equivalent they must produce the same output fro the same input.

In general, this analysis time grows exponentially with the size of the circuit. However, for circuits that are nearly identical, the running time grows in a more linear manner.

8.7 Extending the D-Algorithm

One of the most popular recent techniques for generating test patterns is PODEM: path orientated decision making [Goel 81]. The D-Algorithm makes aribitrary choices about which paths to follow at forks and in which order to backtrack. PODEM uses information generated by other testability analysis tools to find paths which produce a result, positive or negative, quickly. For this reason the PODEM algorithm usually performs much better than the D-Algorithm. However, it still be-

longs to the same class of test pattern generation techniques which exploit sensitive paths.

Consider the task of setting the output of a three input *OR* gate to be 1. This can be done by attempting to set any of the three inputs to 1. Instead of choosing one randomly, PODEM annotates each wire by its controllability value. This could be done using the SCOAP testability measure scheme. Then, it makes sense to try to set to 1 the input which is the easiest to control i.e. the node with the best controllability value. Using SCOAP, this would correspond to choosing the node with the lowest rating.

Now consider the task of setting the output of a three input *OR* gate to 0. This requires all the inputs to be set to 0. Here, it makes sense to try setting the node with the poorest controllability to 0. If this node cannot be set to 0, then there is no point in trying any of the other nodes. Thus, PODEM does not try to simultaneously seek paths back from each node. Instead, it tried the hardest node first, and proceeds to the other nodes only if the hardest node can be set to the required value.

This technique speeds up test pattern generation because a solution to each justification step is found quickly. A further improvement is made if a fault simulator is used after each test pattern is generated. The simulator will expose other faults covered by the automatically generated test pattern. These faults can be removed from the fault list, thus reducing the number of faults that have to be exposed by the expensive automatic test pattern generation program.

It is interesting to note that we can construct a PODEM style interpreter by combining three existing interpretations: SCOAP testability measure, deductuve fault simulation and the D-Algorithm.

The outputs from the above procedure are then fed into the deductive fault simulation interpretation to discover what other faults are exposed by the generated test pattern. The incorporation of a deductive fault simulation into a cycle as described above requires a new interpretation combining form.

The first step would be to simply annotate the circuit with SCOAP testability measures. The non-standard semantics of each node would then be altered to produce sets of result, where each result is orderd using the available testability measures. This has the effect of converting the set into a list, whose earlier elements are the ones most likely to give a result quickest. Using lazy evaluation carefully , we are even less likely to perform unfruitful computations. Our D-algoirthm implemen-

tation only builds as many elements of a list that it absolutely has to. The PODEM implementation always builds a complete list, but the elements are only partially evaluated. Only the testability measures for each element is always know: this is used to sort the list into order, without evaluating the heavyweight expressions to deal with the backtracking.

To demonstrate this idea, consider the following Miranda script:

```
list1 = [(2, undef), (1, undef)]
sort2 [a, b]
  = [a, b], if fst a < fst b
  = [b, a], otherwise

run1 = map fst (sort2 list1)
```

If **undef** is ever evaluated, a run time error occurs. The list `list1` contains pairs, the first element of each pair is defined and the second element is undefined. In a lazy language like Miranda it is possible to sort this list without evaluating the second elements. The function **run1** will definitely cause **sort2** to perform an exchange but it will give the correct result, as shown:

```
Miranda run1
[1,2]
```

Note that we have filtered out the undefined values for output. This is exactly the kind of operations that a non-standard interpretation implemention of PODEM would do to ensure efficient execution.

We have not at present implemented a full PODEM style test pattern generation program. One problem is that by creating more complex interpretations, we are overloading the Miranda system. Our first attempt at realising a PODEM style implementation caused a stack overflow on our Miranda system. We only have access to Miranda running on relatively slow Sun 3 computers with only 4 megabytes of memory. We are currently reimplementing the entire system in Lazy ML. This runs faster and is implemented on more powerful Sun 4 computers with larger 24 megabyte memories. It is hoped that PODEM will run on the new system. Another problem is that PODEM requires backtracking to be considered explicitly, whereas we have so far dealt with it implicitly. However, it still should be possible to elegantly realised a PODEM test pattern generator with respectable run time performance.

8.8 Conclusions

The D-Algorithm has been re-expressed by defining a partial order over the D-cubes. This leads to a clearer understanding of what is meant by D-intersection– the most important part of the D-Algorithm. The new notation helps to give a clearer description of the algorithm and helps to motivate a cleaner implementation.

Non-standard interpretation has been exploited to make the implementation of a complicated algorithm simpler. The method takes advantage of the fact that the circuit analysis has the same shape as the design and that backtracking need not be programmed since this may be handled by generating all possible results and using lazy evaluation to ensure no loss of efficiency.

This interpretation, coupled with others already implemented, forms a small but powerful prototype circuit analysis system. The size is kept to a minimum by factoring out the common part of these analyses so that they do not need to be respecified. This helps to maintain the system and makes the very important job of verifying the software easier.

The backwards consistency phase of this technique has also been used to help verify circuits. However, in general, the running time of this analysis is exponential compared to the size of the circuit. This method is particularly useful when two nearly identical circuits are compared, where the running time rises more linearly. This is a very common case as one completed design is modified to incorporate some "engineering change". However, simulation as a means of verification in the general case is not a realistic proposition. Showing that two designs are equivalent has attracted much attention from those employing formal methods, with a reasonable degree of success.

The D-Algorithm is an old one, and does not compare favourably with more recent techniques. However, it is possible to use the D-Algorithm in conjunction with other testability analysis tools to construct a faster analysis tool. The interpretations for deductive fault simulation, SCOAP testability measure and the D-Algorithm can be combined to make a PODEM-style interpretation. This gives an increase in speed for a very small amount of work since we can easily compose interpretations.

The prototypes produced are very slow and require a large amount of heap space. An optimisation to the standard interpretation would immediately benefit all other interpretations. This has not been attempted, since we want to contain the complexity of our system to make future changes easier. Another approach might be to improve the performance of individual interpretations. However, the operation

of a particular interpretation is still constrained by the underlying interpretation model.

Chapter 9

Circuit Layout

9.1 Introduction

This chapter presents another application of non-standard interpretation. We show that circuit layout can be accomplished using non-standard interpretation [Singh 91]. We use this technique to lay out butterfly circuits which are described in Ruby.

The result is that very complex circuit layouts can be automatically generated from the standard behavioural Ruby description. The extra code required to realise the drawing alternative semantics is very small indeed. Several colour prints of butterfly circuits produced by our interpretation system are presented.

9.2 Functional Geometry

Functional geometry involves using *functions* to represent drawings. New drawings can be made from existing drawing (functions) by combining them using higher order functions. Henderson has shown that the famous Escher fish picture can be formed by combining just four tiles using appropriate higher functions [Henderson 82]. Circuit layout using functional geometry has also been attempted successfully by others e.g. [Sheeran 83].

In this chapter, we combine the principles of non-standard interpretation and functional geometry to produce complex circuit layouts. We take advantage of the fact that our implementation is in a polymorphic high-order language to provide a simple but powerful set of functional geometry primitives.

These primitives are used to provide non-standard semantics for certain Ruby descriptions. We constrain ourselves to the study of butterfly networks which are characterised by a high wiring to processing area ratio. The recursive decompositions found by [Sheeran 89] are used to help draw butterfly and butterfly-related networks automatically.

The problem of drawing any Ruby description is a much harder one, in particular four-sided tiles. This is because four sided tiles are implemented in terms of two sided tiles. Work has been done by the author and others to draw Ruby descriptions containing only two sided tiles. This is quite straightforward, the major complication being the **loop** construct.

9.3 Describing Butterflies

The wiring relations trn, zip, halve and pair (introduced in chapter 2) are used to build some of the new wiring patterns which we need to describe butterflies.

We shall not try to represent the structure of tuples in our drawings. This could be done by varying the spacing between wires to reflect how the tuple is constructed. This means that by looking at a picture, we cannot tell the difference between twelve wires (12) and 12 ; halve. Similarly, pair will not affect how wires are drawn. However, both of these relations are still needed to keep the type right of the information travelling along wires.

One much used wiring pattern in butterfly networks is *riffle*, otherwise known as the 'perfect shuffle'. Riffling involves halving a bus of wires, then transposing the resulting tuple, followed by unpairing. The definition is:

$$\text{riffle} \equiv \text{halve} \; ; \; \text{trn} \; ; \; \text{pair}^{-1}$$

We shall also make use of the inverse of riffle i.e. riffle^{-1}.

A relation R is homogeneous if it relates only signals of the same length. A larger homogeneous circuit can be made by making two copies of a smaller one. One such combining form is **two** defined as:

$$\textbf{two}\ R \equiv [R, R] \setminus \text{halve}^{-1}$$

Another useful operation is the interleaving of two copies of R:

ilv R ≡ (**two** R) \ riffle

There are many useful laws about these combining forms, e.g:

two (R ; S) = **two** R ; **two** S
two ilv R = **ilv two** R
ilv (R ; S) = **ilv** R ; **ilv** S

These laws are easily proved in the usual manner.

Four recursive descriptions of butterfly networks found by Sheeran are:

Ψ_{n+1} R = **ilv** Ψ_n R ; **two**$^{n+1}$ R (A)
= Ψ_n **ilv** R ; **two**$^{n+1}$ R (B)
= **ilv**$^{n+1}$ R ; Ψ_n **two** R (C)
= **ilv**$^{n+1}$ R ; **two** Ψ_n R (D)

where the base case is:

Ψ_0 R = R

If R is a comparator (or sorter) then Ψ R is Batcher's bitonic merger.

It is possible to directly transcribe the above definitions to our implementation of Ruby. The source text for the butterfly definitions is given below.

```
> riffle = halve ; zip ; pair% ;;
> two R = [R, R] \ halve% ;;
> ilv R = (two R) \ riffle ;;

> butfy1 0 R = R ;;
> butfy1 n R = ilv (butfy1 (n-1) R) ; two**n R ;;

> butfy2 0 R = R ;;
> butfy2 n R = butfy2 (n-1) (ilv R) ; two**n R ;;

> butfy3 0 R = R ;;
> butfy3 n R = ilv** n R ; butfy3 (n-1) (two R) ;;

> butfy4 0 R = R ;;
> butfy4 n R = ilv**n R ; two (butfy4 (n-1) R) ;;
```

A sample execution of one of these butterflies is shown below. The component R is represented by a two-input comparator (sorter) called comp2. The execution file contains:

```
> IMPORT butfy ;;
> IMPORT prelude ;;

> STANDARD <L, H> butfy1 0 or ;;
> STANDARD <L,L,L,H> butfy1 1 (or ; split) ;;

> STANDARD <8,3> comp2 ;;
> STANDARD <4,7,9,5> butfy1 1 comp2 ;;
> STANDARD <1,2,3,4, 8,7,6,5> butfy2 2 comp2 ;;
> STANDARD <16,15,14,13,12,11,10,9, 1,2,3,4,5,6,7,8> butfy3 3
comp2 ;;
```

The output produced is:

```
1) Standard: <L,H> butfy1 0 or H
2) Standard: <L,L,L,H> butfy1 1 (or; split ) <H,H,H,H>
3) Standard: <8,3> comp2 <3,8>
4) Standard: <4,7,9,5> butfy1 1 comp2 <4,5,7,9>
5) Standard: <1,2,3,4,8,7,6,5> butfy2 2 comp2 <1,2,3,4,5,6,7,8>
6) Standard: <16,15,14,13,12,11,10,9,1,2,3,4,5,6,7,8> butfy3 3
comp2 <1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16>
```

9.4 Drawing Butterflies

Butterfly circuits are now laid out using non-standard interpretation. The non-standard values that flow along wires will be tuples containing coordinates of the bottom left hand corner, colour information and an accumulator value which gathers together all the line drawing commands needed to produce the drawing on some graphical output device. The non-standard definitions for the nodes use their inputs to work out where to draw themselves and in what colour.

With the exception of forks, all of the non-standard interpretations have not over-ridden the meaning of the basic wiring primitives. However, to draw circuits we have to redefine the meaning of the wiring relations too. For example, the identity relation now has to add extra graphical commands to those that it has received through its input to produce a horizontal line. It must also update the x-coordinate of the bottom left hand corner.

We shall be dealing with only one processing node called R which is a 2 to 2 relation. This shall be drawn as a square tile.

A picture description is made up by a list of picture commands. The picture commands are represented in the implementation by type `draw`:

```
> draw ::= Line pt pt | Text pt [char] | Rect pt (num, num) |
>          Dim num num | Colour num | Set_colour num num num num
>          | Origin (num, num)
> pt == (num, num)
```

A picture description is of type `[draw]`.

