Comparing the NDS and VERA environments for scan chain recognition & removal

M. van Balen

Concerns	: End Report
Period of work	: April 1993 - Dec 1993
Supervisor	: prof. ir. M.T.M. Segers
Advisors	: ir. F.G.M. Bouwman
	: ir. J.M.C. Jonkheid
	: ir. P.W.M. Merkus

M. van Balen

Comparing the NDS and VERA environments for scan chain recognition & removal

ABSTRACT

To cope with testing problems for large and complex logic circuits it is widely acknowledged that one has to partition the circuit into independently testable blocks and apply structural Design-For-Testability techniques. A generally adopted Design-For-Testability technique is *scan design*. In this case all, or a selected part of, the flip flops in the design are replaced by a scannable variant. These scannable variants form a shift-register, by which the design can be set in any desired state.

In this report algorithms are derived for finding and removing such *scan-chains*. We use the implementation of these algorithms as a vehicle for comparing the NDS and VERA environments, discussing (dis)advantages of both worlds.

Keywords : NDS, VERA, OutScan, scan design, design-for-testability

Preface

This work was performed in partial fulfillment of the requirements to become Master of Electrical Engineering at the Eindhoven University of Technology. It was performed at the Philips Research Laboratories in Eindhoven during the period april – december 1993.

Acknowledgements

First of all, I would like to thank my mentors Paul Merkus, Johan Jonkheid, and Frank Bouwman of the Philips ED&T group for their enthusiastic support. I also want to thank Krijn Kuiper, Steven Oostdijk, Hans Bouwmeester, and the other members of the group for their valuable help regarding my work and this report. Furthermore I thank my fellow students, Ruud van der Meer and Frank van de Voort, for the many fruitful discussions concerning our projects, and their pleasant company during my work at Philips.

Finally I want to thank prof.ir. M.T.M. Segers for being my supervising professor, and for giving me the opportunity to carry out this project in his group.

Philips Research Laboratories,

Eindhoven, December 7, 1993

Martijn van Balen

Contents

List of Figures

Chapter 1

Introduction

Testing of Integrated Circuits (ICs) has become a major issue in research and development of Very Large Scale Integration (VLSI). ICs have been growing continuously in number of transistors and complexity. This has caused an increase in the probability of both design errors and manufacturing faults. Manufacturing faults result in defects on the IC, while design errors result in an undesired functionality of the chip. To prevent design errors, the design must be checked at several stages of the design. It is not possible to prevent manufacturing faults, in fact, the yield is typically between 40 and 80 percent. Because the market asks for reliable, zero defect ICs, every single IC produced must pass several tests. One of these tests is the structure test. The idea of structure test is that all structures, created on the silicon surface, must be tested for correctness. The problem of structure testing of ICs is complicated by the enormous amount of possible faults, and the limited accessibility of parts of the ICs via the IC pins.

The requirements for a structure test are strict. The test should be fast because it is performed on every individual IC, and it must still have a high fault coverage since customers do not accept faulty ICs. Furthermore, it is important that we are able to generate this test in a short time in order to prevent lengthy design times. It is generally believed that these requirements can only be met if already during the design phase the IC testability is taken into account. This is referred to as design for testability (DFT).

There are many DFT techniques. In general, they operate by improving the controllability and observability of the circuit. This means that test patterns can be transported more easily to isolated parts of the IC and the responses can be more easily captured. Generating test patterns becomes less complex and the fault coverage increases. The DFT technique that we are concerned with in this report is scan test.

Scan test is a DFT method that improves the testability of a design drastically by creating an extra 'test' operation mode for the design. In normal mode, the design operates just as it is supposed to, according to its specification. In test mode, the memory elements will all be connected into one or more scan paths. To do this, all memory elements must be replaced by versions that are equipped with such a test mode (we will call such a version the scannable variant). A scan path is a shift register that is only active during test mode. Through this scan path test patterns can be applied to the rest of the circuit, that now only contains combinatorial logic. After applying a test pattern during normal mode, the

responses can be captured at the primary outputs and the inputs of the memory elements. The captured responses in the memory elements can then be shifted out during test mode and gathered by the tester. In this way, test pattern generation has only to be done for the combinatorial part of the circuit, thus easing the process of structure testing very much.

This report deals with the specification and implementation of FindScan and RemoveScan. FindScan identifies scan chains in a design, while RemoveScan is able to remove this extra scan functionality from a design. The algorithms have been implemented using two different environments, NDS and VERA.

This report comprises the following activities: First a global introduction to the testing of ICs is given, followed by a discussing of scan test. Then models are given to be able to describe circuits and scan chains in a formal manner. This is followed by a description of the two algorithms and their implementations. Finally the two environments are compared, discussing their (dis)advantages for scan chain recognition and removal.

Chapter 2

The design and test of integrated circuits

Before an IC can be delivered to a customer, its correctness must be determined. This implies that each individual IC must operate conform to its specifications. Since the design of ICs is done in a number of phases, it must be checked that for each phase no errors are introduced (an error meaning a difference between implementation and specification). In this chapter we will first discuss the IC design process in more detail. Secondly, we will take a look at the IC testing process, which is closely related to the design process.

2.1 IC design

In an IC design trajectory we distinguish several phases [Woudsma90]. They range from requirement specification via function specification and structure specification to finally the layout. The layout is used in the manufacturing process to make the IC. These phases are depicted on the left hand side of figure 2.1.

Below, we describe the design phases that are mentioned in figure 2.1.

Requirement specification: First an informal description of an IC will be drawn up, using a natural language, specifying the desired requirements. This specification describes both what the IC should do, formulated in behavioural terms (the function behaviour), and under which conditions (e.g., environmental and parametric constraints).

Function specification: The requirement specification is transformed into a function specification. Here one specifies what the architecture of an IC would be and which functions it should perform. Also, the necessary top level modules and their interactions are determined. This specification is formal. One could, for instance, think of a VHDL description of a design being the function specification.

Structure specification: The function specification is then transformed into a structure specification. In this phase the modules and interconnections resulting from the function specification are worked out in more detail to the level of basic building

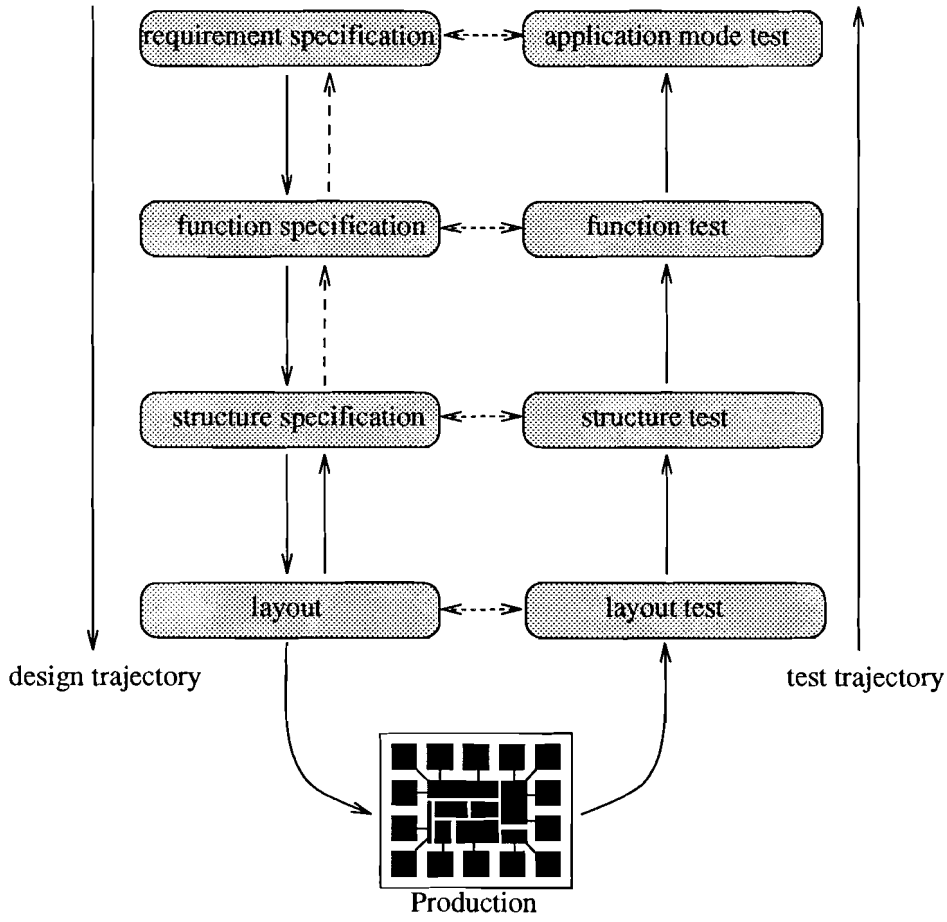

Figure 2.1: The IC design and test trajectory

blocks from the library. This models the electrical connectivity. An example of this specification is an EDIF netlist description of a design.

Layout: The structure specification is then transformed into a layout. This specifies the placement of the different building blocks, and maps these block and their interconnections to polygons, describing the topology of the chip area. An example of this specification is GDS II.

Product: The layout is used to actually manufacture the IC in a foundry.

The trajectory of creating an IC is very complex and highly error prone. Therefore, one has to take precautions to guarantee that each delivered IC conforms to its specification. To accomplish this, verification is used during the design trajectory, and testing is used during the test trajectory (after production of the IC).

To clarify the distinction between *verification* and *testing*, we will discuss them below:

Verification: Comparing the results of two successive phases with each other is called verification. It is denoted by the upwards pointing arrows in the design trajectory in figure 2.1. In this figure two verification steps are denoted by dotted arrows. These verification steps can be done, but, they must be performed by hand.

During the design trajectory, a higher-level description was transformed in a lower-level description. To verify whether this step was performed correct, the opposite action is taken. That is, given the lower-level description, a higher-level description is extracted.

This extracted higher-level description is then compared with the higher-level description used during the design trajectory. Inconsistencies indicate errors.

For instance, a composition of transistors can be transformed into a composition of logic AND and OR gates by means of verification. This can then be compared to the gate level description that was already present this level in the design stage.

The advantages of verification is that it can be done prior to the manufacturing of the chip, and more importantly it doesn't need any stimuli, hence it is exhaustive.

Testing: When the layout of a design is created, the chip can be produced. During the production process, a substantial part of the ICs will become defect. Therefore, each individual IC must be checked to see if it operates according to its specification. During testing, certain stimuli are presented to the input pins of the IC and responses at the outputs are collected and checked against the expected behaviour. An advantage of testing is that it can be done for all design stages, which can be seen in the right hand side of figure 1, where the test trajectory is depicted. Disadvantages are that testing can only be done after the IC is manufactured, and that it cannot be done exhaustively (for large ICs). This is because an exhaustive test means that we should test all possible states of an IC, and for every state we have to test all possible input combinations. For an IC with N flip flops and P inputs this means that we have to test 2^{N+P} different states. If there are only 50 flip flops and 10 inputs, testing this very small chip at 100 Mhz would already take about 350 years....