Instead of defining a graphical wiring function for each wiring form, we define a general purpose higher order wiring function called `draw_wiring`. One of the parameters to this function a the standard wiring function. This function is defined as:

```
> draw_wiring fn (ox, oy, z, acc)
>  = (ox+xdisp, oy, zs, ex++acc)
>    where
>    xsr = fn (in ox oy z)
>    ex = concat [[Colour v, Line npt (dx i npt)] | ((npt, v), i)
>                        <- zip2 xsr (index xsr)]
>         where
>         dx y (x,y') = (x+xdisp,oy+y*dy)
>    xdisp=((#z) div 2)*10
>    zs = [c | (p, c) <- xsr]
> dy = 20
```

Using this higher order function, we obtain the following non-standard definitions:

```
> nsi_riffle = draw_wiring riffle
> nsi_id = draw_wiring id
```

The non-standard definition for the processing node R simply draws R in a box, with two wires on the left and two on the right and does some colouring.

```
> r_width = 40
> r_height = 40

> draw_r (x,oy,[a, b],acc)
>  = (x+r_width,oy,[a, b],acc++
>    [Colour 2, Rect (x, y+2) (r_width, r_height-4),
>     Colour 3,  Text (x+20,y+20) "R"])
>     where
>     y = oy-10
```

Drawing a butterfly of size 0 will draw just R. Figure 9.1 shows Ψ_0 i.e. R. Like all the diagrams in this chapter, this picture was automatically produced by the non-standard interpretation for drawing Ruby and then converted to MacDraw II format for inclusion in this thesis.

Figure 9.1: R

Let us consider for the moment those butterflies networks generated by the first recursive description given (labelled with A). A butterfly of size 1 is show in Figure 9.2. Substituting $n = 0$ in the description gives:

$$\begin{aligned} & \textbf{ilv}\ \Psi_0\ R\ ;\ \textbf{two}^{0+1}\ R \\ = \quad & \textbf{ilv}\ R\ ;\ \textbf{two}\ R \end{aligned}$$

which corresponds with the figure drawn.

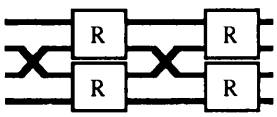


Figure 9.2: Butterfly of size 1.

A butterfly of size 2 is shown in figure 9.3. Notice that there are instances of a size 1 butterfly in this picture.

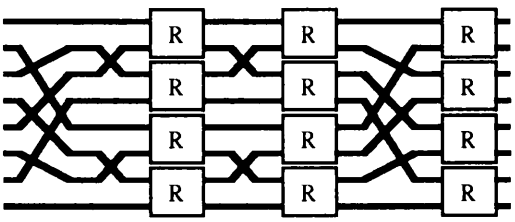


Figure 9.3: Butterfly of size 2.

A butterfly of size 3 is shown in Figure 9.4. Again, there are many instances of a size 2 butterfly in this picture.

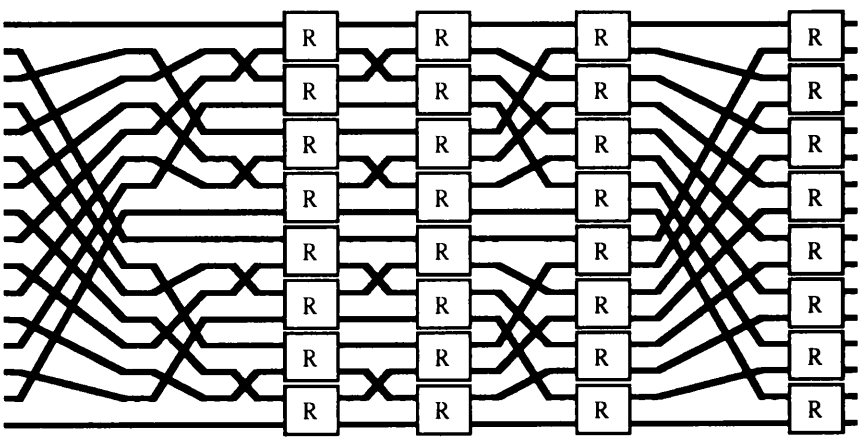


Figure 9.4: Butterfly of size 3.

A butterfly of size 4 takes up a whole page, and a colour plate of it is shown overleaf.

Butterflies of type B and D look the same when they are generated by a non-standard interpretation. Figure 9.5 depicts what is drawn.

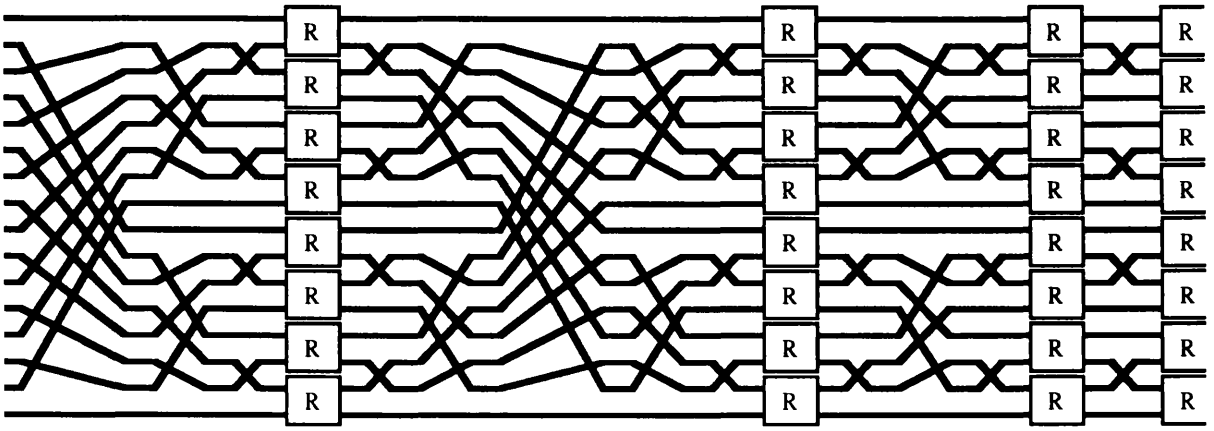


Figure 9.5: Butterflies B and D of size 3.

9.5 Drawing Non-Butterflies

In reference [Dowd et al. 89] a merger similar to Batchler's bitonic merger is presented. The balanced merger, so called because it merges two interleaved sorted lists, has been shown by Sheeran to have a recursive decomposition similar to the butterfly [Sheeran 91]. The following are Ruby descriptions of that butterfly-like network. Representatives from the following circuit descriptions have also been drawn.

```
> alt = two**n (one swap) ;;

> vee R = (ilv R) \ alt ;;

> wfly1 0 R = R ;;
> wfly1 n R = vee (wfly1 (n-1) R) ; two**n R ;;

> wfly2 0 R = R ;;
> wfly2 n R = wfly2 (n-1) (vee R) ; two**n R ;;

> wfly3 0 R = R ;;
> wfly3 n R = vee**n R ; wfly3 (n-1) (two R) ;;

> wfly4 0 R = R ;;
> wfly4 n R = vee**n R ; two (wfly4 (n-1) R) ;;

> rs 0 R = R ;;
> rs n R = ilv (rs (n-1) R) ; wfly1 n R ;;

> ex = ilv r ; vee r ; two r ;;
```

There circuits `wfly1` to `wfly4` are rather like butterflies, except they are based on `riffle ; alt` instead of `riffle`. Here, `vee` corresponds to `ilv`, but using `riffle ; alt` instead of riffle. Substituting `vee` for `ilv` in the original butterfly descriptions yields `wfly1` to `wfly4`.

These circuits are based on a component with four values at the domain and range. We need a new definition for drawing R:

```
> draw_r4 (x,oy,[a, b, c, d],acc)
>  = (x+r_width,oy,[a, b, c, d],acc++
>     [Colour 2, Rect (x, y+2) (r_width, 2*r_height-4),
>      Colour 3,  Text (x+20,y+40) "R"])
>      where
>      y = oy-10
```

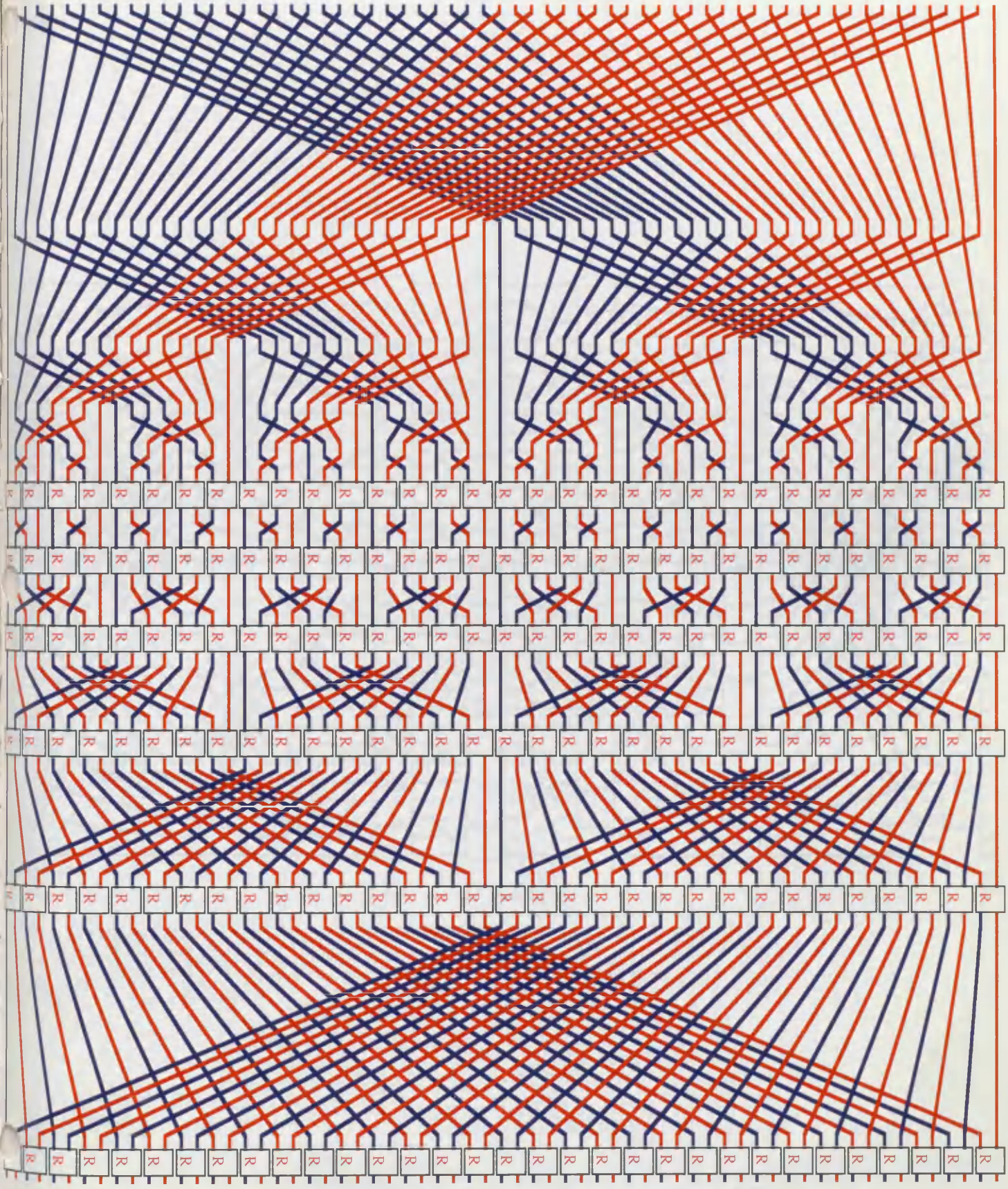
This draws a rectangle with four wires in the domain and four in the range.

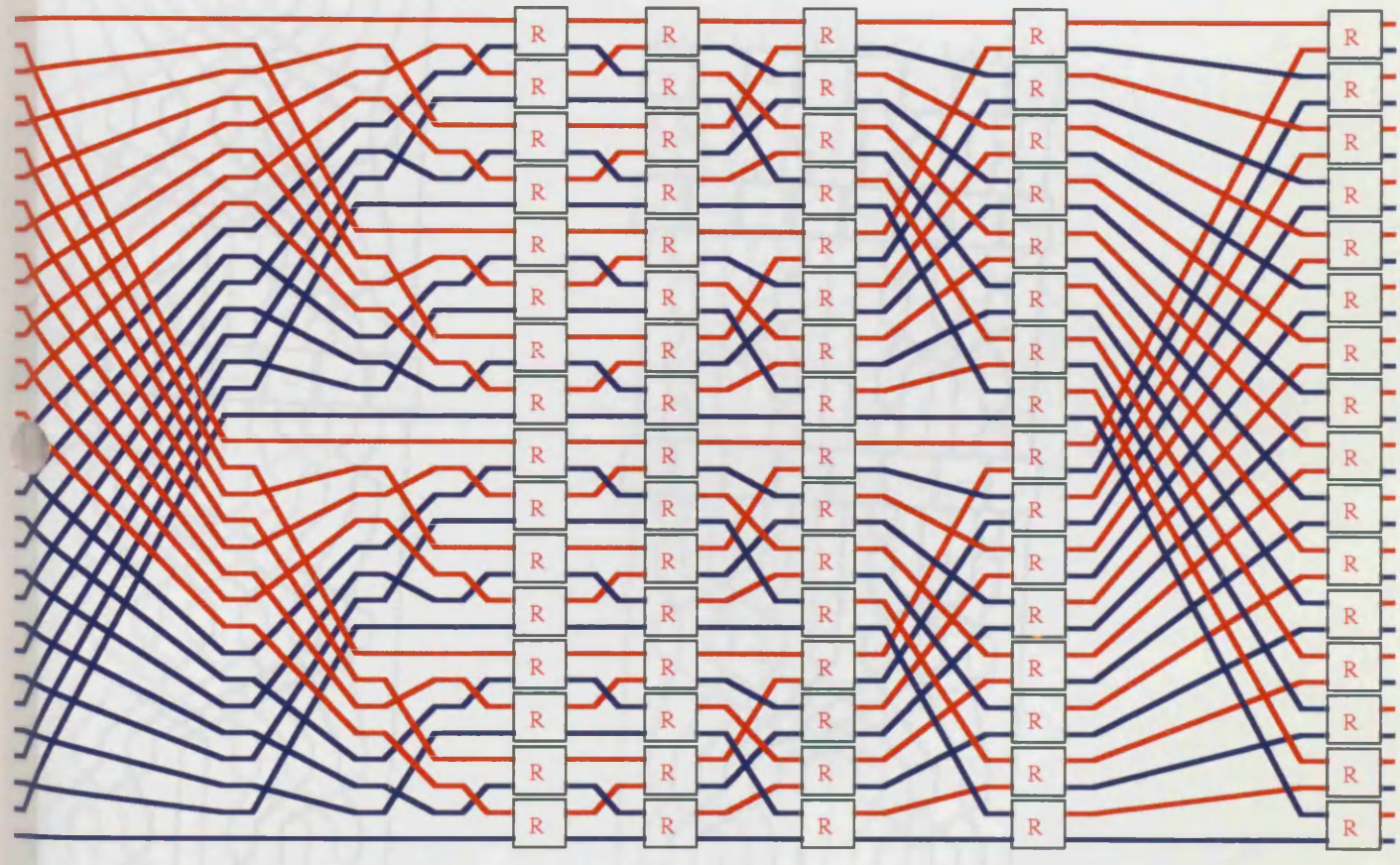
The semantics for serial composition is not altered. This is because each drawing function increments the current x-coordinate by the width it requires. Alternatively, the alternative semantics of the leaf nodes could be changed to give information about how wide the component is. Then, serial composition could be redefined to take account of this and increment the running x-coordinate by itself rather than have it done explicitly in the code for the leaf nodes.

The alternative semantics for parallel composition is defined only for the case where two circuits are placed in parallel. However, these two circuits may themselves be parallel compositions. It is defined as follows:

```
> parid r (ox, oy, nl, acc)
>  = (ox'', oy, a++n'', acc''++acc'++acc)
>     where
>     acc' = idwires (ox, oy, ox''-ox, #nl div 2)
>     (ox'', oy'', n'', acc'') = r (ox, oy+dy*(#nl div 2), b, [])
>     [a, b] = halve nl
```

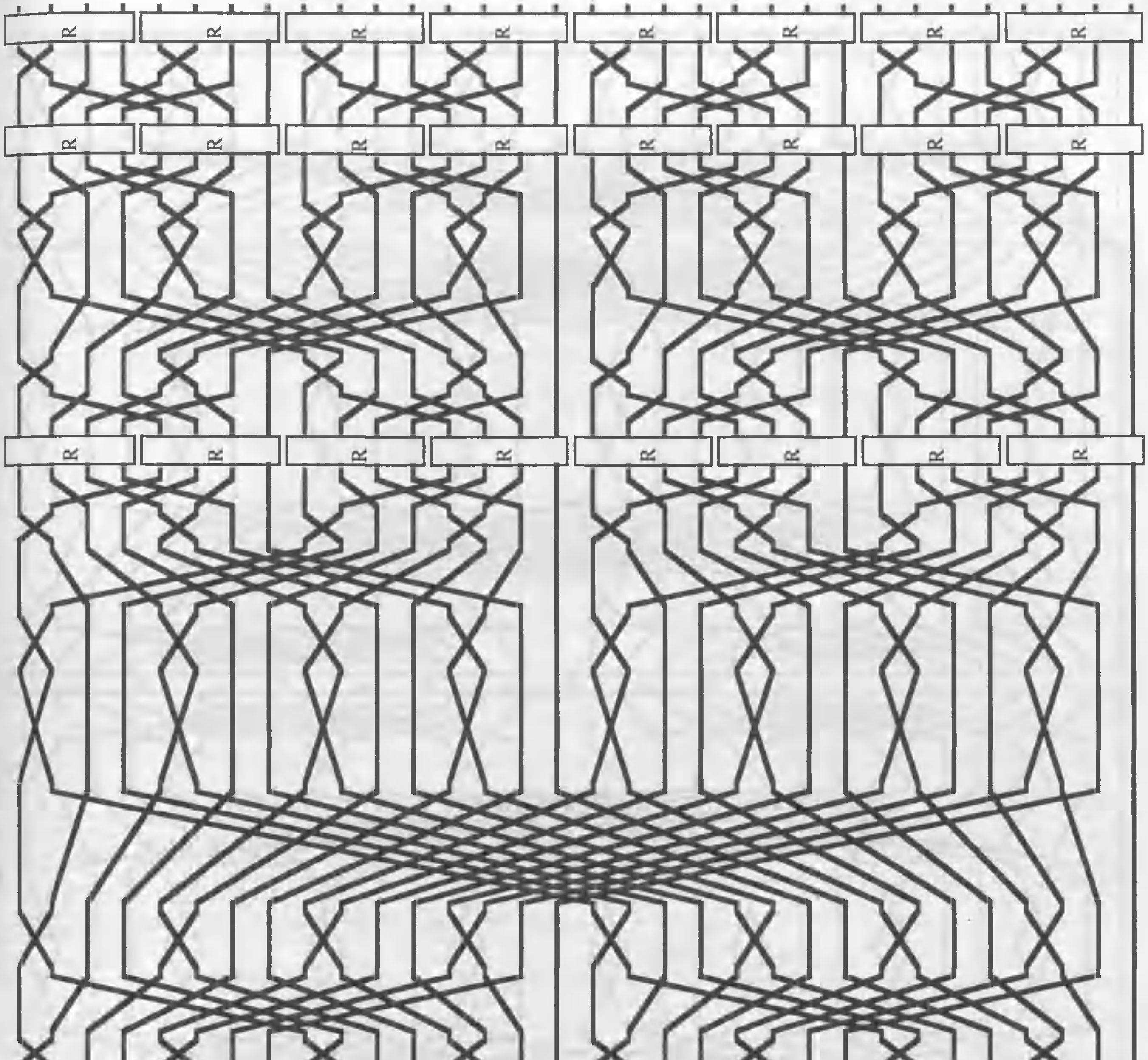
The next few pages show drawings produced by non-standard interpretation for some of the circuits described above. The first is a butterfly (A) of size 5. The following page shows a butterfly (A) of size 4. Butterflies C and B/D are shown next, both of size 4. The next page shows three circuits: (a) a periodic balanced sorter (b) a recursive sorter based on the balanced merger (c) a shuffle-exchange network. The next two pages show black and white drawings of vee-based butterflies with 4 to 4 components.

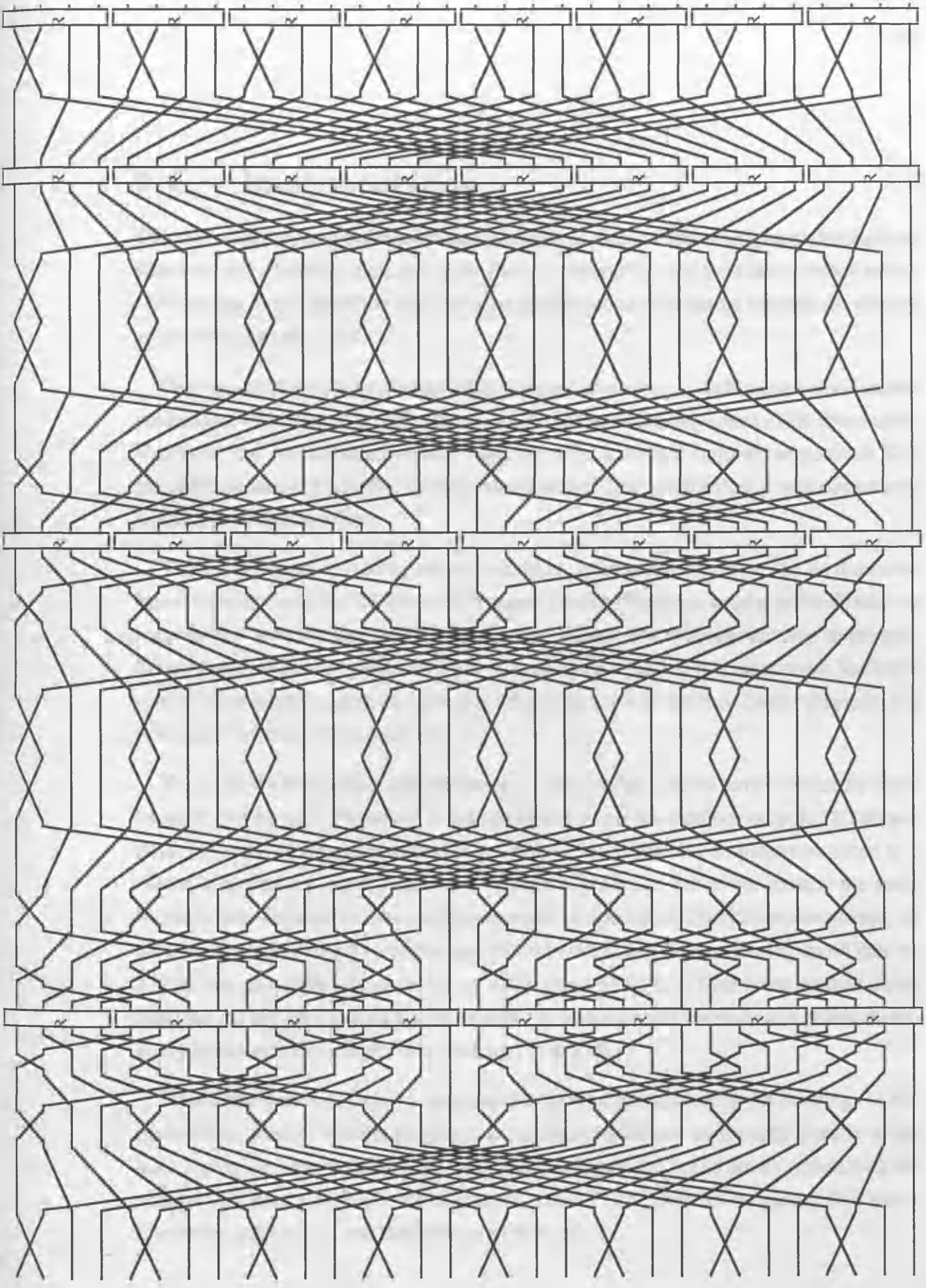




A butterfly of size 4

R
(a)
(b)
(c)


R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R



9.6 Implementation

The non-standard interpretation system produces as output a file of graphical descriptions based around a Miranda algebraic type. Several other programs have been written which translate the output from the non-standard interpretation to a format suitable for driving graphical output devices.

One variant of the above interpretation involved producing LaTeX picture environment commands. This allows butterfly drawing to be included directly into LaTeX documents. However, the picture environment supports only a limited number of possible line gradients. It was not possible to draw butterflies of size greater than 4 without exactly aligning the wires in riffles.

The butterflies produced by the non-standard interpretation system can be drawn on Sun computers running the SunView window system (formerly known as SunTools). A program was written using the Unix tools Lex and Yacc and the programming language C for parsing lists of the type `draw`. This program then executed the appropriate SunView system calls to drive graphics canvases. The program will produce colour displays in a re-sizeable window with scrollbars.