2.2 IC testing

Since IC production yields are typically between 40% and 80%, tests should be applied to every IC produced, in order to meet quality requirements. Therefore, the IC design phases are followed by extensive testing. This testing can also be divided into several phases. In general we can state that every phase in the design trajectory has an equivalent phase in the test trajectory [Beenker 90, Claasen 89], see also figure 2.1.

Each test phase has a different goal, but they all have in common that they increase the confidence one has about the IC being manufactured according to the original specification. The following test phases are depicted on the right hand sight of figure 2.1:

Layout testing: Layout testing is not done very extensively. This originates from the fact that matching a layout with the specification is practically not possible at this moment. What can be (and is) done is checking if all layout masks were correctly aligned when producing the chip. This checking can be easily done right after production by checking if certain markers, that were present on each mask, are well aligned on the chip.

Structure testing: Structure tests look for defects that result in an incorrect behaviour and are caused by the IC production process. It should be applied on every IC produced, because each single IC may contain structural faults. This is the main reason that for structure testing limitation of the test time is very important.

In the trade-off between test time and the possibility of an incorrect IC passing the test, normally a (high) number of test patterns are applied. These test patterns are generated by a test pattern generator according to certain fault models. Fault models are used to define the meaning of "fault". Many real faults can be modeled by faults defined by such a fault model, but generally they will never cover all possible faults.

Examples of fault models are the stuck-at, and the bridging-fault fault model.

Structure test (only) requires knowledge of the structure. This implies that test patterns can be generated automatically, based upon a chosen fault model.

Function testing: Test patterns for function testing are usually produced by the designer of the IC. They include tests at the critical ends of the function specification, and will normally only be applied to a few samples of the ICs produced, because structure testing should already have proven that the IC structure corresponds with the structure specification. Hence, this kind is only needed to assure that the ICs produced are in conformity with the function requirement.

Sometimes structure testing is not enough to prove the correctness of the structure. In those cases, function testing is performed on every sample, to increase the confidence one has in the testing trajectory.

Function test requires the knowledge of the function of the design. This implies that the designer most produce the test patterns.

Application mode testing: In this test the IC is installed in an application environment, or software is used to model such an environment. This test examines the

correctness of the IC design in its application and such proves that the IC is suitable for such an application.

Also, a characterisation test can now be performed. This test aims at varying the performance of the circuit under varying environmental and electrical conditions (for example temperature, voltage and humidity). During this phase the actual electrical specification of the circuit can be determined. Characterisation is also called "performance testing".

Application mode testing requires knowledge of the application. One has to know the specific application to build it, or to be able to simulate it.

The area of interest to us in this thesis is structure testing. Therefore, we look at this kind of testing more closely in the following section.

2.2.1 Structure testing

The large density of modern circuits results in an enormous amount of possible fault cases. The main problem in structure testing is the question how to detect such faults, given the limited accessibility via the IC pins.

A naive, but straightforward, strategy for structure testing is exhaustive testing. Here all possible test patterns are applied and the responses are collected. These responses can then be compared to the expected responses. The major drawback of this method is that for larger circuits the test would take so much time that it is not suitable for practical purposes. Especially for structure testing this is unacceptable, because the structure test must be applied to every IC produced.

In the case of circuits containing memory elements, i.e., sequential circuits, the problem is even worse. For these circuits the output not only depend on the input but also on the current state of the circuit. In order to test a sequential circuit exhaustively, one has to traverse all internal states of the circuit and for each state all test patterns should be applied.

The intractability of the exhaustive test strategy has led to a search for other, practically more useful, strategies. The next chapter will give an introduction to scan test, one of the strategies that can be used to ease structure testing.

Chapter 3

Scan test

The previous chapter showed that it was virtually impossible to perform exhaustive structure tests on large sequential designs. This was due to the fact that a lot of test patterns are needed to traverse all internal states of a circuit. Furthermore, the problem remained of how to access structures on the chip via the input pins. These facts result in increasing test complexity, which can be converted into costs associated with the testing process, such as the cost of test pattern generation, the cost of test equipment, and the cost related to the testing process itself, namely the time required to detect and/or isolate a fault. Because these costs can be high (and may even exceed design costs), it is important that they be kept within reasonable bounds. One way to accomplish this is by the process of design for testability (DFT).

3.1 Design for testability

Testability has been defined in the following way [Bennets 84]:

> An electronic circuit is testable if test-patterns can be generated, evaluated, and applied in such a way as to satisfy predefined levels of performance (e.g. detection, location, application) within a pry-defined cost budget and time scale.

Design for testability (DFT) can then be described as the design effort that is specifically employed to ensure that a device is testable.

There are two important attributes related to testability, namely controllability and observability. Controllability is the ability to establish a specific signal value at each node in a circuit by setting values on the circuit's inputs. Observability is the ability to determine the signal value at any node in a circuit by controlling the circuit's inputs and observing its outputs.

Structure testing mainly involves applying test patterns to the circuit that we are testing, and then observing the responses. If no specific DFT technique is used, the entire circuit can only be controlled through its input pins. It may be clear that for a large design (thousands of gates, or more) the controllability will soon become very poor, since the

number of input pins is restricted (no more than several tens or hundreds) and structures on the IC may be fairly isolated (not directly accessible from the input pins). Also, the observability will become very poor because of the same reasons and the restricted number of output pins. The poor controllability and observability makes the process of creating test patterns very difficult. This may become so hard that it is not possible anymore to get a high fault coverage within reasonable time and costs. At this point the decision must be made to accept a lower fault coverage or apply DFT techniques to increase controllability and observability. Since a lower fault coverage often is not acceptable, the choice then falls on DFT.

There are a number of DFT techniques. Most of them deal with either the re-synthesis of an existing design or the addition of extra hardware to the design. This means that they affect such factors as chip area, I/O pins, and circuit delay. Hence, a critical balance exists between the amount of DFT to use and the gain achieved. Test engineers and design engineers usually disagree about the amount of DFT hardware to include in a design. In this report, we will focus only on the DFT technique that is of importance to us, namely scan test. The remainder of this chapter contains an introduction to scan test, and discusses the arguments for and against it.

3.2 Introduction to scan test

One of the most popular structured DFT techniques is referred to as scan design [Fujiwara 85]. The classical Huffman model of a (synchronous) sequential circuit is shown in figure 3.1. In this *canonical model* the memory elements (registers) are separated from the rest of the circuit, so that the remaining part of the circuit is combinatorial. The combinatorial logic has a number of primary inputs (PI) and a number of secondary inputs (SI, the outputs of the registers). The output of the combinatorial logic consists of primary outputs (PO) and secondary outputs (inputs to the registers). Since the total circuit is sequential, testing it may be complicated if the circuit is large.

In figure 3.2 the scan version of the circuit is shown. The registers are now replaced by scan registers. A scan register is a register that can operate in two modes. In the *normal mode*, it acts as a normal register, just as the ones that were used in figure 3.1. In *test mode*, the scan register will clock its data in from the 'scan-in' input instead of its normal data input. After the clock pulse, this data will be present on the 'scan-out' output. The 'test' input determines the mode in which the scan register will operate. Hence, a scan register has three special pins associated with it besides the original pins of a register, namely the 'scan-in', the 'scan-out' and the 'test' pin.

The scan registers can now be transformed into one or more scan paths, by connecting the 'scan-out' pin of one scan register with the 'scan-in' pin of the next scan register. This means that the registers now form a shift register when put in test mode. In figure 3 this is illustrated.

Testing has now become much more easier. Since all memory elements can be easily controlled and observed via the scan path (in test mode) the inputs and outputs of scan registers can be treated as primary inputs and outputs of the combinatorial logic.

This means that we are able to shift test patterns in the scan path and apply them to

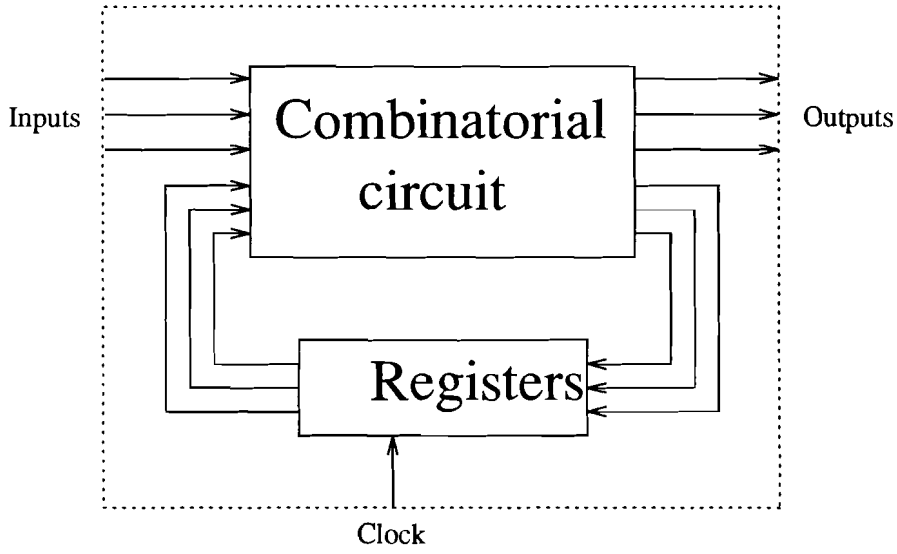

Figure 3.1: The canonical model of a circuit

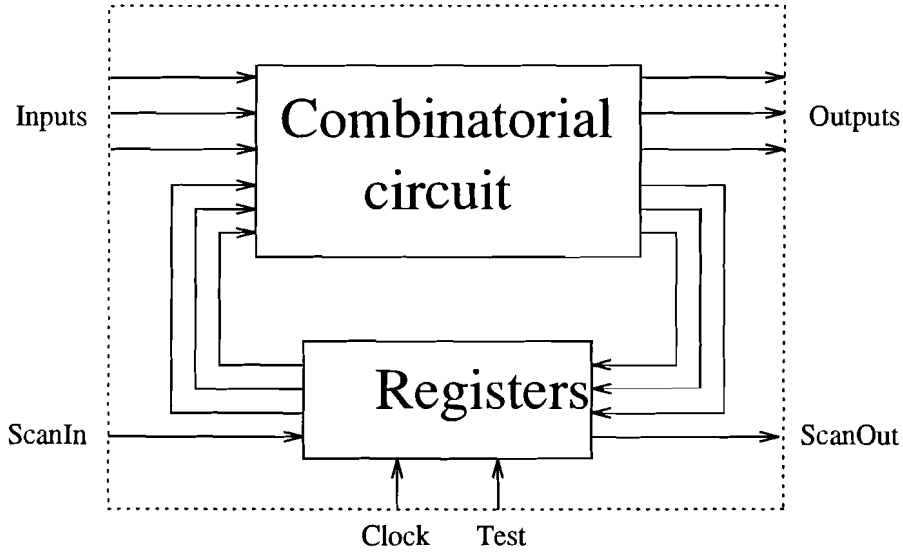

Figure 3.2: The canonical model of a scannable circuit

the combinatorial logic. The outputs of the combinatorial logic can be captured at the primary outputs of the circuit and at the inputs of the scan registers. Instead of performing a sequential test on the entire circuit, it now suffices to perform a combinatorial test on the combinatorial logic, together with a test to check that the shift register is operating correctly. These two tests are much easier and faster to generate than the sequential test. Also, the fault coverage will normally be better.