To produce a black and white hardcopy, a utility called ScreenDump, written by Mark Dunlop, can be used. However, it was desirable to produce colour output. To achieve this, the output files were transferred to a colour Macintosh II computer connected to a colour laser printer. A program was written in LightSpeed Pascal for parsing the same Miranda data structure and then calling appropriate Macintosh QuickDraw commands for drawing butterfly circuits in colour on a Macintosh display. These drawing could then be pasted into the clipboard or saved as PICT files. The PICT files were read in using MacDraw II and then pasted into Microsoft Word documents for inclusion in this thesis. Every drawing in this chapter was produced in this way.

From MacDraw II, colour output could be produced by sending the drawings to the colour laser printer. Another program for sending QuickDraw commands directly to the laser writer was also produced. This program works faster, but is not as versatile as the version that produces PICT format output, since PICT files can be pasted into many Macintosh applications and then annotated by hand.

9.7 Conclusions

Ruby contains information about circuit layout as well as behaviour. All the non-standard interpretations presented up to this point have involved finding alternative meanings for Ruby behaviour. However, we can also produce layouts by changing the standard behaviour to draw pictures. We have drawn butterfly-type circuits using this technique.

However, we have not shown how to draw an arbitarary Ruby description. This is a hard problem, since it is not easy to see how to lay out four sided tiles, or even tell what is a four sided tile. This is because Ruby does not offer any primitive support for four sided tiles. The drawing of circuits with four-sided tiles might be possible by using contextual information.

Another problem is how to deal with generic circuits like **map** AND. We can choose to draw an arbitrary representative of this class, or we could make up a special symbol, perhaps involving vertical dots to shown that this picture does not represent a particular circuit. Our implementation avoids this problem by drawing only fixed sized circuits. Generic combining forms can be used, as long as there is enough context information available to determine the size of the domain or range values for generics.

One interesting application for non-standard interpretation might be for synthesis of mask level layout. A non-standard interpretation could be devised for synthesising a mask level layout for each processing node and wiring form. This could be done by using an existing CAD tool. The combining forms then glue together the constituent layouts to form complete circuits. This would be a viable approach because Ruby contains enough topological information. This means that we avoid the hard problem of how to automatically lay out large circuits. The layout of the high-level blocks is done explicitly in Ruby. The relatively tedious task of producing layouts for AND, OR, FORK, ID etc. can be done economically using automatic layout tools.

One advantage of this method is that it allows the designer to experiment quickly with different layout. The designer can concentrate on the layout at a high level of abstraction, knowing that the cells near the leaves of the design tree will be dealt with automatically.

Chapter 10

Improving Testability

10.1 Introduction

Ruby possesses many powerful algebraic laws and properties that help designers reason about circuits. Circuit designs can be derived from a high level specification using correctness preserving transformations that guarantee the validity of the final design.

Many of the existing rules about Ruby transform its physical layout in order to save area or increase speed. This chapter presents transformations that improve the testability of Ruby descriptions.

The first step is to declare a strategy for testing Ruby descriptions (section 10.2). Testability has eluded precise formal descriptions: much of the existing traditional techniques for improving testability work by trying to increase *controllability* and *observability*. This is accomplished by improving access to internal nodes by introducing extra circuitry (as shown in chapter 2). The first part of the test strategy is to employ traditional techniques for improving testability. Ruby is particularly suited to describing regular circuits which often have replicated cells . The second part of the test strategy is to capitalise on this replication by testing repeated blocks in parallel.

Having set out the test strategy, the next step is the introduction of a testability transformation scheme (section 10.3). Transformations are defined for the most common Ruby combining forms and for the sole sequential element $\mathcal{D}$.

The use of the testability transformation can lead to very messy and cluttered Ruby descriptions. Also, the designer might have to do some trivial re-wiring to connect together subcircuits that propagate testability information in an incompatible manner.

Particular attention is paid to the transformation of serial composition since this is the combining form that is likely to induce the highest degree of untestability. The next step is a superficial extension to Ruby to allow the expression of testability requirements while still retaining notational elegance.

The remainder of the chapter explores the possibility of exploiting the behavioural and layout information conveyed by Ruby with a view to producing more testable circuits. By careful examination of the behaviour of a composite circuit, a decomposition may be found which results in a more testable structure.

10.2 Testing Strategy

The transformations for improving testablility introduced here have been drawn up with the following considerations in mind. Firstly, the methods used for improving the testability of circuits are traditional ones that aim to increase the controllability and observability of internal nodes. This is accomplished by allowing direct or improved access to internal nodes. This is done by splitting sequential networks into parallel networks. Secondly, the emphasis is on evaluating whether a given fabricated design is working or not: there is no attempt to diagnose the fault. This results in cheaper transformations which give a "go/no-go". This is often all that is required.

If a regular circuit is converted to a flattened netlist before applying testing analysis, the advantage of regularity is lost and the analysis must be redone for each repeated circuit. However, since we are satisfied with a "yes/no" answer, it is possible to test each replicated cell in parallel. This saves much testing circuitry and test application time. The test technique for replicated cells involves distributing the same signal to each replicated cell and comparing the outputs of each cell. If there is any difference in output response between the various cells there there must be at least one faulty cell. An assumption is made about how circuits fail: it is assumed that all the replicated cells of some circuit F will not *all* fail in the same manner

Sequential circuits are tested by using the traditional technique of decomposition to a set of combinational circuits and then testing these. The sequential elements are chained together to form a shift-register to allow economic testing.

10.3 Transforming Combining Forms

10.3.1 Introduction

In this section, a testability transformation called $\mathcal{T}$ is introduced. This transformation is defined over the various combining forms and elements of Ruby. Each transformation aims to increase the testability of a circuit by increasing the controllability and observability of internal nodes and by performing as much parallel testing as possible.

A Ruby description can be analysed using existing tools or the non-standard interpretation system presented earlier. Areas of poor testability can then be transformed using $\mathcal{T}$.

The remainder of this section defines plausible transformations for some of the most common Ruby constructs.

10.3.2 Serial Composition

Serial composition connects two communicating circuits together via an internal connection. This internal connection can be very hard to control and observe.

The testability of serial composition is improved by transforming it to a network which allows the internal connection to be directly controlled and observed. Consider the circuit in figure 10.1(a) which shows circuits A (domain x) and B (range z) connected via an internal connection y. The circuits A and B might be quite testable independently, but the composition makes it harder to apply test vectors to B thus reducing B's controllability. Since any test information about A must be propagated through B, A's observability is reduced.

The situation is exacerbated if the range of A is not a superset of the test program for B. This can cause some faults in B which were testable with B in isolation to become untestable. Let P_A be the test test program for A and P_B be the test program for B. Let F_A be the set of faults in A (similarly for B). The function $C(P,G)$ gives the set of faults in circuit G which are exposed by the test vectors in set P. The cover for A is then:

$$\frac{|C(P_A,A)|}{|F_A|}$$

The cover for B is defined analogously.

In addition to exercising A with P_A, A must be made to produce as many elements of P_B as possible at its range. The additional elements are given by the relation $P_A' \; A \; P_B$. Thus the test program for the composition A ; B is given by the union of P_A and P_A':

$$P_{A;B} = P_A \cup P_A' \tag{10.2}$$

In general, it is possible that A cannot generate all of P_B at its output i.e. $rng\ A \subset P_B$. This constrains the set of possible values at the intermediate signal y to be $rng\ A \cap P_B$. This cover the of composition is now given by the left hand side of the inequality:

$$\frac{|C(P_A,A)| + |C(rng\ A \cap P_B, B)|}{|F_A| + |F_B|} \leq \frac{|C(P_A,A) \cup C(P_B,B)|}{|F_A| + |F_B|}$$

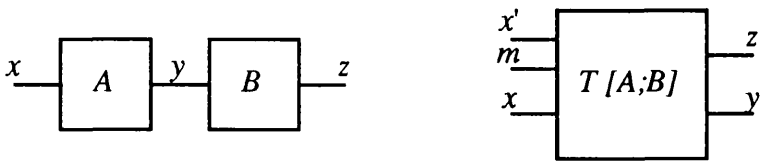
This inequality becomes an equality when $rng\ A \geq P_B$. The degree to which the cover is reduced depends on the behaviour of A and B and the test program P_B.

Reduced cover can occur when the composition of A and B contains redundant circuitry. Although A and B can be free of redundant subcomponents, their composition might not be, especially if A's range is a subset of all the possible 2^n values for an n-bit output. This situation is likely to occur in hierarchical design when ready built general components are plugged together. The resulting implementation may not only satisfy the specification but may also have unnecessary extra behaviour. This can occur when very general or non-cohesive components are used.

One way to overcome this is to analyse the composition with a view to removing any redundancy. This is undesirable in general because it diminishes one of the advantages of hierarchical design using correctness preserving transformations. Such tailoring of ready built and proved designs is a large burden on the designer: if the redundancy is removed by using ad-hoc methods then the resulting circuit would have to be re-verified in a bottom-up manner. Much of the work that went into designing a ready built and correct by construction (or exhaustive testing) library module is lost.

The problem can be alleviated by choosing a slightly different test program for P_B. For any given design B, there is likely to be more than one possible test program that gives optimum or near-optimum test cover. Normally, after fault collapsing etc. one test set is arbitarily chosen and the rest discarded. However, if this choice were delayed until B is actually used in some larger design, then a better choice could be made for the test program. For example, two test programs for B, P_1 and P_2, might be equally good for testing B in isolation but one might be better than the other when the composition A ; B has to be tested. Let t be an essential test for A. If $t \in P_1$, $t \notin P_2$ then by choosing P_1 some faults in A are also covered. These extra faults are discovered by using fault simulation. This suggests that P_1 would be a better choice than P_2.

A fault f in B might be covered by the essential test t_1 in P_1 and t_2 in P_2. If, however, $t_1 \notin rng\ A$ then P_1 will not expose fault f. If $t_2 \in rng\ A$ then this would be a better choice since fault f can still be exposed. By making these kinds of analyses for all the tests, a much better choice can be made between possible test programs. This will help to reduce (or halt) the reduction of the fault cover that may occur when two circuits are composed in series. The penalty to pay is delayed and more expensive test pattern analysis which has to be performed in a contextual manner instead of in isolation for each module.



(a) (b)

Figure 10.1: (a) Serial composition (b) external behaviour transformation.

The problem of diminished controllability and observability of the internal connection in serial composition can be reduced by transforming the design to allow direct controllability and observability of the internal connection. This involves changing the behaviour of the circuit. The circuit in figure 10.1(a) is transformed to have the external behaviour of the circuit shown in figure 10.1(b). The input has added to it a boolean input m which describes the mode in which the circuit is operating in. In normal mode, circuit B receives its input from the output of A (normal serial composition). In test mode, B's input is taken from the second additional input x', thus increasing the controllability of the internal connection z. The output is modified by always making the value at the internal connection z visible through the second element of the 2-tuple range, thus increasing the observability of the internal connection.

It is possible that only some of the wires in the intermediate connection z are difficult to observe. In this case, appropriate selectors can be applied to the second element of the range tuple to filter out the desired wires. Similarly, it might be the case that not all of the intermediate wires are difficult to test. This case involves the more difficult task of redesigning the transformed circuit to share some of the wires in x with x'.

This transformation is specified formally by:

$$\langle x, \langle m, x' \rangle \rangle \; \mathcal{T}[\![A;B]\!] \; \langle y, z \rangle \quad \rightarrow \quad \exists q \; .(\neg m(q{=}z) \vee m.(q{=}x')) \wedge (x \; A \; z) \wedge (q \; B \; y) \tag{10.3}$$

When m is true, then circuit is not in test mode so it behaves like $A;B$.When m is false then the circuit is in test mode. Circuit B should now receive its inputs from the test input x' instead of x.