Testing a scan design in this way will be referred to as scan test in this report.

3.3 Advantages and disadvantages of scan test

One might think that scan test is the preferred solution for testing a (complex) sequential circuit when reading the previous section. This is however not always true. There are a number of costs to be observed when using scan test. [Bennets 93] gives an overview of the arguments for and against scan test. Below, we give a summary of some solid arguments used by [Bennets 93].

Advantages of scan test:

- Test pattern generation for the combinatorial parts of the circuit can be done fully automated. Furthermore, the pattern generation software will always find a test for a fault if there is one. Popular test pattern generation algorithms for combinatorial circuits include Podem [Goel 81] and Fan [Fujiwara 83].
- The fault-simulation costs are lower because the fault simulator is needed only for the combinatorial parts of the circuit. The fault coverage should be 100
- The design debug capabilities by using scan paths to explore the behaviour of the intended circuit are better.
- The design environment stays manageable because of the existence of design tools and rule checkers. Also, there is a strong belief that scan enforces well-behaved clock schemes. Such schemes can reduce timing problems in the final design. The final benefit of a controlled design environment is lower risk of a major design change (= quicker time to market).
- The ability to locate the cause of a defect has increased because of the partitioning through the scan path.

Disadvantages of scan test:

- Scan test introduces extra silicon and pins. Pins cost money, especially if the need for scan causes an increase in package size. In case pins are the most expensive, it's possible to use existing pins, in exchange for some extra controller area on the chip.
- Scan memory elements are usually enhanced versions of regular memory elements. The enhancement is normally done by adding a multiplexer function to the front end of the memory element. This extra functionality can be seen to increase propagation delays hence the potential impact on performance. There are ways to avoid this problem, but the preferred way is that the design library is enhanced to contain dedicated scan cells.

For a more complete discussion of these topics, the reader is referred to [Bennets 93].

It may be clear that the designer has to weigh these arguments against each other to decide whether or not he should use scan test for his design. In general, it can be stated that when extra silicon, pins and delays are not critical factors, there are no serious reasons why the designer should not choose for scan test. When one or more are critical, the designer should see if the problem can be worked around in some way, because the advantages of using scan test are clear. Partial scan, as discussed in [Voort 93], is a way of reducing the costs that come with scan test, and may therefore be very interesting to the designer.

Chapter 4

Hierarchical, logic circuits

In this chapter we will discuss the term *circuit*. A circuit, also referred to as *system*, or *design*, is a collection of *cells*. The behaviour of the cells is such, that the circuit performs a required task.

4.1 Cells

A *cell* can be seen as a *black box*, processing the information carried by its inputs to produce its output (see figure 4.1). These input and output connections are called the *ports* of the cell. The way in which the inputs are processed to produce the outputs is called the *behaviour* of the cell.

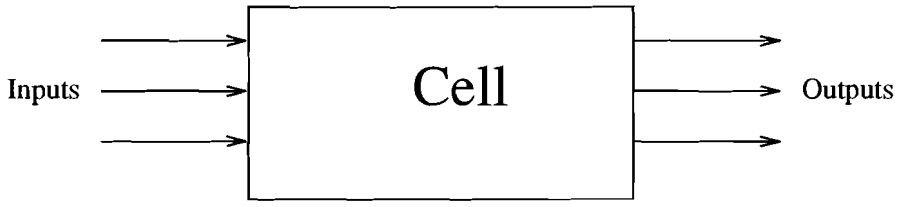

Figure 4.1: The cell as a black box

Depending on the level of abstraction, port values may be voltages, logic values, data words, etc. These values are also time dependent. In this paper we will use the logic abstraction level, i.e., ports are considered to carry (time dependent) logic values.

When timing relations are ignored, and only value transformation is considered, we speak of the *logic function* of a cell. The logic function is represented by the *functional model* of that cell. Time does play a role in such a model, but it will be abstracted.

Besides describing a cell by its function model, we can also specify the function of a cell by using a *structural model*. Use this model, a cell is a box, containing a collection of interconnected smaller boxes, called *elements*, or *children*. These elements in their turn may also be modeled by interconnection of lower-level cells. This results in a *hierarchical*

circuit, in which cells are continuously described in terms of lower-level cells, until cells are reached which are described by a function model.

When a cell C contains an element which refers to another cell C_2, we say that cell C contains an *instance* of C_2. The internal interconnection of C will specify how its elements are interconnected, and how the ports of C are connected to these elements. Since these elements actually refer to other (lower-level) cells, they do not own ports, but the cells to which they refer do. So interconnecting will be described using the *ports* of C, and the *port references* of its (lower-level) elements.

4.1.1 Leaf cells

A *leaf cell* is a cell which is not described in term of lower-level cells, but contains functionality "by definition". These *built-in* properties/attributes are logic functions, which are described using a function model.

4.1.2 Non-leaf cells

A *non-leaf cell* is a cell that does contains children, i.e., it is described using a structural model.

4.2 Ports

Ports of a (leaf or non-leaf) cell represent the interface to and from the external world. They are electrical connectors which can be connected to other ports, or to some external world source/destination. Using the direction of information flow, we distinguish four types of ports:

input ports: Information flow is always directed from a higher-level cell to lower level cell(s).

output ports: Information flow is always directed from a lower-level cell to higher level cell(s).

inoutput ports: Information flow can be directed from a higher-level cell to lower level cell(s), or vice versa.

undirected ports: Information flow direction is not specified, and must be derived by examination of the structural specification.

4.3 Nets

Nets are used to model the interconnections. All ports and port references of such a net are electrically connected, i.e., they form a "galvanic unity". At an arbitrary point in time, these ports and port references carry the same value, the value of the net.

4.3.1 Instances

One could wonder why a non-leaf cell contains elements which *refer to* other cells, instead of just *containing* those other cells. The advantage of the first approach, i.e., using instances of cells, can be clarified by looking at the following example:

Suppose a designer has to design a four-input AND, having two-input NANDs and inverters leaf-cells as building blocks. This can be done with three two-input NANDs and three inverters. The designer however will think in term of building the four-input AND with three two-input ANDs, which in their turn can be build up using a two-input NAND with an inverter. This approach is depicted in figure 4.2.

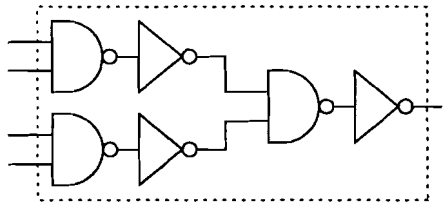

Figure 4.2: Example of a design without instances

With instances, the designer can first build an two-input AND, using a two-input NAND and an inverter. Then he can use three different *instances* of this AND, to construct the desired four-input AND. This approach is depicted in figure 4.3.

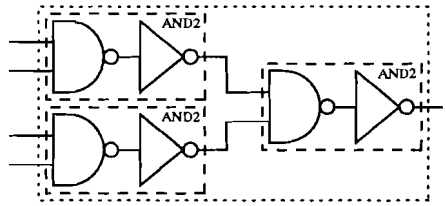

Figure 4.3: Example of a design with instances

It looks like the only thing that has changed is that the three NAND/inverter combinations have got a box drawn around them. But the major difference is the fact that the three boxes have the same name: AND2. After AND2 is constructed, we can use it to construct a higher-level cell (in this case AND4), using the leaf-cells and the AND2 cell.

If cells would contain other cells directly, every cell could be "instantiated" only once. In case of figure 4.2, every NAND used would be a separate leaf-cell, with no explicit or implicit relation to the other NANDs. They would each have their own function specification.

Another advantage of using instances in the formal structural model is the fact that the practical environments which are used, i.e., NDS and VERA, also use the instance concept. Transforming algorithms, which are written using the formal models, into an implementation is therefore rather straightforward.

4.4 Asynchronous and synchronous sequential circuit

Sequential circuits contain memory. This implies that the value of an output port at a certain time, may not only be determined by the input port values at that time, but also by the state of the memory elements. The values stored in these memory elements determine the *internal state* of the cell. Since cells are finite, they will also have a finite number of possible internal states.

Based upon the way in which the memory is realised, we distinguish *asynchronous*, and *synchronous* circuits.

4.4.1 Asynchronous circuits

An asynchronous circuit contains feedback between combinatorial inputs and outputs, shown in figure 4.4. Because of this, a change of value of an input port at an arbitrary point of time, may result in a change of the internal state of the circuit. It is even possible that by such an input value change the internal state of such a circuit will never be stable, until the input values have changed again.

Since the designing of an asynchronous circuit involves less restrictions than designing a synchronous circuit, realising a function using asynchronous circuits will almost certainly result in a smaller (never a larger) layout. The major drawback of an such a circuit however, is the fact that its behaviour heavily depends on the delay between input and output changes of cells. This not only makes analysis of these circuits much harder, but also complicates the testing process.

This report focusses on *scan chains*, which can be seen as subcircuits found in *synchronous* circuits. From now on, we will only discuss synchronous circuit in this report.

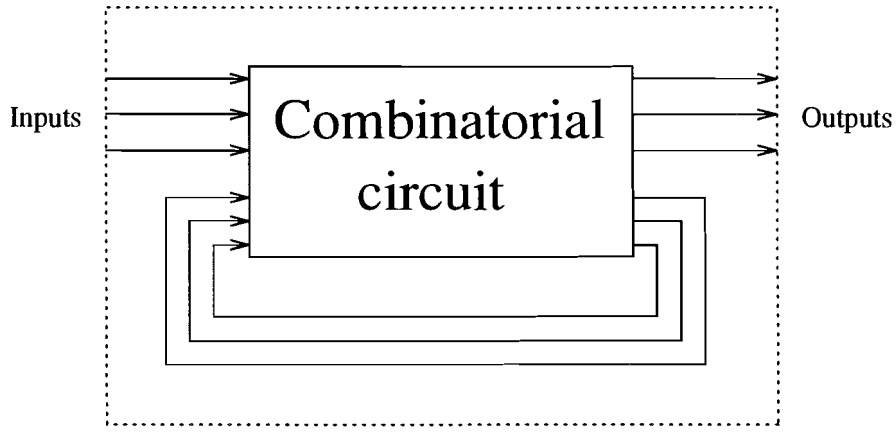

Figure 4.4: Structure of an asynchronous sequential circuit

4.4.2 Synchronous circuits

In case of a synchronous circuit special memory elements are used. Such a memory element is clocked by a synchronous clock. This clock generates so called *clock pulses*, which divide time into *time slots*, starting at $t = t_1, t_2, t_3, \ldots$ Events at time $t_1, t_2, t_3 \ldots$ are initiated by clock pulses on the clock lines.

Each time a clock pulse is received, the input values are sampled, and the next internal state and the output values are determined. Both are a function of the sampled inputs and the current state.

Because the memory elements can now be separated from the combinatorial logic, we can consider an arbitrary sequential circuit to have a *canonical structure* of the form showed in figure 4.5.

In this picture, the clock lines of the memory elements are directly connected to input ports. It is however possible that the clock lines of the children of a cell, are driven by input ports, through combinatorial logic. This logic cannot be part of the "normal logic" of the cell, since this would violate the restrictions drawn upon synchronous circuits, resulting in an asynchronous circuit.

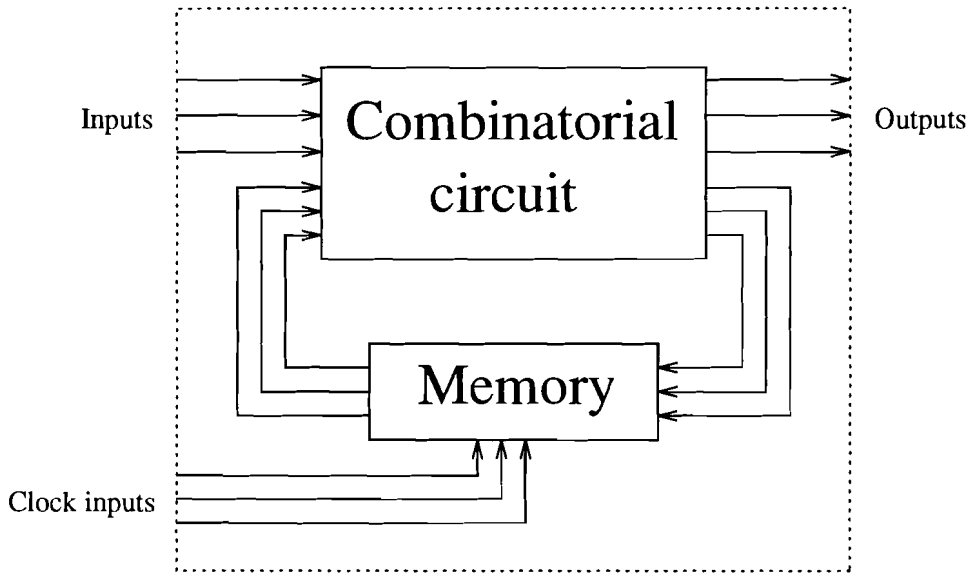

Figure 4.5: Canonical structure of a synchronous sequential circuit

4.5 Clock ports of a synchronous cell

We can distinguish several kinds of sequential devices, all clocked in a different ways:

Pulse-triggered: Input values are sampled during the '1' (or '0') period of the clock port. Output values change during this same period.

Edge-triggered: Input values are sampled just before the rising (or falling) edge of the clock port. Output values change just after this edge.

Master-Slave: Input values are sampled just before the rising (or falling) edge of the clock port. Output values change just after the next, falling (or rising), edge. Some master-slave flip flops have separate clock-ports: one for the master part, and one for the slave-part.

Pulse triggered flip flops cannot be used in scan chains, since during one active clock period, data can be propagated through more than one flipflop. With such a configuration, its impossible to put each scan flipflop into a desired state.

The other two flip flops can be used in scan chains. After sampling its input values at a clock-edge, the master-slave flipflop changes its output value(s) at the next, opposite type, edge. Since this is before the next input-sample edge, it can be modeled by an edge-triggered flipflop. In case of master-slave flip flops with separate master, and slave clock-lines, we must use the master clock-line, since it determines the input sample time.

Definition 4.1: [**clock pulse**] A *clock pulse* is a change of value on a clock port. We distinguish two kinds of clock ports, denoted by their *polarisation*:

Positively polarised: A clock pulse is a 0 to 1 transition. Events occur on the *rising edge* of this clock port.

Negatively polarised: A clock pulse is a 1 to 0 transition. Events occur on the *falling edge* of this clock port.

□

In this modeling of clock ports we have associated a positive (negative) polarity with a rising (falling) edge. This association is arbitrary chosen, we could just as well haven chosen to swap the meaning of positive and negative.

When a cell is clocked by more than one clock port, we will assume that their clock pulses will always occur at the same moments in time. We then can think of such a set of clock port as being "the clock" of that cell. Note that this is a different interpretation of multiple clock lines than the master-slave flipflop with separate master and slave clock-lines.

4.6 Functionality of cells

The functionality of a cell is in fact the collection of the functionality of all its output ports. The functionality of such a port can be described by a boolean function, describing the value on such a port at a certain point in time, depending on the input port values and the current internal state of the cell.

We can distinguish two kinds of output ports:

Combinatorial: The output port value does not depend on the internal state of the cell.

Sequential: The output port value does depend on the internal state of the cell.

4.6.1 Functionality of leaf cells

Leaf cells contain functionality "by definition", i.e., for every output port, the function is specified by a boolean function.

4.6.2 Functionality of non-leaf cells

Non-leaf cells contain instances of lower-level cells. Suppose the functionality of these lower-level cells is known. Because the references to the ports of these lower-level cells are connected to nets of the parent, the functionality of the lower-level cells specifies a relation between values on the nets of the parent cell.

Combining all relations between net-values specified by the children, using the connection between ports of the parent and those nets, result in the functional description of the parent cell. This functional description uses the union of all internal states of the children. This union can been seen as the internal state of the parent cell.

We have assumed that the functionality of the children cells was known. This is not a restriction. Since the lowest-level non-leaf cells only use instances of leaf cells, the functionality of the parent can be determined. After this, the cells which use instances of leaf cells, and instances of the just processed lowest-level non-leaf cells, can be processed. This can be continued until all cells are processed.

4.6.3 Circuit models

In the next chapter we will discuss a *structural model* for circuits. A circuit will be modeled by a set of cells, in which a cell may use instances of other cells. Leaf cells are cells which have no children, and no nets. Non-leaf cells contain children, which are interconnected by a set of nets. This model will be used to formally define the concept of "scan chains".

Chapter 5

A structural model of hierarchical circuits

In order to be able to recognise scan chains in a given circuit, we'll have to define precisely what is meant by "scan chain". Since a scan chain is a part of a circuit, we first have to define what we mean by "circuit".

In this chapter we will define a model, with which we can formally describe the structure of a circuit. Using the Edif terminology, a structural model for multiple instance, hierarchical circuits is given. This enables the reader who is familiar with Edif terms, to use an intuitive interpretation of these terms.

5.1 Ports, and the set of ports

Definition 5.1: [**set of ports**] The *set of ports* (denoted by $PORT$) is defined as the set which contains all *ports*. A port is a basic entity which will be used as a base for further definitions.

□

Definition 5.2: [**set of directions**] The *set of directions* (denoted by DIR) is defined by

$$DIR = \{input, output, inout, undirected\}$$

□

Definition 5.3: [**direction of a port**] The *direction of a port* (denoted by dir) is a relation on $PORT$, defined by

$$dir \ : \ PORT \rightarrow DIR$$

□

Definition 5.4: [**input ports**] The *set of input ports* (denoted by $IPORT$) is defined by

$$IPORT = \{p \in PORT | dir(p) \in \{input, inout\}\}$$

□

Definition 5.5: [**output ports**] The *set of output ports* (denoted by $OPORT$) is defined by

$$OPORT = \{p \in PORT | dir(p) \in \{output, inout\}\}$$

□

5.2 Instances of a set of cells

The definition of a *set of instances* is relative to a set of cells. The definition of a set of cells will be given later on, since it depends on several definitions which depend on the definition of set of cells.

Definition 5.6: [**set of instances**] Let $CELL$ be a set of cells. The *set of instances* (denoted by $INST$) is a relation on $CELL$. $INST_{CELL}$ is the set which contains all *instances* of $CELL$. An instance is a basic entity which will be used as a base for further definitions.

□

Definition 5.7: [**instance**] I is an *instance* of a set of cells $CELL$, iff $I \in INST_{CELL}$.

□

5.3 Port references of a set of cells

Definition 5.8: [**set of port references**] Let $CELL$ be a set of cells. The *set of port references* (denoted by $PORTREF_{CELL}$) is a relation on $INST_{CELL}$ is defined by

$$PORTREF_{CELL} = INST_{CELL} \times PORT,$$

with tuple element names (I, P).

□

Definition 5.9: [**port of a port reference**] Let $CELL$ be a set of cells. The *port of a port reference* (denoted by *port*) is a relation on $PORTREF_{CELL}$ defined by

$$port: \ PORTREF_{CELL} \rightarrow PORT$$

□

5.4 Nets of a set of cells

Definition 5.10: [**set of nets**] Let $CELL$ be a set of cells. The *set of nets* (denoted by NET_{CELL}) is a relation on $CELL$defined by

$$NET = \mathcal{P}(PORT \times PORTREF),$$

with tuple element names (P, PR), where:

$$\forall n \in NET: \ |n.P| + |n.PR| \in I\!N^+$$

□

Definition 5.11: [**net**] n is a *net* iff $n \in NET$

□

A net n is a tuple, containing a finite set of ports, and a finite set of port references. At least one of these sets is non-empty.

5.5 Cells, and sets of cells

In this section we will define the *declaration cell of an instance*, a *set of cells*, and the *descendent* relation(s) on a set of cells. Since these definitions are mutually dependent, we have to use forward references, i.e., references to objects which haven't been defined yet.

Definition 5.12: [**declaration cell of an instance**] Let $CELL$ be a set of cells. The *declaration cell of an instance* (denoted *cell*) is a relation on $INST_{CELL}$ defined by

$$cell: \ INST_{CELL} \rightarrow CELL,$$

with $CELL$ the set of cells, which will be defined next.

□

Definition 5.13: [**set of cells**] A *set of cells* $CELL$ is defined by

$$CELL = \mathcal{P}(PORT) \times \mathcal{P}(INST_{CELL}) \times \mathcal{P}(NET_{CELL}),$$

with tuple element names (P, I, N), where

- All ports of a cell of $CELL$ must be a member of exactly one net of that cell.

  $$\forall C \in CELL: \ (\forall p \in C.P: \ (\exists^1 n \in C.N: \ p \in n.P))$$

- All port references of a cell of $CELL$ must be a member of exactly one net of that cell.

  $$\forall C \in CELL: \ (\forall i \in C.I: \ (\forall p \in cell(i).P: \ (\exists^1 n \in C.N: \ (i,p) \in n.PR)))$$

- All ports used in a net of a cell of $CELL$ must be ports of that cell.

  $$\forall C \in CELL: \ (\forall p \in (C.N).P: \ p \in C.P)$$

- Each port reference used in a cell of $CELL$ contains a port an an instance. This port must be a port of the declaration cell of this instance.

  $$\forall C \in CELL: \ (\forall (i,p) \in (C.N).PR: \ p \in cell(i).P)$$

- All port references used in a net of a cell of $CELL$ must refer to instances of that cell.

$$\forall C \in CELL:\ (\forall (i,p) \in (C.N).PR:\ i \in C.I)$$

- Ports may only be part of one cell.

$$\forall C, C' \in CELL:\ C \neq C' \Rightarrow C.P \cap C'.P = \emptyset$$

- Instances may only be part of one cell.

$$\forall C, C' \in CELL:\ C \neq C' \Rightarrow C.I \cap C'.I = \emptyset$$

- A cell may not (indirectly) use an instance of itself. This is defined using the *descendence* relation, which will be definition next.

$$\forall C \in CELL:\ C \overset{+}{\not\sqsubseteq} C$$

□

Definition 5.14: [**cell**] C is a *cell* iff there exists a set of cells $CELL$ such that $C \in CELL$.