The specification of a 2 to 1 multiplexer is given below. This circuit is assumed to be part of an available library of specified, implemented and proved correct circuits.

$$\langle a, b, c \rangle \; MUX \; d \quad \Leftrightarrow \quad a.(d{=}b) \vee \neg a.(d{=}c)$$

This component can be used to realise the expression:

$$\neg m(q{=}z) \vee m.(q{=}x')$$

in specification (10.3), giving:

$$\langle x, \langle m, x' \rangle \rangle \; \mathcal{T}[\![A;B]\!] \; \langle y, z \rangle \quad \equiv \quad \langle m,x',z \rangle MUX \; q \wedge (x \; A \; z) \wedge (q \; B \; y)$$

Using the definition of serial composition, the multiplexer is composed in serial with circuit B.

$$\langle x, \langle m, x' \rangle \rangle \; \mathcal{T}[\![A;B]\!] \; \langle y, z \rangle \quad \equiv \quad \langle m,x',z \rangle \; (MUX \; ; B) y \wedge (x \; A \; z)$$

Since A takes its input from the bottom and $MUX \; ; B$ from the top, these circuits are composed in parallel:

$$(x, (m, x')) \; \mathcal{T}[\![A;B]\!] \; \langle y, z \rangle \quad \equiv \quad \langle x, \langle m,x',z \rangle \rangle \; [A, MUX \; ; B] \; \langle r, s \rangle \wedge r=y \wedge s=z$$

The wires at the output of the composition are the wrong way round: the output of B should go to the bottom (i.e. y) and the output of A should go to the top (i.e. z). This problem can be fixed by swapping these wires:

$$\langle x, \langle m, x' \rangle \rangle \; \mathcal{T}[\![A;B]\!] \; \langle y, z \rangle \quad \equiv \quad \langle x, \langle m,x',z \rangle \rangle \; [A, MUX \; ; B] \; ; \; \mathsf{swap} \; \langle y, z \rangle$$

This right hand side is not directly realisable because z occurs in the domain and the range. The **loop** combining form can be used to feed back the z from the range back to the domain. However, z must also appear as the second element of the range tuple. For this reason, z is duplicated by a fork. A rewiring relation is required to keep the type of the domain type right i.e. the tuple must be transformed from $\langle y, \langle z,z \rangle \rangle$ to $\langle \langle y, z \rangle,z \rangle$. This is accomplished by the relation *reorg*1. Similarly, the domain tuple must be reorganized so that A and MUX receive the right signals: this is done by *reorg*2.

$$\langle x, \langle m, x' \rangle \rangle \; \mathcal{T}[\![A;B]\!] \; \langle y, z \rangle \quad \equiv \quad \langle x, \langle m, x' \rangle \rangle reorg2 \; ; \; \mathsf{loop} \; ([A, MUX \; ; B] \; ; \; \mathsf{swap} \; ; \; [\iota, \mathsf{fork}] \; ; reorg1)$$

$\langle y, z \rangle$

$\langle y, \langle z,z \rangle\rangle$ *reorg*1 $\langle\langle y, z\rangle,z\rangle\rangle$

$\langle x, \langle m, x'\rangle, z\rangle$ *reorg*2 $\langle x,\langle m,x',z\rangle\rangle$

This is now a realisable description. Since it was derived from the specification using correctness preserving transformations, it must also be correct by construction. This description can be simplified by applying the law:

$$[A, B] \; ; \; [C, D] \qquad = \qquad [A \; ; \; C, \quad B \; ; \; D] \tag{10.4}$$

which results in:

$\langle x, \langle m, x'\rangle\rangle \; \mathcal{T} \llbracket A;B \rrbracket \; \langle y, z\rangle \quad \equiv \quad \langle x, \langle m, x'\rangle\rangle$ *reorg*2 ;
loop ([A ; fork, MUX ; B] ; swap ; *reorg*1)
$\langle y, z\rangle$

A picture of this circuit is shown in figure 10.2. The wiring reorganizations are not shown for clarity.

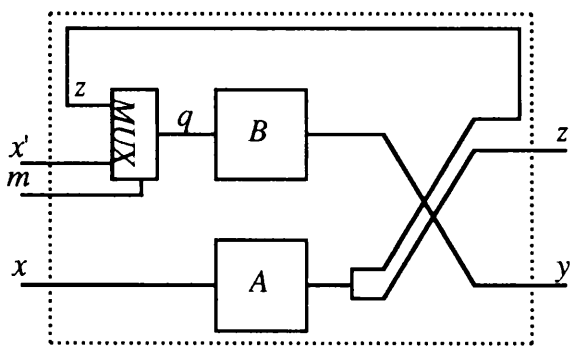


Figure 10.2: Transformed serial composition.

This transformation has added circuitry which can also fail. The complexity of A ; B must be large enough to overcome this overhead. An alternative arrangement to that shown in figure 10.2 could involve feeding m through as an output.

Notice that this arrangement allows the parallel testing of A and B. In test mode, the output z gives the results of tests to A applied at x and the output y gives the results of the tests applied to B at x'. This is because in test mode, the series connection is effectively broken and the circuit behaves as if it were a parallel composition.

Using the very approximate testability measure $(i+o)/g$ where i is the input of inputs, o is the number of outputs and g is the number of gates, the testability effort of a series composition has been reduced to be the same as the parallel case i.e. from $(ia+ob)/(ga+gb)$ to $(ia+oa)/ga + (ib+ob)/gb$, where ia, io and ga are A's inputs, outputs and number of gates (similarly for B).

10.3.4 Map

Since **map** F replicates F, test pattern analysis can be done for one F and reused in every other instance. One straight forward way to test circuits in a **map** structure would be to use the following transformation rule:

$$\mathcal{T}[\![\textbf{map}\ F]\!] \quad \rightarrow \quad \textbf{map}\ (\mathcal{T}[\![F]\!]) \tag{10.9}$$

This involves more than doubling the number of wires that go into and out of the circuit generated by the **map** . Since all the F's are identical and in general don't interact with each other, they can be tested simultaneously by applying the test vectors to each F. This gives a substantial saving of input lines since only one test mode wire and a signal the size of one F's domain needs to be added. However, the number of output lines has still doubled.

It is unnecessary to propagate the observability output of each F in test mode to an observable output. The testability information required is 'is there a faulty F in the parallel structure?'. Assuming that all the circuits will not fail at once in the same way, the network is almost certainly working if all the F's give the same answer to the same stimuli (assuming that the internal state elements have been initialised consistently). So it is sufficient to additionally propagate the result of only one $\mathcal{T}[\![F]\!]$ and a bit indicating whether all the mapped F's produced the same output (i.e. are working).

A formal description of such a transformation is:

$$\langle xs, t\rangle\ \mathcal{T}[\![\,\textbf{map}\ F\,]\!]\ \langle ys, \langle w, z\rangle\rangle$$
$$=_{def}\ \langle xs, t\rangle\ dist_R\ ;\ \textbf{map}\ (\mathcal{T}[\![F]\!])\ ;\ \text{split}\ ;\ [\textbf{map}\ \pi 1, G]\ \langle ys, \langle w, z\rangle\rangle$$
$$\text{where}\qquad G = \text{split}\ ;\ [comp, \pi 1] \qquad (10.10)$$

The normal input is the list xs and the test input is t (which is distributed to each mapped F). The result contains the normal output list ys along with w which indicates whether there were any faults found in the mapped F's and z which contains the test output value of one of the Fs.

The component *comp* is a generic comparator which takes a list of signals on its domain and gives true at its range if they are all the same, and false if they are not.

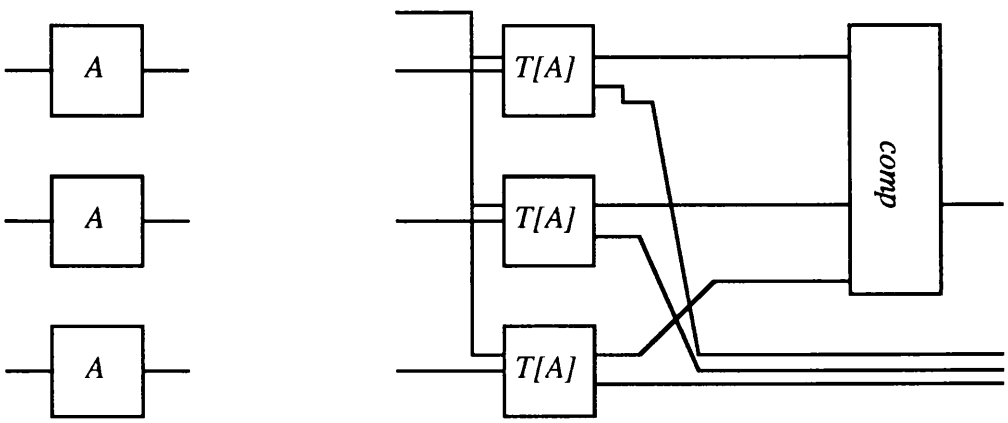


Figure 10.4: (a) Instance of **map** F (b) corresponding $\mathcal{T}$ transformed circuit.

An instance of **map** F and its transformation is shown in figure 10.4.

10.3.5 Loop

The problem with testing loop circuits is the hidden internal feedback line and the internal state that most loop circuits posses. The transformation adopted for loop circuits breaks the feedback loop to allow it to be directly controllable and observable by undoing the loop. The sequential elements are dealt with in a later section.

We choose to analyse loop circuits by breaking the feedback loop. The transformation used is:

$$\mathcal{T}[\![\textbf{loop}\ F]\!] \quad \rightarrow \quad \mathcal{T}[\![F]\!] \tag{10.11}$$

This simply undoes the loop. We have to remember that the resulting circuit is a pair to pair relation. The second element of the range pair is used to form the feedback path which is now broken.

An alternative method for analysing loop circuits is to unfold the loop, as is done by other analysis like symbolic evaluation.

10.3.6 The Delay Element

The delay element $\mathcal{D}$ is the only sequential component in Ruby. A transformation for this element will in turn affect all Ruby sequential circuits. Traditional techniques for dealing with state elements include isolating the delay circuitry from the combinational circuitry so that these can be tested as two separate subsystems. This reduces the complexity of testing from $O(2^{n+m})$ to $O(2^n)+O(2^m)$ where n is the number of delay elements and m is the number of state elements when performing exhaustive testing.

The behaviour of the $\mathcal{D}$ element is modified so that in test mode it can be directly controlled. The following transformation is used:

$$\langle x, \langle m, x' \rangle \rangle\ \mathcal{T}[\![\mathcal{D}]\!]\ y \quad \rightarrow \quad m\ (x\ \mathcal{D}\ y\ )\ \vee \neg m\ (x'\ \mathcal{D}\ y) \tag{10.12}$$

Here, x is the normal input to the delay element and m is the test mode. If the circuit is not under test, then the transformed circuit's output is a single time unit delayed version of x. When the circuit is being tested, then the input x', which is assumed to be part of a scan-path chain, is related to y through $\mathcal{D}$.

10.3.6 Row and Col

Since all the elements of these structures are the same, then it should be possible to tests these elements in parallel, in a similar manner to **map**. For **row** F, a row of F's can be placed below a distributed common test signal along with the normal vertical input, giving the transformation:

$$\langle h, \langle m, h' \rangle, v \rangle\rangle \; \mathcal{T}[\![\, \textbf{row}\ F \,]\!] \; \langle y, \langle z,w \rangle\rangle$$
$$\rightarrow \quad \langle h, \langle m, h' \rangle, v \rangle\rangle\ \textbf{snd}\ ;\ \mathrm{dist}_L\ ;\ \textbf{row}\ (\mathcal{T}[\![F]\!])\ ;\ \textbf{fst}\ (\mathrm{split}\ ;\ \textbf{snd}\ comp)\ \langle y, \langle z,w \rangle\rangle \tag{10.17}$$

Here, h is the horizontal input, v the vertical input, with m the the mode bit and h' the direct control value corresponding to h. The normal horizontal output is y and the vertical normal output is z. The w line is false if there was a discrepancy found between the F's in test mode. Figure 10.5 shows a picture of the transformation for an instance of **row** F.

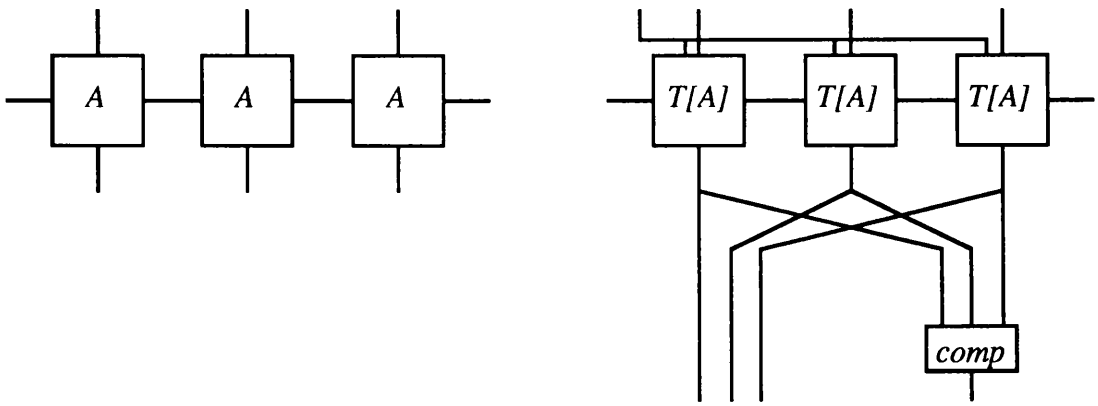


Figure 10.5: (a) **row** F (b) and its transformation.

The transformation for **col** is defined analogously.

10.3.7 Repeated composition

Since all the cells in a repeated composition are the same, it is possible to test all of them in parallel and then 'and' the outputs. One possible scheme is shown in figure 10.6.