□

5.6 Descendence

Definition 5.15: [k^{th} **descendent,** $k \geq 0$] k^{th} *descendent* (denoted by $\overset{k}{\sqsubseteq}$) is a relation on $CELL$ defined by

$$\overset{k}{\sqsubseteq}:\ CELL \times CELL \rightarrow \mathbb{B},$$

where for $C, C'' \in CELL$:

$$\begin{array}{lcll} C'' \overset{0}{\sqsubseteq} C & \equiv & C'' = C, & \text{and,} \\ C'' \overset{k+1}{\sqsubseteq} C & \equiv & \exists C' \in CELL:\ (\exists i \in C'.I:\ C'' = cell(i)) \wedge C' \overset{k}{\sqsubseteq} C & \end{array}$$

Let $C, C' \in CELL$. If $C' \overset{1}{\sqsubseteq} C$ holds we say an instance of C' is used in cell C. If $C' \overset{n}{\sqsubseteq} C$ holds for some $n > 1$ then there exist $n-1$ other cells

$$C_1, C_2, \ldots, C_{n-1} \in CELL$$

such that

$$C' \overset{1}{\sqsubseteq} C_1 \overset{1}{\sqsubseteq} C_2 \overset{1}{\sqsubseteq} \cdots \overset{1}{\sqsubseteq} C_{n-1} \overset{1}{\sqsubseteq} C$$

□

Definition 5.16: [**descendent**] The *descendent* relation (denoted by $\overset{*}{\sqsubseteq}$) is a relation on $CELL$ defined by

$$\overset{*}{\sqsubseteq}: CELL \times CELL \rightarrow \mathbb{B},$$

where for $C, C'' \in CELL$:

$$C' \overset{*}{\sqsubseteq} C \equiv \exists k \in \mathbb{N} : C' \overset{k}{\sqsubseteq} C$$

The descendent relation is thus the reflexive and transitive closure of the k^{th} descendant relation.

□

Definition 5.17: [**true descendent**] The *true descendent* relation (denoted by $\overset{+}{\sqsubseteq}$) is a relation on $CELL$ defined by

$$\overset{+}{\sqsubseteq}: CELL \times CELL \rightarrow \mathbb{B},$$

where for $C, C'' \in CELL$:

$$C' \overset{+}{\sqsubseteq} C \equiv \exists k \in \mathbb{N}^+ : C' \overset{k}{\sqsubseteq} C$$

The true descendent relation is thus the transitive closure of the k^{th} descendant relation.

□

5.7 Descendent graph

Definition 5.18: [**descent graph**] The *descent graph* of a set of cells $CELL$ (denoted by G_{CELL}) is defined by

$$G_{CELL} = \mathcal{P}(CELL) \times \mathcal{P}(CELL \times CELL),$$

with tuple element names (V,E) (*vertices*, and *edges*), where

$$V = CELL \wedge \forall C, C' \in CELL : (C, C') \in E \equiv C' \overset{1}{\sqsubseteq} C$$

□

Theorem 5.1 *Let $CELL$ be a set of cells. The descendence graph G_{CELL} is a directed acyclic graph (DAG).*

Proof: *Suppose G_{CELL} contains a cycle, i.e. there is a $n \in \mathbb{N}$, such that*

$$C, C_1, C_2 \ldots, C_n \in G_{CELL}.V,$$

and

$$\{(C, C_1), (C_1, C_2), \ldots, (C_{n-1}, C_n), (C_n, C)\} \subseteq G_{CELL}.E$$

Hence

$$C \stackrel{+}{\sqsubseteq} C$$

But $C \stackrel{+}{\not\sqsubseteq} C$ *for each* $C \in CELL$*, so we can conclude that* G_{CELL} *is acyclic.*

Definition 5.19: [port set of a cell] Let $CELL$ be a set of cells. The *port set of a cell* (denoted by P) is a relation on $CELL$ defined by

$$P : \ CELL \rightarrow \mathcal{P}(PORT)$$

where:

$$\forall C \in CELL : \ P(C) = C.P$$

□

This relation can be used to retrieve the set of ports of a cell.

Definition 5.20: [input port set of a cell] Let $CELL$ be a set of cells. The *input port set of a cell* (denoted by IP) is a relation on $CELL$ defined by

$$IP : \ CELL \rightarrow \mathcal{P}(PORT)$$

where:

$$\forall C \in CELL : \ IP(C) = P(C) \cap IPORT$$

□

This relation can be used to retrieve the set of input ports of a cell.

Definition 5.21: [output port set of a cell] Let $CELL$ be a set of cells. The *output port set of a cell* (denoted by OP) is a relation on $CELL$ defined by

$$OP : \ CELL \rightarrow \mathcal{P}(PORT)$$

where:

$$\forall C \in CELL : \ OP(C) = P(C) \cap IPORT$$

□

This relation can be used to retrieve the set of output ports of a cell.

Definition 5.22: [port reference set of a cell] Let $CELL$ be a set of cells. The *port reference set of a cell* (denoted by PR) is a relation on $CELL$ defined by

$$PR: \ CELL \rightarrow \mathcal{P}(PORTREF)$$

where:

$$\forall C \in CELL: \ PR(C) = \{(i,p) \in PORTREF | i \in C.I \wedge p \in P(cell(i))\}$$

□

This relation can be used to retrieve the set of port references of a cell.

Definition 5.23: [input port reference set of a cell] Let $CELL$ be a set of cells. The *input port reference set of a cell* (denoted by IPR) is a relation on $CELL$ defined by

$$IPR: \ CELL \rightarrow \mathcal{P}(PORTREF)$$

where:

$$\forall C \in CELL: \ IPR(C) = PR(C) \cap \{(i,p) \in PORTREF | p \in IPORT\}$$

□

This relation can be used to retrieve the set of input port references of a cell.

Definition 5.24: [output port reference set of a cell] Let $CELL$ be a set of cells. The *output port reference set of a cell* (denoted by OPR) is a relation on $CELL$ defined by

$$OPR: \ CELL \rightarrow \mathcal{P}(PORTREF)$$

where:

$$\forall C \in CELL: \ OPR(C) = PR(C) \cap \{(i,p) \in PORTREF | p \in OPORT\}$$

□

This relation can be used to retrieve the set of output port references of a cell.

Definition 5.25: [leaf cell] Let $CELL$ be a set of cells. *Leaf cell* (denoted by $leaf$) is a relation on $CELL$ defined by

$$leaf: \ CELL \rightarrow \mathbb{B},$$

where:

$$\forall C \in CELL: \ leaf(C) \ \equiv \ C.I = \emptyset$$

A cell for which this relation holds, is called a *leaf cell.* If this relation doesn't hold for a cell its called a *non-leaf cell.*

□

Definition 5.26: [**connection**] Let $CELL$ be a set of cells, and $C \in CELL$. *Connection* (denoted by γ_C) is a relation on $P(C)$ defined by

$$\gamma_C : P(C) \times P(C) \rightarrow \mathbb{B},$$

where for $p, p' \in P(C)$:

$$\gamma_C(p,p') \equiv \exists n \in C.N : p \in n \wedge p' \in n$$

□

Definition 5.27: [**net of a port**] Let $CELL$ be a set of cells, and $C \in CELL$. *Net of a port* (denoted by net_C) is a relation on $P(C)$ defined by

$$net_C : P(C) \rightarrow NET,$$

where for $p \in P(C)$:

$$net_C(p) = \{p' \in P(C) | \gamma_C(p,p')\}$$

□

Definition 5.28: [**usage of a port**] Let C be a cell, $n \in C.N$. The *usage of a port* $p \in n.P$ (denoted by $used(p)$) is a relation on $P(C)$, defined by

$$used_C : P(C) \rightarrow \mathbb{B}$$

where for $p \in n.P$:

$$used(p) \equiv (\exists p' \in n.P : dir(p) \neq dir(p')) \vee (\exists pr \in n.PR : dir(p) = dir(pr.P))$$

So if p is an input port, $used_C(p)$ holds iff there exists a port which *reads* from the net (i.e. an output port, or an input port reference). If p is an output port, $used_C(p)$ holds iff there exists a port which *writes* to the net (i.e. an input port, or an output port reference).

□

Definition 5.29: [**usage of a port reference**] Let $CELL$ be a set of cells, $CinCELL$, and $n \in C.N$. The *usage of a port reference* $pr \in n.PR$ (denoted by $used(pr)$) is a relation on $PORTREF_{CELL}$, defined by

$$used_C : PORTREF_{CELL} \rightarrow \mathbb{B}$$

where for $pr \in n.PR$:

$$used(pr) \equiv (\exists p \in n.P : dir(p) = dir(pr.P)) \vee (\exists pr' \in n.PR : dir(pr) \neq dir(pr'))$$

So if pr is an input port reference, $used_C(pr)$ holds iff there exists a port which *writes* to the net (i.e. an input port, or an output port reference). If pr is an output port reference, $used_C(pr)$ holds iff there exists a port which *reads* from the net (i.e. an output port, or an input port reference).

□

5.8 Terminology

In the model presented here, we use terms like cell, port, etc. Unfortunately there exist many different datastructures and languages which are used to describe designs. To describe a certain construction, they use different terms for the same objects.

In this report we will use our model, the Edif, NDL, and VERA TD/ND languages, as well as the NDS/LDS datastructures. In figure 5.1 an overview is given of the terminology used by all these languages.

Formal model	Edif	VERA TD	NDL	NDS/LDS
Set of cells	Design	(the file)	(the file)	Design
Cell	Cell	Type	Macro	Declaration Block
Instance	Instance	Element	implicit	Instantiation Block
Port	Port	Terminal	implicit	Declaration Port
Port reference	PortRef	implicit	implicit	Instantiation Port
Net	Net	Node	implicit	Net

Figure 5.1: The terminology used by different languages

5.9 An example of a design

In figure 5.2 an example of a circuit is given. It can be describes using the "set of cells" model. Let $CELL$ denote this set of cells.

$CELL$ consists of seven cells:

$$CELL = \{\text{INV}, \text{NAND}, \text{MUX}, \text{DFF}, \text{BUF}, \text{SFF}, \text{COUNT}\}$$

$INST_{CELL}$ consists of eight instances. Each instance refers to a cell, denoted by the *cell* relation:

$$\begin{array}{llll} cell(inst1) = \text{SFF} & & cell(inst5) = \text{MUX} \\ cell(inst2) = \text{DFF} & & cell(inst6) = \text{SFF} \\ cell(inst3) = \text{BUF} & & cell(inst7) = \text{INV} \\ cell(inst4) = \text{NAND} & & cell(inst8) = \text{BUF} \end{array}$$

There are four leaf cells, INV, NAND, MUX and DFF:

$$\begin{array}{ll} \text{INV} & = (\{A, Q\}, \emptyset, \emptyset) \\ \text{NAND} & = (\{A, B, Q\}, \emptyset, \emptyset) \\ \text{MUX} & = (\{A, B, S, Q\}, \emptyset, \emptyset) \\ \text{DFF} & = (\{D, Cl, Q\}, \emptyset, \emptyset) \end{array}$$

The structure of the three non-leaf cells is given by:

$$
\begin{array}{lll}
\text{BUF} & = & (\ \{A, Q\}, \\
 & & \{inst7, inst8\}, \\
 & & \{ \\
 & & \ (\{A\},\ \{(inst7, A)\}), \\
 & & \ (\emptyset,\ \{(inst7, Q), (inst8, A)\}), \\
 & & \ (\{Q\},\ \{(inst8, Q)\}) \\
 & & \} \\
 & &) \\
\text{SFF} & = & (\ \{D, DT, SE, Cl, Q\}, \\
 & & \{inst5, inst6\}, \\
 & & \{ \\
 & & \ (\{D\},\ \{(inst5, B)\}), \\
 & & \ (\{DT\},\ \{(inst5, A)\}), \\
 & & \ (\{SE\},\ \{(inst5, S)\}), \\
 & & \ (\emptyset, \{(inst5, Q), (inst6, D)\}), \\
 & & \ (\{Cl\},\ \{(inst6, Cl)\}), \\
 & & \ (\{Q\},\ \{(inst6, Q)\}) \\
 & & \} \\
 & &) \\
\text{COUNT} & = & (\ \{DT, SE, Cl, Q0, Q1\}, \\
 & & \{inst1, inst2, inst3, inst4\}, \\
 & & \{ \\
 & & \ n1, n2, n3, n4, n5, n6, n7 \\
 & & \} \\
 & &)
\end{array}
$$

with

$$
\begin{array}{lll}
n1 & = & (\emptyset,\ \{(inst1, D), (inst4, Q)\}) \\
n2 & = & (\{DT\},\ \{(inst1, DT)\}) \\
n3 & = & (\{SE\},\ \{(inst1, SE)\}) \\
n4 & = & (\{Q0\},\ \{(inst1, Q), (inst2, D), (inst4, A)\}) \\
n5 & = & (\{Q1\},\ \{(inst2, Q), (inst4, B)\}) \\
n6 & = & (\{Cl\},\ \{(inst1, Cl), (inst3, A)\}) \\
n7 & = & (\emptyset,\ \{(inst2, Cl), (inst3, Q)\})
\end{array}
$$

In figure 5.3 the descendent graph of the circuit of figure 5.2 is given. The three descendence relations map to the following graph relations:

$C' \overset{k}{\sqsubseteq} C$ means: a directed path of length k exists from C to C'

$C' \overset{*}{\sqsubseteq} C$ means: $C = C'$, or a directed path exists from C to C'

$C' \overset{+}{\sqsubseteq} C$ means: a directed path exists from C to C'

From this graph it's easy to verify that, among other, the following descendent relations

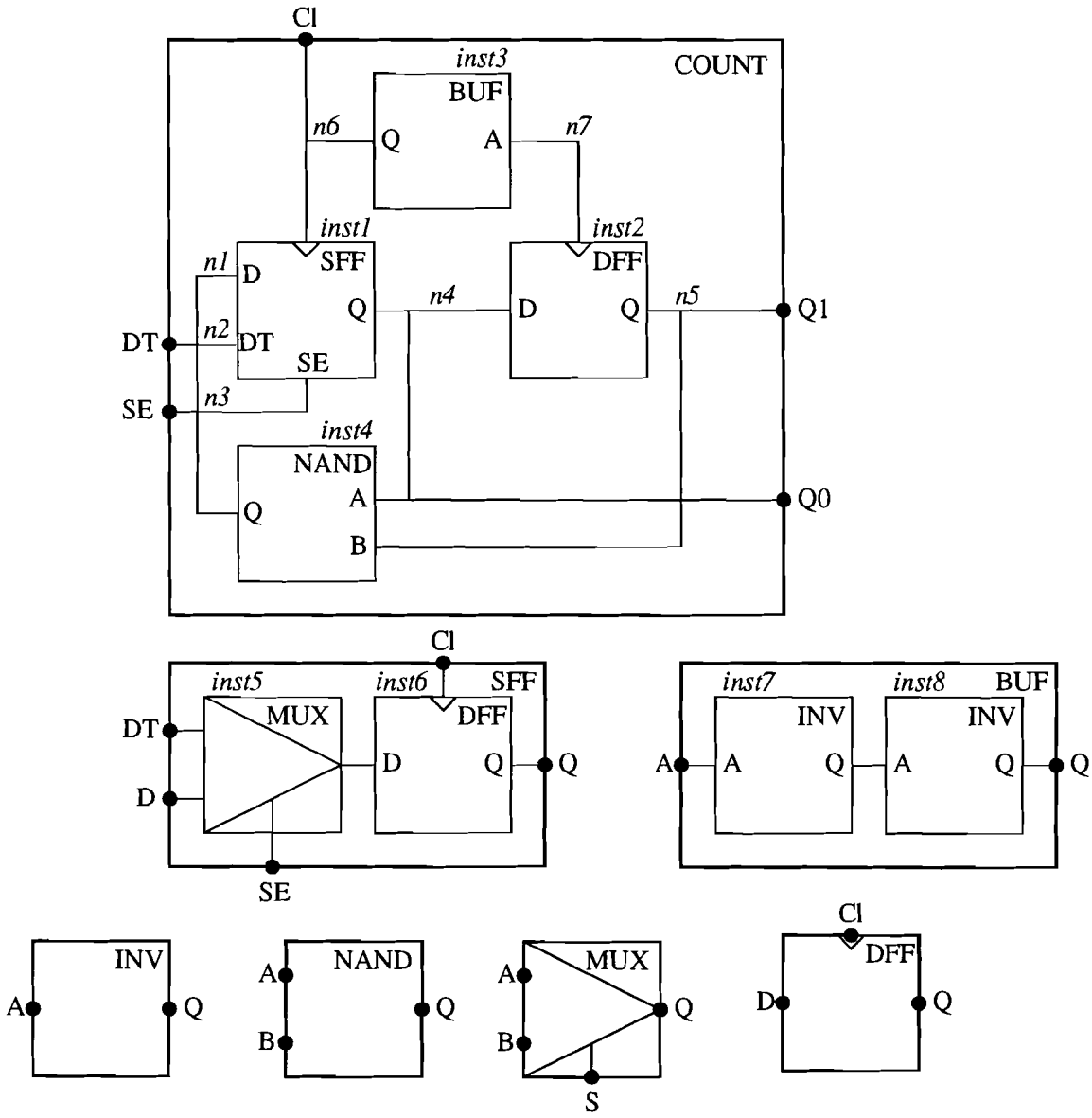

Figure 5.2: Example of a circuit: COUNT

hold:

$$INV \overset{1}{\sqsubseteq} BUF$$
$$DFF \overset{1}{\sqsubseteq} SFF$$
$$DFF \overset{2}{\sqsubseteq} COUNT$$
$$DFF \overset{+}{\sqsubseteq} COUNT$$

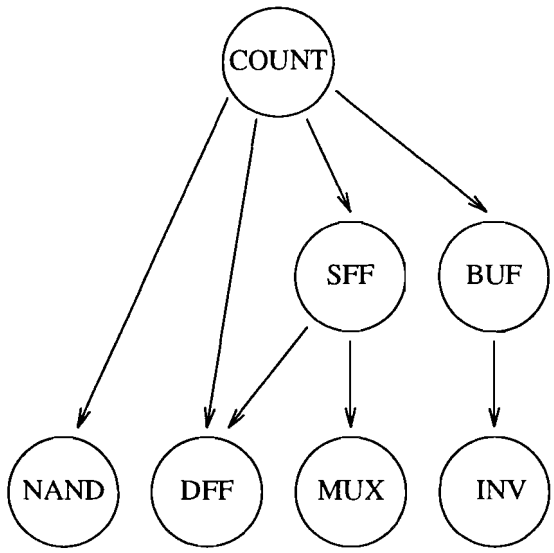

Figure 5.3: Descendent graph of circuit "COUNT"

Chapter 6

A functional model of synchronous cells

6.1 Values and time

We will associated a port, a net, or a port reference with a *value*:

Definition 6.1: [**set of values**] The *set of values* (denoted by VAL) is defined by

$$VAL = \{0, 1\}$$

□

Definition 6.2: [**value**] v is a *value* iff $v \in VAL$.

□

We only use the values '0', and '1'. This prohibits the modeling of buses, where ports must be able not to interfere with the bus to which they are connected, i.e., write the value 'Z' (high-impedant). Also wired-or, and wired-and constructions cannot be described.

Since we do not intent to describes buses, or wired-or/and constructions, we don't need the value 'Z'. Also the values 'X' (don't care) and 'U' (unknown) are not used, since we don't need them for scan chain recognition.

Definition 6.3: [**value on a port**] Let $p \in PORT$. The value carried by port p at some point in time t (denoted by $\nu(p, t)$) is defined by

$$\nu : PORT \times I\!N \rightarrow VAL$$

□

Definition 6.4: [**value on a net**] Let $CELL$ be a set of cells, and $n \in NET_{CELL}$. The value carried by net n at some point in time t (denoted by $\nu(n, t)$) is defined by

$$\nu : NET_{CELL} \times I\!N \rightarrow VAL$$

□

Time is modeled by an integer, i.e., time is divided into *time slots.* This suffices for synchronous logic circuits.

When describing combinatorial logic, an output port value only depends on the input port values, so we can omit time t, and denote the value on a port by $\nu(op)$.

6.2 Internal state of a cell

Since there will be a finite set of memory elements, a cell C will have a finite set of states in which it can be. We can therefore associate each state with a unique natural number.

Definition 6.5: [(internal) state of a cell] Let $CELL$ be a set of cells, and $C \in CELL$. The *(internal) state* of C (denoted *state*) is a relation on C and the time t defined by

$$state : CELL \times \mathbb{N} \rightarrow \mathbb{N}$$

□

Let $CELL$ be a set of cells, $C \in CELL$, and denote

$$IP(C) = \{ip_1, ip_2, \ldots, ip_{|IP(C)|}\}$$

then:

$$state(C, t+1) = f_C(state(C,t), \nu(ip_1,t), \nu(ip_2,t), \ldots, \nu(ip_{|IP(C)|},t))$$

6.3 Functionality of a cell

6.3.1 Functionality of an output port of a cell

In general, the value of an output port, in time slot $t+1$, is a function of the input port values in time slot t, and the internal state in time slot t.

Let $CELL$ be a set of cells, $C \in CELL$, and again denote

$$IP(C) = \{ip_1, ip_2, \ldots, ip_{|IP(C)|}\}$$

then:

$$\nu(op, t+1) = f_{op}(state(C,t), \nu(ip_1,t), \nu(ip_2,t), \ldots, \nu(ip_{|IP(C)|},t))$$

6.3.2 Combinatorial and sequential output ports

Definition 6.6: [combinatorial] Let $CELL$ be a set of cells, $C \in CELL$. *Combinatorial* (denoted by *comb*) is a relation on $OP(C)$ defined by

$$comb : OP(C) \rightarrow \mathbb{B}$$

where for $op \in OP(C)$:

$$comb(op) \equiv \forall st, st' \in STATE(C): f_{op}(st, \ldots) = f_{op}(st', \ldots)$$

□

This means that the output port value of a combinatorial output port does *not* depend on the internal state of its cell, only on its input port values. This implies that the response to input value changes is *not* time-dependent.