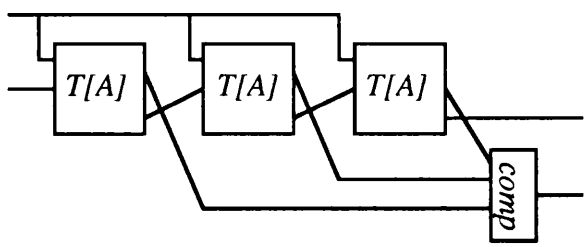


Figure 10.6: A possible transformation for repeated composition.

The test vector and mode is distributed to each cell from the top. The test outputs are compared to check if they are all the same by the comparator at the bottom.

A similar technique can be applied to **tri** and **irt** triangles.

10.4 Augmenting Ruby for $\mathcal{T}$

The transformations in section 10.3 have the unfortunate affect of transforming beautiful designs into ugly, complicated designs. The rules are also not totally consistent with each other e.g. the test output of a **col** does not feed in directly to the test input of a normal serial composition. It is the designer's responsibility to organise the interface between different types of structures, although this is usually trivial. If descriptions become too complicated to reason about easily, then Ruby will becomes less effective as a design tool.

All of the transformations in $\mathcal{T}$ are of a mechanical nature. They all transform Ruby expressions to Ruby expressions. Instead of making the designer explicitly use the rules of $\mathcal{T}$ of transform circuits extra constructs could be added to Ruby to describe where testability transformation should take place. This augmented version of Ruby could then be translated into a series of semantically equivalent $\mathcal{T}$ transformations along with any glue transformations required.

As an example, consider the problem of a hard to test internal node in a serial composition $A\ ;\ B\ ;\ C\ ;\ D\ ;\ E$. Let the difficult node be between C and D. The testability of a series network is improved if it could be mode more parallel. A pseudo-parallel construct could be defined with the following 'top level' semantics:

$$\langle A\ ;\ B\ ;\ C, D\ ;\ E\rangle \qquad \Leftrightarrow \qquad A\ ;\ B\ ;\ C\ ;\ D\ ;\ E \tag{10.18}$$

This indicates that $A\ ;\ B\ ;\ C$ should be placed in parallel with $D\ ;\ E$ for the purposes of testing. However, the designer uses the expression on the right hand to to reason about this construct i.e. normal serial composition. There is no overhead of having to reason and route extra wires: these are handled automatically by the transformation from augmented Ruby to transformations in $\mathcal{T}$ on standard Ruby.

A possible transformation from enhanced Ruby to the standard Ruby under $\mathcal{T}$ could be:

$$\langle A\ ;\ B\ ;\ C, D\ ;\ E\rangle \qquad \rightarrow \qquad \mathcal{T}[\![(A\ ;\ B\ ;\ C\ )';\ (D\ ;\ E)']\!] \tag{10.19}$$

Here, $\mathcal{T}$ is modified to transform dashed expressions by leaving them unchanged.

Other transformations could be used to link up all the $\mathcal{D}$ latches into a shift-register chain for LSSD testing. It might be profitable to have more than one LSSD shift-register, or to exclude certain delay elements for global shift registers. One reason for doing this would be to reduce the overhead of a snakinging LSSD path through a large circuit.

10.5 Conclusions

The increasingly important requirement to accommodate testability constraints early on in the design of digital systems makes the design task more complex. Many of the advantages of using hierarchical algebraic languages are diminished by having to consider in great detail the testability characteristics of each construct.

To aid the designer a testability transformation $\mathcal{T}$ utilizing traditional test techniques has been presented. This technique uses proved transformation rules to help transform an existing design to be more testable.

Transforming various parts of the design introduces the need to perform much tedious re-wiring amongst the various methods used in $\mathcal{T}$ for propagating fault information. This problem is overcome by extending the description language, Ruby, with constructs which identify those parts of the design which have to be made more testable. The tedious wiring can now be automated in a translation from augmented Ruby to standard Ruby. This increases the testability of Ruby designs while still retaining tractable descriptions with pleasing algebraic properties.

An alternative approach suggested by Sheeran is to make each circuit a pair to pair circuit. The first element of the pairs are used to drive standard Ruby descriptions. The second are used to drive Ruby circuits that have been transformed, perhaps by the method given above. Depending on whether the circuit is being tested, either the first or the second element will be 'active'. This method has the advantage that circuits can be composed together just as easily as before.

Chapter 11

Conclusions

We have shown that non-standard interpretation provides a good framework for expressing many hardware analyses. Drawn from the domain of testability, we have shown how several non-trivial analyses can be expressed easily and quickly. We claim that non-standard interpretation provides a paradigm for hardware analyses. This is not surprising if we consider that most hardware analyses manipulate circuit representations which are isomorphic to the circuit description. However, often the circuit has to be respecified before analysis, and the structure analysed bears no connection to the structure built when compiling the circuit description. This duplicates much effort which can be saved by using just one representation. It also avoids the problem of inconsistency between representations.

The motivation for using non-standard interpretation arose from the desire to use just one circuit representation for many analyses. The first non-standard interpretation was written to estimate the number of test patterns required to exhaustively test a circuit. This kind of approximation analysis seemed similar to the approximations used for functional programs to discover strictness.

Taking inspiration from the analyses used by functional programmers to detect strictness in lazy functional languages, we have adapted their abstract interpretation techniques to work on Ruby hardware descriptions. In doing so, we have had to settle for the more general technique of non-standard interpretation, which provides a weaker association between the standard and non-standard interpretations. This makes it harder to prove safety properties of our analyses.

A powerful algebraic relational hardware description language was used for performing non-standard interpretation. A large subset of the language was implemented— a major undertaking. A restriction has been placed on the nature of information flow, although this does still allow us to capture most circuits of interest. The standard interpretation can be used to simulate circuits in Ruby. This has been used to simulate circuits ranging from individual gates, through arithmetic circuits like 32-bit adders to large butterfly networks. Even by itself, the standard interpretation is a very useful tool.

The non-standard interpretations extend the power of the system. Normally, type checking is an integral part of a compiler. However, we have demonstrated that type-checking can be easily captured as a non-standard interpretation. This making type checking much easier to provide, and the language implementation much simpler. However, the system by default does not run the type checking non-standard interpretation on every design— it is up to the user to execute each design with the type checker. This is similar to the case that exists in C where a separate program, often lint, is used to perform type checking.

Ruby provides a geometric interpretation for circuits as well as a behavioural description. A drawing interpretation exists for producing circuit layouts automatically. The interpretation for producing colour drawings of butterfly circuits demonstrates strikingly how much can be done by non-standard interpretation. The alternative semantics needed to draw these butterflies fits easily onto a page. We have produced a useful tool for a small investment by re-using much code from the standard semantics. The re-use was made possible by the carefully chosen non-standard interpretation model. The layout interpretation was written in just one afternoon.

We have presented more than just a method for providing alternative semantics easily with much re-use. We have also shown how to combine non-standard interpretations to produce new interpretations. Combining interpretations is a powerful technique, and future work could involve finding even more combinators. The theoretical implications of combining interpretations are only just being explored by functional programmers. Any advance in the theory would be directly relevant to the non-standard interpretation technique we have presented.

Combining forms have been introduced to combine small interpretations into bigger ones. This allows problems to be sub-divided into smaller problems and solved independently. This is good software engineering practise and allows circuits with complex bi-directional data flow to be modelled by a series of uni-directional interpretations. These uni-directional interpretations can be implemented efficiently using functions, rather than relations which would be required to implement a bi-directional interpretation.

A typical integrated circuit design project involves the use of several analysis programs. Usually, these programs are poorly integrated and it is not possible to combine them. The technique we have presented allows interpretations to be combined easily to form new interpretations. One convincing example is of how a testability measure interpreter (SCOAP), an automatic test pattern generation interpreter (D-algorithm) and a fault simulator (deductive fault simulation) have been combined to produce a more efficient path oriented automatic test pattern generator (PODEM).

Attempting to cast a circuit analysis as a non-standard interpretation disciplines us to think of the analysis in a precise manner. At first sight, it may not seem possible to express an algorithm as complex as the D-algorithm as a non-standard interpretation. However, after coming up with a more precise description of D-intersection and finding a decomposition of the algorithm into subproblems we were able to cast this algorithm as a NSI.

The implementation runs very slowly, although full performance cannot be expected of a prototype. There is some scope for improving the performance of the central non-standard interpreter builder. This would then automatically speed up all other non-standard interpretations. However, we have decided to stick with a simple and correct implementation. Future work in this area might involve using formal transformations to improve the performance of the system. We have not been able to compare the performance of this system with commercial systems.

The analyses we have presented deal with only discrete non-standard values. However, there is no reason why analogue quantities should not be used as non-standard values. A very low level analogue description could be abstracted to produce a digital version. Providing the abstraction is safe, this could be used to analyse circuits at low levels of abstraction.

The choice of a lazy functional language allowed backtracking analyses to be cast elegantly as non-standard interpretations. The backtracking was performed implicitly by constructing a list of possible paths but only evaluating the portion of the list required to produce the result. Lazy evaluation is also useful for describing streams when simulating sequential circuits, since a stream is modelled by an infinite list (or a tuple of infinite lists). The interface with the windowing system also relies on lazy evaluation to function correctly. In principle, it is possible to implement the non-standard interpretation technique in any language, but lazy functional languages seem to particularly suited.

Ruby was a good choice of hardware description language because of the small number of simple, well-defined combining forms. It is very easy to translate between a Ruby description and its abstract representation. Compiling a language like VHDL would have been a more difficult task. VHDL has a very large syntax and complicated semantics and not every VHDL construct can be naturally overriden. Compactness and clear compositionality are requirements for non-standard interpretation to work correctly.

Our standard interpretation of Ruby was coded to allow Ruby descriptions to be run in one direction at a time. To make a more general Ruby interpreter, we can replace the semantics of the active nodes with set to set functions and then perform a PROLOG style analysis of the circuit representation. We could then give the values of some inputs and some outputs and compute possible values at the remaining inputs and outputs. This would be even more inefficient than the current implementation. Currently, we give all the inputs and compute the outputs or vice versa.

Non-standard interpretation can be used to build testability analysis tools. Once these have been used the next step is to transform the circuit to improve its testability. We have presented a few simple techniques for transforming some Ruby expressions to improve testability in general. These transformations are just formalisations of traditional manual techniques specialised to Ruby. Non-standard interpreters could be used to produce circuit transformers by using the existing testability analyses and adding transformation analyses.

The following list contains some possible avenues for future research.

1. The current Miranda implementation is very slow. This is partly because Miranda is interpreted. A native coded compiled version of Miranda is being produced, but we expect to recode the system in Lazy ML. This is compiled lazy functional language which has much better run-time performance. Faster implementations should also be able to support more attractive and easy to use user interfaces based around the X11 system and XView toolkit.

2. Conversion to the Haskell lazy functional language would be advantageous. This language is emerging as the standard lazy functional language, and incorporates the most recent advances in programming language theory and practice. It is also efficiently compiled to object code.

3. Another avenue for future work could involve trying to recover safety properties. It might be possible to recover safety by combining common parts of existing interpretations. The work on causal relations [Hutton 90] may be of benefit in this area.

4. Objected orientated programming has many obvious benefits for circuit simulation [Wolf 91]. For example, a simple two valued logic simulator could be extended to a three valued simulator by inheriting the common operations over the different logic systems. By thinking about how one interpretation is different and how it is similar to the standard interpretation, we can find a good application for objected oriented techniques. This would make non-standard interpretations easier to perform and specifications would look more natural. With hindsight, we notice that the non-standard interpretation system we have presented has hand-coded into it notions very similar to inheritance and type extension as well as data abstraction. However, we had to employ the features of a functional programming language (e.g. polymorphism) to emulate these characteristics of objected oriented programming. It would be an interesting piece of further work to reimplement our non-standard interpretation system in an object orientated language such as Smalltalk or C++.

5. We hope to flesh out the layout interpretations to produce output suitable for automatic layout and routing tools. These would then provide a quick route to silicon and an almost automatic translation between high level Ruby specification and CIF layout. It might be possible to perform routing and layout by using non-standard interpretation. Another analysis which could be captured as a non-standard interpretation could be design rule checking.

6. Field programmable logic devices are gaining popularity and can be used to realise complex designs. We hope to map our butterfly networks onto a field programmable logic device manufactured by Algortonix. The software used to program the Algortonix has clever routers which realise complex wiring patterns efficiently. The code used to drive the programming software will be produced by non-standard interpretation.

7. Another project under consideration is compiling Ruby to occam2 and then simulating it on a 64 transputer array manufactured by Parsytec. The transputer array can be configured arbitrarily using a crossbar switch. The translation between Ruby and occam2 can be performed easily as a non-standard interpretation. If this is successful, then this equipment could be used to perform expensive circuit calculations like test pattern generation very quickly in parallel. One important consideration is how much hardware to map to each transputer, since the number of available transputers is limited.