Definition 6.7: [**sequential**] Let $CELL$ be a set of cells, $C \in CELL$. *sequential* (denoted by *seq*) is a relation on $OP(C)$ defined by

$$seq: OP(C) \rightarrow \mathbb{B}$$

where for $op \in OP(C)$:

$$seq(op) \equiv \neg comb(op)$$

□

This means that the output port value of a sequential output port does depend on the internal state of its cell, *not* only on its input port values. This implies that the response to input port value changes is time-dependent.

6.4 Determining the functionality of non-leaf cells

This was already (informally) described in a previous chapter. We must transform the functions of lower-level cells into functions operating on nets of the parent, using the parents internal state. Then we solve this set of equations and rewrite them using the net_C relation.

Chapter 7

A structural model of scan chains

In this chapter we will give a definition of scan chains. All definitions are related to a set of cells. To prevent the use of several "Let $CELL$ be a set of cells", we will assume from now on that a set of cells is chosen, and will denote this set of cells by $CELL$.

Definition 7.1: [**set of enable ports**] The *set of enable ports* (denoted by $ENABLE$) is defined by

$$ENABLE = IPORT \times \mathbb{B},$$

with tuple element names (P, VAL).

□

Definition 7.2: [**enable port**] e is an enable port iff $e \in ENABLE$.

□

Definition 7.3: [**set of clock ports**] The *set of clock ports* (denoted by $CLOCK$) is defined by

$$CLOCK = IPORT \times \mathbb{B}$$

with tuple element names (P, POL).

□

Definition 7.4: [**clock port**] c is an clock port iff $e \in CLOCK$.

□

Definition 7.5: [**set of chain elements**] The *set of chain elements* (denoted by $CHAINEL$) is defined as the set which contains all *chain elements*. A chain element is a basic entity which will be used as a base for further definitions.

□

Definition 7.6: [**clock port**] el is a chain element iff $el \in CHAINEL$.

□

Definition 7.7: [**position of a chain element**] The *position of a chain element* (denoted by *pos*) is a relation on $CHAINEL$ defined by

$$pos: \ CHAINEL \rightarrow \mathbb{N},$$

□

Definition 7.8: [**instance of a chain element**] The *instance of a chain element* (denoted by *inst*) is a relation on $CHAINEL$ defined by

$$chain: \ CHAINEL \rightarrow INST_{CELL},$$

□

Definition 7.9: [**chain of a chain element**] The *chain of a chain element* (denoted by *chain*) is a relation on $CHAINEL$ defined by

$$chain: \ CHAINEL \rightarrow CHAIN,$$

where for $el \in CHAINEL$:

$$chain(el) \in CHAIN_{cell(inst(el))}$$

In this definition we've used $CHAIN_C$ (with $C \in CELL$). $CHAIN_C$ will be defined later on. The restriction states that a chain element must refer to a chain, which is part of the cell to which the instance of that chain element refers.

□

Definition 7.10: [**set of scan chains**] Let $C \in CELL$. The *set of scan chains* of C (denoted by $CHAIN_C$) is defined by

$$\begin{aligned} CHAIN_C \ = \ & IP(C) \times OP(C) \times \mathbb{B} \ \times \mathcal{P}(ENABLE) \times \mathcal{P}(CLOCK) \\ & \times \mathcal{P}(CHAINEL) \times \mathbb{N}, \end{aligned}$$

with tuple element names (I, O, INV, E, C, EL, n).

For all $CH \in CHAIN_C$ the following holds:

- $(CH.E).P \subseteq IP(C) \setminus \{CH.I\}$
- $(CH.C).P \subseteq IP(C) \setminus \{CH.I\}$
- $(CH.E).P \cap (CH.C).P = \emptyset$
- $\forall e, e' \in CH.E: \ e.P = e'.P \Rightarrow e.VAL = e'.VAL$
- $\forall c, c' \in CH.C: \ c.P = c'.P \Rightarrow c.POL = e'.POL$
- $\forall 0 \leq i < |CH.EL|: \ (\exists^1 el \in CH.EL: \ pos(el) = i)$

- Let $el \in CH.EL$. We define:

$$\begin{array}{lcl} SI(el) & = & (inst(el), chain(el).I) \\ SO(el) & = & (inst(el), chain(el).O) \\ el_i & = & \text{the element } el \in CH.EL, \text{ for which } pos(el) = i \end{array}$$

In case $CH.EL \neq \emptyset$, the following holds:

- $\gamma_C(I, SI(el_0))$
- $\gamma_C(O, SO(el_{|CH.EL|-1}))$
- $\forall 0 < i < |CH.EL| - 1 : \ \gamma_C(SO(el_{i-1}), SI(el_i))$
- $CH.INV = \bigoplus 0 \leq i < |CH.EL| : \ chain(el_i).INV$
- $CH.n = \sum 0 \leq i < |CH.EL| : \ chain(el_i).n$
- For all $cn \in CLOCKNET(CH)$:

$$(\forall cl \in CH.C : \ \nu(net_C(cl.P)) = val(cl.POL)) \Rightarrow \nu(cn.N) = val(cn.POL)$$
$$(\forall cl \in CH.C : \ \nu(net_C(cl.P)) = \overline{val(cl.POL)}) \Rightarrow \nu(cn.N) = \overline{val(cn.POL)}$$

- For all $en \in ENABLENET(CH)$:

$$(\forall en \in CH.E : \ \nu(net_C(en.P)) = \overline{en.VAL}) \Rightarrow \nu(en.N) = \overline{en.VAL}$$

In this definition we have used the set of clocknets and the set of enable nets of chain CH, $CLOCKNET(CH)$ and $ENABLENET(CH)$. These sets will be defined later.

□

Definition 7.11: [**scan chain**] *ch* is a *scan chain* iff there is a $C \in CELL$, for which $ch \in CHAIN(C)$.

□

Definition 7.12: [**set of clocknets of a chain**] Let $C \in CELL$, and $CH \in CHAIN(C)$. The *set of clocknets* of CH (denoted by $CLOCKNET(CH)$) is a relation on $CHAIN(C)$ defined by:

$$CLOCKNET : \ CHAIN(C) \rightarrow \mathcal{P}(C.N \times \mathbb{B})$$

with tuple element names (N, POL), where:

$$CLOCKNET(C) = \{(n,p) \in C.N \times \mathbb{B} | \exists el \in CH.EL : \ f_c(el, (n,p))\}$$

with

$$f_c(el, cn) \ \equiv \ \exists c \in chain(el).C : \ (inst(el), c.P) \in (cn.N).PR \wedge c.POL = cn.POL$$

□

Definition 7.13: [**set of enablenets of a chain**] Let $C \in CELL$, and $CH \in CHAIN(C)$. The *set of enablenets* of CH (denoted by $ENABLENET(CH)$) is a relation on $CHAIN(C)$ defined by:

$$ENABLENET : \ CHAIN(C) \rightarrow \mathcal{P}(C.N \times VAL)$$

with tuple element names (N, VAL), where:

$$ENABLENET(C) = \{(n, v) \in C.N \times \mathbb{B} | \exists el \in CH.EL : \ f_e(el, (n, v))\}$$

with

$$f_e(el, en) \ \equiv \ \exists e \in chain(el).E : \ (inst(el), e.P) \in (en.N).PR \wedge e.VAL = en.VAL$$

□

A scan chain ch of cell C consists of the following:

- $ch.I$: the scan input port
- $ch.O$: the scan output port
- $ch.INV$: a boolean, $false$ indicates a non-inverting chain, $true$ indicates an inverting chain.
- $ch.E$: the set of enable ports, with their enable value
- $ch.C$: the set of clock ports, with their relative polarity
- $ch.EL$: the set of chain elements, which concatenated form the chain ch. In case this set is empty,
- $ch.n$: the length of this chain, specifying the number of clock cycles needed to shift data from the scan input to the scan output.

Definition 7.14: [leaf scan chain] Let $C \in CELL$. *Leaf scan chain* (denoted by $leafchain_C$) is a relation on $CHAIN(C)$ defined by

$$leafchain_C : \ CHAIN(C) \rightarrow \mathbb{B}$$

where:

$$\forall CH \in CHAIN(C) : \ leafchain_C(CH) \equiv CH.EL = \emptyset$$

□

7.1 Scan chain definition and reality

The theory is suitable to described scan chains which contain only inverters and/or buffers in their enable- or clock lines.

Chapter 8

FindScan: the algorithm

In this chapter we will give a description of FindScan, an algorithm capable of locating scan chain in a cell. In this chapter we will assume that $CELL$ is a set of cells.

Scan chains of leaf cells are specified by definition, they can be considered already "calculated". When the chains of an arbitrary cell $C \in CELL$ should be calculated, we first have to calculate the chains of its children. This can be visualised by inspection of the decendence graph of a set of cells.

In figure 8.1 a decendence graph of a set of cells is given. From this picture we conclude that cells A,B,C,D,E,F are leaf cells (they do not instantiate other cells). Therefore the scan chains of cell G, H or J can be calculated. In order to calculate the chains of cell K, we have to process cells G and H first. Since these restrictions only state the order in which pairs of cells should be processed, there remain several valid processing sequences.

If the cells are processed in such an order, that it complies with the above restriction, we can assume the following, without loss of generality: when processing cell $C \in CELL$, the chains of its children are know.

8.1 Restrictions on clock ports

Let $C \in CELL$ be a cell, and $CH \in CHAIN(C)$ be a non-leaf chain. This implies the following: for all $cn \in CLOCKNET(CH)$:

$$\begin{array}{lll} (\forall c \in CH.C:\ \nu(net_C(c.P)) = val(c.POL)) & \Rightarrow & \nu(cn.N) = val(cn.POL) \\ (\forall c \in CH.C:\ \nu(net_C(c.P)) = \overline{val(c.POL)}) & \Rightarrow & \nu(cn.N) = \overline{val(cn.POL)} \end{array}$$

Several (sub)circuits can be though of which satisfy these relations. The most obvious one is to use inverters and buffers to establish the restriction. To determine the clock ports of a chain, given its elements, our algorithm traverses chains of inverters and buffers.