8. One very useful tool for Ruby would be a transformation assistant program. This would allows the user to manipulate Ruby expressions, perhaps using a structure editor. There would be a library of transformations available. As the user performed transformations, the system would show the effect each transformation had on the layout. It could also perform gate count estimates or power calculations. These additional views of the could be constructed as non-standard interpretations. One possible tool which could be used to help build this system is the Cornell Synthesiser Generator. This tool makes it easy to build structure editors. A non-standard interpretation of the decorated graph of the abstract syntax could be used to produce layouts and performance estimates.

9. The testability transformation we have suggested has been motivated by the traditional techniques currently used in industry. They are really too complex to be of any practical benefit. It is the author's opinion that a formal analysis of the hardware description should somehow give clues about how testable a circuit is and how it can be transformed to improve testability. Also, formal rules could be formulated stating how to compute the testability of a composite design when the testability of the constituent designs is known. For example, if we know how to generate test patterns for circuits A and B then we look for a way of deriving the test patterns required for A ; B without starting from scratch.

10. Although non-standard interpretation works well for analyses which have a similar 'shape' to the circuit under analysis, there are still many analyses which cannot be cast in this way. Instead of relying on an isomorphic relationship between the standard and non-standard representations, we could look for a homomorphism. This would greatly increase the type of analyses we could capture while still retaining a formal relationship between the standard and non-standard representations.

11. Currently the non-standard interpretations produced work in 'batch' mode. A circuit is submitted along with some inputs and some outputs is produced. However, many designers produce circuits incrementally, adding a bit at a time to a circuit. It is wasteful to recompute from scratch when work from a previous calculation can be used. It should be easy to add hooks into the system to provide such support. This would then allow the non-standard interpretation system be used as part of a larger incremental design system.

Appendix A

A.1 Source for Deductive Fault Simulator

```
> || deduc.m    Deductive Fault Simulation of Combinational Circuits

> %include "ruby"
> %include "standard"
> %include "~/miranda/general.lit"

> deduc_interp
>  =         [(And', and_ded, undef),
>             (Or',  or_ded, undef),
>             (Not', not_ded, undef)]

> deduc_nsi = standard deduc_interp

> not_ded (Tuple [Label n, Logic vhere]) (Tuple [FaultSet fx, vin])
>  = Tuple [FaultSet (outfault n vhere fx), Logic vhere]
> not_ded x other = error ("not_ded: " ++ show_tuple other)


> and_ded (Tuple [Label n, Logic vhere])
>         (Tuple [Tuple [FaultSet fx, Logic x],
>                 Tuple [FaultSet fy, Logic y]])
>  = Tuple [FaultSet (outfault n vhere fo), Logic vhere]
>    where
>    fo = cn vhere (intersection (cn x fx u) (cn y fy u)) u
>    u = union fx fy
> and_ded x other = error ("and_ded: " ++ show_tuple x ++ " with
input "
>                          ++ show_tuple other)

> or_ded (Tuple [Label n, Logic vhere])
>        (Tuple [Tuple [FaultSet fx, Logic x],
>                Tuple [FaultSet fy, Logic y]])
>  = Tuple [FaultSet (outfault n vhere fo), Logic vhere]
>    where
```

```
>     fo = cn vhere (union (cn x fx u) (cn y fy u)) u
>     u = union fx fy
> or_ded x other = error ("or_ded: " ++ show_tuple other)

> outfault n False fs = (SA1 n) : fs
> outfault n True fs = (SA0 n) : fs

> cn False fs u = fs
> cn True fs u = mkset (u--fs)
```

A.2 Source for SCOAP Testability Measure

A.2.1 Controllability measure

```
> %include "ruby"
> %include "standard"
> %include "~/miranda/general.lit"

> cont_interp
>  =         [(And', and_cont, undef),
>             (Or',  or_cont, undef),
>             (Not', not_cont, undef)]

> cont_nsi = standard cont_interp


> not_cont nv (Tuple [a, x])
>  = Tuple [x, Cont (c0x+1, c1x+1)]
>    where
>    Cont (c0x, c1x) = x
> not_cont x other = error ("not_cont: " ++ show_tuple x ++ " with "
++
>                         show_tuple other)


> and_cont nv (Tuple [x, y])
>  = Tuple [Tuple [snd_tuple x, snd_tuple y],
>             Cont (min [c0x, c0y]+1, c1x+c1y+1)]
>    where
>    Cont (c0x, c1x) = snd_tuple x
>    Cont (c0y, c1y) = snd_tuple y
> and_cont x other = error ("and_cont: " ++ show_tuple x ++ " with
input "
>                            ++ show_tuple other)

> or_cont nv (Tuple [x, y])
>  = Tuple [Tuple [snd_tuple x, snd_tuple y],
>           Cont (c0x+c0y+1, min [c1x, c1y]+1)]
>    where
>    Cont (c0x, c1x) = snd_tuple x
>    Cont (c0y, c1y) = snd_tuple y
> or_cont x other = error ("or_cont: " ++ show_tuple other)
```

A.2.2 Observability mesure

```
> %include "ruby"
> %include "standard"
> %include "~/miranda/general.lit"

> obsv_interp
>  =        [(And', undef, and_obsv),
>            (Or',  undef, or_obsv),
>            (Not', undef, not_obsv),
>            (Fork', undef, fork_obsv)]

> obsv_nsi = standard obsv_interp


> not_obsv (Cont (c0, c1)) (Tuple [ignore, Nr ob])
>  = Tuple [Scoap (c0, c1, ob+1), Nr (ob+1)]
> not_obsv x other = error ("not_obsv: " ++ show_tuple x ++ " with "
++
>                        show_tuple other)


> and_obsv (Tuple [Cont (c0x, c1x), Cont (c0y, c1y)])
>          (Tuple [ignore, Nr ob])
>  = Tuple [Tuple [h, Nr (ob+1+c1y)], Tuple [h, Nr (ob+1+c1x)]]
>          where
>           h = Tuple [Scoap (c0x, c1x, ob+1+c1y), Scoap (c0y, c1y,
ob+1+c1x)]
> and_obsv x other = error ("and_obsv: " ++ show_tuple x ++ " with
input "
>                           ++ show_tuple other)



> or_obsv (Tuple [Cont (c0x, c1x), Cont (c0y, c1y)])
>          (Tuple [ignore, Nr ob])
>  = Tuple [Tuple [h, Nr (ob+1+c0y)], Tuple [h, Nr (ob+1+c0x)]]
>          where
>           h = Tuple [Scoap (c0x, c1x, ob+1+c0y), Scoap (c0y, c1y,
ob+1+c0x)]
> ord_obsv x other = error ("or_obsv: " ++ show_tuple x ++ " with
input "
>                           ++ show_tuple other)

> fork_obsv v (Tuple xs)
>||     = error ("fork_obsv trace: " ++ show_tuple xs)
>     = foldl1 (min_merge_tuples (Tuple xs)) xs

> min_merge_tuples e (Tuple [x, Nr a]) (Tuple [y, Nr b])
>  = Tuple [x, Nr (min [a, b])], if ~is_num x & ~is_num y
> min_merge_tuples e (Tuple as) (Tuple bs)
>  = Tuple [min_merge_tuples e a b | (a, b) <- zip2 as bs]
> min_merge_tuples e x y
>  = error ("min_merge_tuples: e=" ++ show_tuple e ++ " x=" ++
>           show_tuple x ++ " against " ++
>           show_tuple y)

> ext_obsv (Tuple [blah, Nr d]) = Nr d
> ext_obsv (Tuple xs) = Tuple (map ext_obsv xs)
> ext_obsv other = error ("ext_obsv: " ++ show_tuple other)
```

A.3 Partial Evaluation Interpretation

```
> || Partial Evaluation Interpretation

> %include "ruby"
> %include "standard"
> %include "~/miranda/general.lit"

> partial_eval_interp
>  =         [(And', and_sym, undef),
>             (Or',  or_sym, undef),
>             (Not', not_sym, undef)]


> and_sym n (Tuple [Symbolic x, Symbolic y])
>   = Symbolic (AndSymbol x y)
> or_sym n (Tuple [Symbolic x, Symbolic y])
>   = Symbolic (OrSymbol x y)
> not_sym n (Symbolic x) = Symbolic (NotSymbol x)

> partial_eval_nsi = standard symbolic_interp

> simplify (NotSymbol (NotSymbol x)) = x
> simplify (NotSymbol (AndSymbol x y))
>  = OrSymbol (simplify (NotSymbol x)) (simplify (NotSymbol y))
> simplify (NotSymbol (OrSymbol x y))
>  = AndSymbol (simplify (NotSymbol x)) (simplify (NotSymbol y))
> simplify (AndSymbol SymbolFalse x) = SymbolFalse
> simplify (AndSymbol x SymbolFalse) = SymbolFalse
> simplify (AndSymbol SymbolTrue x) = x
> simplify (AndSymbol x SymbolTrue) = x
> simplify (OrSymbol SymbolTrue x) = SymbolTrue
> simplify (OrSymbol x SymbolTrue) = SymbolTrue
> simplify (OrSymbol SymbolFalse x) = x
> simplify (OrSymbol x SymbolFalse) = x
```

References

[Abramsky 85] Samson Abramsky. Strictness Analysis and Polymorphic Invariance. Programs as Data Objects. Springer-Verlag LNCS 217, 1985.

[Abramsky 86] Samson Abramsky. Strictness Analysis and Polymorphic Invariance. Proceedings of the DIKU Workshop on Programs as Data Objects, LNCS 217, Springer-Verlag. 1986.

[Abramsky 90] Samson Abramsky. Abstract Interpretation, Logical Relations and Kan Extensions. Not published (yet).

[Akella et al. 90] Venkatesh Akella, Ganesh Gopalakrishnan. High Level Test Generation via Process Composition. Designing Correct Circuits 90. Mary Sheeran and Geraint Jones (eds.). Springer Verlag, 1991.

[Armstrong 72] D. B. Armstrong. A Deductive Method for Simulating Faults in Large Circuits. IEEE Trans. Computers, C-21(5). 1972.

[Brackenbury 87] Linda E. M. Brackenbury. Design of VLSI Systems— A Practical Introduction. Macmillan Computer Science Series. 1987.

[Bossen 71] Douglas C. Bossen. Cause-Effect Analysis for Multiple Fault Detection in Combinational Networks. IEEE Transactions on Computers, Vol. C-20, No. 11, November 1971.

[Boute 86] Raymond T. Boute. Representational and denotational semantics of digital systems. Rep. No. 61, Dept. Computer Science, Univ. Nijmegen. January 1985.

[Boute 88] Raymond T. Boute. System Semantics: Principles, Applications, and Implementation. ACM Transactions on Programming Languages and Systems. Vol. 10 No. 1 pp.118-155. 1988.

[Burn et. al. 85] G. L. Burn, C. L. Hankin, S. Abramsky. The Theory of Strictness Analysis for Higher Order Functions. Programs as Data Objects. Springer-Verlag LNCS 217, 1985.

[Caneghem et. al. 86] Michel Van Canegham, David H. D. Warren (eds.). Logic Programming and its Applications. Ablex Series in Artifical Intelligence. 1986.

[Cohn & Gordon 86] Avra Cohn, Mike Gordon. A Mechanized Proof of Correctness of a Simple Counter. University of Cambridge Computer Laboratory Technical Report No. 94. July 1986.

[Cohn 87] Avra Cohn. A Proof of Correctness of the Viper Microprocessor: The First Level. University of Cambridge Computer Laboratory Technical Report No. 104. January 1987. Also on VLSI Specification, Verification and Synthesis. G. Birtwistle and P. Subrahmanyan eds., Kluwer Academic Publishers 1988.

[Devadas et. al. 89] S. Devadas, H.-K. T. Ma, A. R. Newtton, Sangiovanni-Vincentelli. A Synthesis and Optimization Procedure for Fully and Easily Testable Sequential Machines. IEEE Transactions on Computer-Aided Design of Integerated Circuits and Systems, Vol. 8, No. 10, October 1989.

[Davio et. al. 83] M. Davio, J.-P. Deschamps, A. Thayse. Digital Systems with Algorithm Implementation. Wiley, 1983.

[Dowd et al. 89] M. Dowd, Y. Perl, L. Rudolph, M. Saks. The periodic balanced sorting network. JACM, Vol. 36, No. 4, October 1989.

[EDIF Committee] Electronic Design Interface Format Steering Comittee. EDIF — Electronic Design Interface Format Version 1 0 0, Texas Instruments, Dallas, Texas, 1985.

[Gibson 83] J. R. Gibson. Electronic Logic Circuits. Edward Arnold. 1983.

[Goel 81] Prabhakar Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. IEEE Transactions on Computers, Vol. C-30, No. 3, March 1981.

[Goldstein 79] L. H. Goldstein. Controllability/Observability Analysis of Digital Circuits. IEEE Transactions on Circuits and Systems, Vol. CAS-26, No. 9. Sept. 1979.

[Gosling 80] John B. Gosling. Design of Arithmetic Units for Digital Computers. 1980.

[Grason 79] J. Grason. TMEAS: A Testability Measure Program. Bell Laboratories, Murray Hill, N.J. 1979.

[Hayes 71] John P. Hayes. On Realizations of Boolean Functions Requiring a Minimal or Near-Minimal Numberof Tests. IEEE Transactions on Computers, Vol. C-20, No. 12, December 1971.

[Henderson 82] Peter Henderson. Functional Geometry. Proc. ACM Symposium on Lisp and Functional Programming. August, 1982.

[Herbert 86] John Mary Joseph Herbert. Application of Formal Methods to Digitial Systems Design. PhD Thesis, Corpus Christi College, The University of Cambridge. 1986.

[Hill 86] S. A. Hill. Simulating Digital Circuits. Technical Report No. 42, Computing Laboratory, University of Kent. 1986.

[Hsieh 71] Edward P. Hsieh. Checking Experiments for Sequential Machines. IEEE Transactions on Computers, vol. C-20, No. 10, October 1971.

[Hudak et al. 85] Paul Hudak, J. Young. A set-theoretic characterisation of function strictness in the lambda calculus. Technical Report YALEU/DCS/RR-391, Yale University. 1985.

[Hughes 86] John Hughes. Stictness Detection in Non-Flat Domains. Proceedings of the DIKU Workshop on Programs as Data Objects, LNCS 217, Springer Verlag. 1986.

[Hunt 85] W. A. Hunt. FM8501: A Verified Microprocessor. PhD Thesis. Institute for Computing Science. University of Texas at Austin. 1985.

[Hutchison et. al. 87] David Hutchison, Peter Silvester. Computer Logic: Principles and Technology. Ellis Horwood Series in Computers and their Applications. 1987.

[Hutton 90] Graham Hutton. Functional Programming with Relations. Glasgow Workshop on Functional Programming. Kei Davis, John Hughes (eds). Springer Verlag Workshops in Computing. 1991.

[Ivanov et al. 89] A. Ivanov, V. K. Agarwal. An Analysis of the Probabalisitc Behaviour of Linear Feedback Signature Registers. IEEE Transactions on Computer-Adied Design of Integerated Circuits and Systems. Vol. 8, No. 10, October 1989.

[Johnson 83] Steven Dexter Johnson. Synthesis of Digital Designs from Recursive Equations. Technical Report No. 141, Indiana University Computer Science Department. 1983.

[Johnson 84] Steven Dexter Johnson. Synthesis of Digital Designs from Recursion Equations. ACM. May 1984.

[Jones et. al. 90] Geraint Jones, Mary Sheeran. The Study of Butterflies. Third Annual Glasgow Workshop on Functional Programming. Ullapool. August 1990.

[ND Jones 85] N. D. Jones, P. Sestoft, H. Sondergaard. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. Lisp and Symbolic Computation 2. 1989.

[Kahn 74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. Information Processing 74. North Holland. 1974.

[Kloos87] Delgado Kloos. Semantics of Digital Circuits. Springer-Verlag LNCS Vol. 285. 1987.

[Lala 85] Parag K. Lala. Fault Tolerant and Fault Testable Hardware. Prentice-Hall. 1985.

[Launchbury 90] John Launchbury. Lecture Notes on Domain Theory. Lectures given at Glasgow University. 1990.

[Launchbury 90] John Launchbury. Projection Factorisations in Partial Evaluation. PhD Thesis. Computing Science Dept. Report CSC 90/R. The University of Glasgow. 1990.

[Lewin 85] Douglas Lewin. Design of Logic Systems. Van Nostrand Reinhold, 1985.

[Luk 90] Wayne Luk. Analysing Parameterised Designs by Non-Standard Interpretation. Proc. Application Specific Arrays and Processors. IEEE Computer Press. 1990.

[Mano 84] M. Morris Mano. Digital Design. Prentice-Hall. 1984.

[McCluskey 62] Edward J. McCluskey, H. Schorr. Essential Multiple-Output Prime Implicants. Mathematical Theory of Automata. Proc. Polytechnic Inst. Brooklyn Symp. Vol. 12, pp. 437—457. April 1962.

[McCluskey et. al. 71] Edward J. McCluskey, Frederick W. Clegg. Fault Equivalence in Combinational Logic Networks. IEEE Transactions on Computers, Vol. C-20, No.11, November 1971.

[McCluskey 86] Edward J. McCluskey. Logic Design Principles: With Emphasis on Testable Semicustom Circuits. Prentice-Hall. 1986.

[Mead et. al. 80] Carver Mead, Lynn Conway. Introduction to VLSI Systems. Addison-Wesley. 1980.

[Meshkinpour 84] Farshad Meshkinpour. On Specification and Design of Digital Systems using an Applicative Hardware Description Language. Report No. CSd-840046, UCLA Computer Science Department. 1986.

[Melham 87] Thomas F. Melham. Abstraction Mechanisims for Hardware Verification. University of Cambridge Computer Laboratory Technical Report No. 106. January 1987.

[Melham 90] Thomas F. Melham. Formalising Abstraction Mechanisims for Hardware Verification in Higher Order Logic. PhD Thesis. University of Cambridge. August 1990.

[Milner 78] Robin Milner. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences. 1978.

[Morison 82] J.D. Morison, N.E. Peeling, T.L. Thorp. ELLA: A Hardware Description Language. RSRE 1982.

[Morrison et al. 83] J.D. Morison, N.E. Peeling, T.L. Thorp. Hardware Specification - A Use for Hardware Description Languages? RSRE 1983.

[Muehldorf et. al. 81] Eugen I. Muehldorf, Anil D. Savkar. LSI Logic Testing — An Overview. IEEE Transactions on Computers, Vol. C-30, No. 1, January 1981.

[Mukherjee 86] Introduction to nMOS and CMOS VLSI Systems Design. Prentice-Hall. 1986.

[Murphy 90] David Murphy. Type Refinement in Ruby. Glasgow Workshop on Functional Programming. Kei Davis, John Hughes (eds). Springer Verlag Workshops in Computing. 1990.

[Mycroft 83] A. Mycroft. F. Strong Nielson. Strong abstract interpretation using power domains. ICALP 1983, LNCS 154, Springer-Verlag. 1983.

[MyCroft et. al.85] Alan Mycroft, Neil D. Jones. A Relational Framework for Abstract Interpretation. Programs as Data Objects.Springer-Verlag LNCS 217, 1985.

[O'Donnell 86] John T. O'Donnell. Hardware Description with Recursion Equations. Indiana University Computer Science Department Technical Report No. 212. 1986.

[O'Donell 88] John T. O'Donell. Hydra: Hardware descriptions in a functional language using recursion equations and higher order combining forms. The Fusion of Hardware Design and Verification. ed. George Milne. North-Holland. 1988.

[Oliver et. al. 89] J. L. Oliver, F. Özgüner. Design of Concurrent Error-Detetcting Systolic Arrays Using $|g3N|_M$ Codes. IEEE Transactions on

Computer-Adied Design of Integerated Circuits and Systems, Vol. 8, No. 10, October 1989.

[Patel et al. 85] D. Patel, M. Schlag and M. Ercegovac. nuFP: An Environment for the Multi-Level Specification, Analysis, and Synthesis of Hardware Algorithms. Proceedings of IFIP Conference on Functional Programming Languages and Computer Archetecture. Springer-Verlag LNCS 201, 1985.

[Peyton Jones 87] Simon L. Peyton-Jones. The Implementation of Functional Programming Languages. Prentice-Hall. 1987.

[Roth 66] John Paul Roth. Diagnosis of Automata Failures; A calculus and a Method. IBM Journal of Research and Development, Vol. 10, No. 4. July 1966.

[Roth 80] John Paul Roth. Computer Logic and Testing. Computer Science Press. 1980.

[Rubin 87] Steven M. Rubin. Computer Aids for VLSI Design. Addison-Wesley. 1987.

[Russell et. al. 86] G. Russell (Ed.), D.J. Kinniment, E.G. Chester, M.R. McLauchlan. CAD for VLSI. Van Nostrand Reinhold. 1986.

[Schlag 84] Martine Schlag. Extracting Geometry from FP for VLSI Layout. Report No. CSD-840043, UCLA Computer Science Department. 1984.

[Sellers 68] F. F. Sellers, M. Y. Hsiao, L. W. Bearnson. Analyzing errors with the Boolean Difference. IEEE Transactions on Computing. Vol. EC-17, pp. 676-683. July 1968.

[Sheeran 83] Mary Sheeran. μFP - An Algebraic VLSI Design Language. PhD. Thesis, PRG, University of Oxford, 1983.

[Sheeran 84] Mary Sheeran. muFP, a language for VLSI design. Proc. Symposium on Lisp and Functional Programming (ACM). 1984.

[Sheeran 86] Mary Sheeran. Describing and reasoning about circuits using relations. Theoretical foundations of VLSI Design. Proc. 1986 Leeds workshop. Cambridge Tracts in Theoretical Computer Science 10. Cambridge University Press, 1990.

[Sheeran 87] Mary Sheeran. Relations + Higher Order Functions = Hardware Descriptions. University of Glasgow Computing Science Technical Report CSC/87/R1. 1987.

[Sheeran 88] Mary Sheeran. Retiming and Slowdown in Ruby. The Fusion of Hardware Design and Verification. July 1988.

[Sheeran 90a] Mary Sheeran, Geraint Jones. Circuit Design in Ruby. Lyngby Lecture Notes. 1990.

[Sheeran 90b] Mary Sheeran. Sorts of Butterflies. Proc. IVth Higher Order Workshop. Banff 1990. G. Birtwistle ed. Springer Workshops in Computing. 1991.

[Singh 89a] Satnam Singh. Implementation of a Non-Standard Interpretation System. Glasgow University Functional Programming Workshop. Springer-Verlag, 1989.

[Singh 89b] Satnam Singh. Application of Non-Standard Interpretation: Testability. Design of Correct VLSI Circuits. Proc. IFIP-IMEC Workshop, Belgium. North Holland, 1989.

[Singh 90] Satnam Singh. Differentiating Strictness. Functional Programming, Glasgow 1990. Kei Davis and John Hughes (eds). Workshops in Computing. Springer-Verlag, 1991.

[Singh 91] Satnam Singh. Circuit Layout using NSI. Advanced Research Workshop on Correct Hardware Design Methodologies. Turin, 1991. North-Holland.

[Stephenson 76] J. E. Stephenson, J. Grason. A Testability Measure for Register Transfer Level Digital Circuits. Digest 6th Int. Symp. Fault-Tolerant Computing. FTCS-6, Pittsburgh, pp.101-107 June 1976.

[Stoy 77] Joseph E. Stoy. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press in Computer Science. 1977.

[Tocci 80] Ronald J. Tocci. Digital Systems: Principles and Applications. Prentice-Hall. 1980.

[Trullemans 86] C. Trullemans (ed.). Algorithmics for VLSI. Academic Press. 1986.

[Turner 79] D. A. Turner. SASL Language Manual. University of Kent, UK. 1979.

[Turner 85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. Proceedings of the IFIP Conference on Functional Programming Languages and Computer Architecture. Springer Verlag LNCS 201, 1985.

[Varma et. el. 89] Devadas Varma, E. A. Trachtenberg. Design Automation Tools for Efficient Implementation of Logic Functions by Decomposition. IEEE Transactions on Computer-Adied Design of Integerated Circuits and Systems, Vol. 8, No. 8, August 1989.

[Wadler 85] Philip L. Wadler. Representing Failure as a List of Successes. Proceedings of the IFIP Conference on Functional Programming Languages and Computer Architecture. Springer Verlag LNCS 201, 1985.

[Wadler 87] Philip L. Wadler. Strictness Analysis on Non-Falt Domains. Abstract Interpretation of Declarative Languages. Samson Abramsky, Chris Hankin (eds.). Ellis Horwood, 1987.

[Wadler et. al. 87] Philip L. Wadler, John Hughes. Projections for Strictness Analysis. Springer Verlag LNCS 274, 1987.

[Weste 85] Neil Weste and Kamran Eshranghian. Principles of CMOS VLSI Design: A Systems Perspective. Addison-Wesley, 1985.

[Wilkins 86] B.R. Wilkins. Testing Digital Circuits. Van Nostrand Reinhold. 1986.

[Wolf 91] Wayne Wolf. Object-Oriented Programming for CAD. IEEE Design & Test. March 91.

[Wu et. al. 90] Cheng-Wen Wu, Peter Cappello. Easily Testable Iterative Logic Arrays. IEEE Trans. on Computers. Vol. 39, No. 5, May 1990.