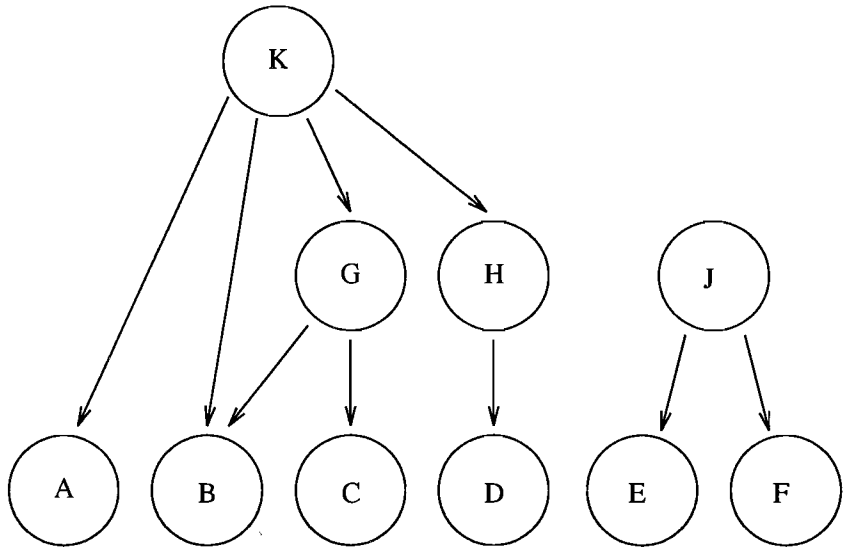

Figure 8.1: A descendence graph

8.2 Restrictions on enable ports

Let $C \in CELL$ be a cell, and $CH \in CHAIN(C)$ be a non-leaf chain. This implies the following: for all $en \in ENABLENET(CH)$:

$$(\forall e \in CH.E : \ \nu(net_C(e.P)) = e.VAL) \Rightarrow \nu(en.N) = en.VAL)$$

In this case also, several (sub)circuits can be though of which satisfy this relation. Again we have restricted the algorithm to recognition of inverter/buffer chains.

Definition 8.1: [**subchain**] Let C be a cell. The *set of subchains* of C (denoted by $SUBCHAIN(C)$) is defined by

$$SUBCHAIN(C) = C.N \times C.N \times \mathbb{B} \times \mathcal{P}(ENABLE) \times \mathcal{P}(CLOCK) \times \mathcal{P}(CHAINEL) \times \mathbb{N}$$

with tuple element names (I, O, INV, E, C, EL, n).

All restrictions drawn upon a *scan chain* apply to a subchain.

□

However, is this case $I = O$ is possible, this in the so-called *null-chain*:

$$(n, n, false, \emptyset, \emptyset, \emptyset, 0), \ \ n \in C.N$$

The null-chain merely consists of one single net. Its stands for the obvious statement: "when net n carries a value at time t, net n carries that value at time t".

Definition 8.2: [**set of scanchains of a subchain**] Let C be a cell, and $SC \in SUBCHAIN(C)$. The *set of scanchains of a subchain* (denoted by $scanchain_C$) is defined

by

$$scanchain_C : SUBCHAIN(C) \rightarrow \mathcal{P}(CHAIN(C))$$

where:

$$scanchain_C(SC) = \{CH \in CHAIN(C) | \; CH.I \in (SC.I).P \wedge CH.O \in (SC.O).P \wedge CH.E = SC.E \wedge CH.C = SC.C \wedge CH.EL = SC.EL \wedge CH.n = SC.n\}$$

□

Thus the set $scanchain_C(CH)$ is not empty iff net $SC.I$ contains an input port of C, and net $SC.O$ contains an output port of C. It will contain more than one chain if net $SC.O$ contains more than one output port.

In figure 8.2 the FindScan algorithm is given. Knowing the scan chains of the children of a cell C, the chains of C itself will be determined.

8.3 Example of a FindScan session

In figure 8.3 an example cell is given.

Cell C contains six instances: two inverters (INV), two multiplexers (MUX), one D flipflop (DFF), and one scan flipflop (SFF). All of these children contain scan chains:

C	$CHAIN(C)$
INV	$\{(\mathrm{A}, \mathrm{Q}, true, \emptyset, \emptyset, \emptyset, 0)\}$
MUX	$\{(\mathrm{A}, \mathrm{Q}, false, \{(S, 0)\}, \emptyset, \emptyset, 0), (\mathrm{B}, \mathrm{Q}, false, \{(S, 1)\}, \emptyset, \emptyset, 0)\}$
DFF	$\{(\mathrm{D}, \mathrm{Q}, false, \emptyset, \{Cl, true\}, \emptyset, 1)\}$
SFF	$\{(\mathrm{DT}, \mathrm{Q}, false, \{(SE, 1)\}, \{Cl, true\}, \emptyset, 1)\}$

When FindScan is executed on cell C, it starts at the net of an output port, in this case net $n12$, which is connected to port Q. FindScan starts with the following null-chain:

$$(n12, n12, false, \emptyset, \emptyset, \emptyset, 0)$$

1. Net $n12$ is driven by $(inst4, \mathrm{Q})$, where $inst4$ is a SFF. Cell SFF has one chain which ends at Q:

   $$(\mathrm{DT}, \mathrm{Q}, false, \{(SE, 1)\}, \{Cl, true\}, \emptyset, 1)$$

   Since $(inst4, Cl)$ is driven by port Cl, and port $en2$ drives $(inst4, SE)$ through an inverter, the subchain now becomes:

   $$(n10, n12, false, \{(en2, 0)\}, \{Cl, true\}, \{el1\}, 1)$$

```
Concatenate(S, s̃)
{
    let S' ← ∅
    forall s ∈ S
    {
        let EL ← {el ∈ CHAINEL | ∃el' ∈ s.EL : chain(el) = chain(el')∧
                pos(el) = pos(el') + 1}
        let ẼL ← {el ∈ CHAINEL | chain(el) = chain(s̃.EL), inst(el) = inst(s̃.EL)∧
                pos(el) = |EL| + 1}
        S' ← S' ∪ (s.I, s̃.O, s.INV ⊕ s̃.INV, s.E ∪ s̃.E, s.C ∪ s̃.C, EL ∪ ẼL, s.n + s̃.n)
    }
    return S'
}

Subchains(N_i, n_o)
{
    if n_o ∈ N_i
        return {(n_o, n_o, false, ∅, ∅, ∅, 0)}
    else
    {
        let pr ∈ n_o.PR, such that pr ∈ OPR(C)
        let S ← ∅
        forall s ∈ {ch ∈ CHAIN(cell(pr.I)) | ch.O = pr.P}
        {
            let S' ← Subchains(N_i, net_C((pr.I, s.I)))
            if IsCompatible(S', s)
                S ← S ∪ Concatenate(S', s)
        }
    }
    return S
}

FindScan(C)
{
    let S ← ∅
    let N_i = {net_C(ip) | ip ∈ IP(C)}
    forall op ∈ OP(C)
        forall s ∈ Subchains(N_i, net_C(op))
            S = S ∪ scanchain_C(s)
    return S
}
```

Figure 8.2: The FindScan algorithm

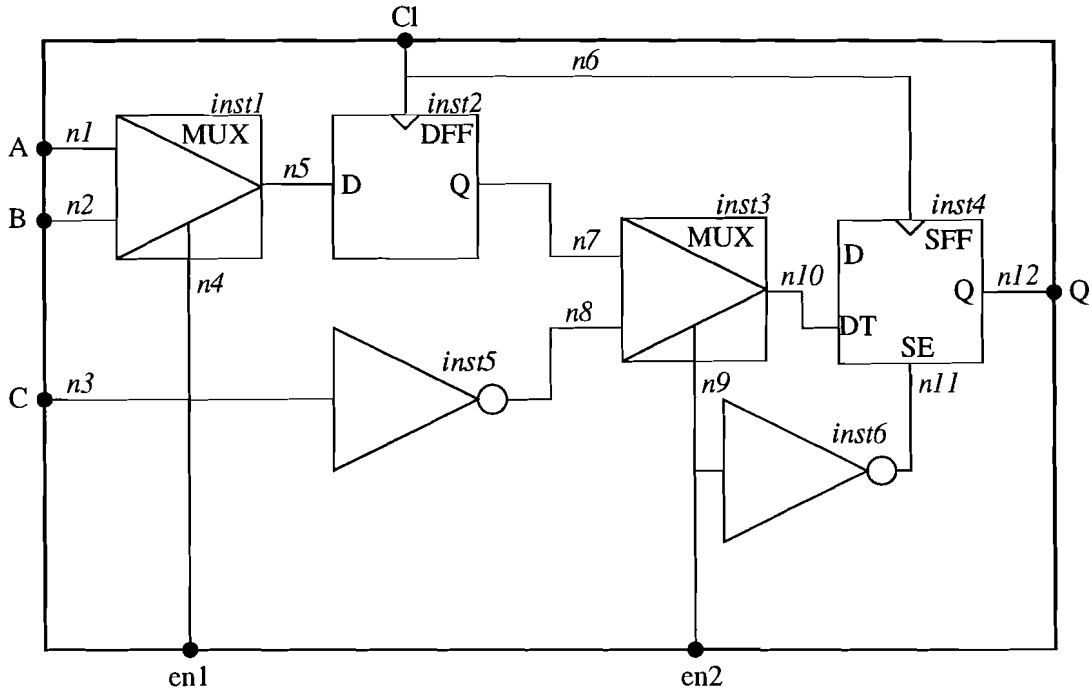

Figure 8.3: A cell containing scan chains

2. Net $n10$ is driven by $(inst3, Q)$, where $inst3$ is a MUX. Cell MUX has two chains ending at Q:

   $$(\mathrm{A}, \mathrm{Q}, false, \{(S,0)\}, \emptyset, \emptyset, 0), \text{ and } (\mathrm{B}, \mathrm{Q}, false, \{(S,1)\}, \emptyset, \emptyset, 0)$$

   $(inst3, S)$ is driven by $en2$. Extension of our subchain with the first chain results in:

   $$(n7, n12, false, \{(en2,0)\}, \{Cl, true\}, \{el1, el2\}, 1)$$

   Extension with the second chain of $inst3$ is not possible. This would result in something like

   $$(n8, n12, false, \{(en2,0),(en2,1)\}, \{Cl, true\}, \{el1, el2\}, 1)$$

   Which contains a set of enable ports, which cannot be satisfied all at the same time.

3. Extension by the chain through $inst2$ results in

   $$(n5, n12, false, \{(en2,0)\}, \{Cl, true\}, \{el1, el2, el3\}, 1)$$

4. Finally, extension of this subchain by the chains through $inst1$, results in two subchain:

   $$(n1, n12, false, \{(en1,0),(en2,0)\}, \{Cl, true\}, \{el1_a, el2, el3, el4\}, 1)$$
   $$(n2, n12, false, \{(en1,1),(en2,0)\}, \{Cl, true\}, \{el1_b, el2, el3, el4\}, 1)$$

These last subchains both start at a net which is driven by an input port of C, and they both end at a net which is connected to an output port of C. This implies they correspond to scan chains of C:

$$\begin{aligned} CH1 &= (A, Q, false, \{(en1, 0), (en2, 0)\}, \{Cl, true\}, \{el1_a, el2, el3, el4\}, 1) \\ CH2 &= (B, Q, false, \{(en1, 1), (en2, 0)\}, \{Cl, true\}, \{el1_b, el2, el3, el4\}, 1) \end{aligned}$$

with

$$\begin{array}{llllll} chain(el1_a) & = & MUX_{ch1} & inst(el1_a) & = & inst1 \\ chain(el1_b) & = & MUX_{ch2} & inst(el1_a) & = & inst1 \\ chain(el2) & = & DFF_{ch1} & inst(el2) & = & inst2 \\ chain(el3) & = & MUX_{ch1} & inst(el13) & = & inst3 \\ chain(el4) & = & SFF_{ch1} & inst(el4) & = & inst4 \end{array}$$

All possible concatenations of chain elements ending at net $n12$ have been checked, so it can be concluded that:

$$CHAIN(C) = \{CH1, CH2\}$$

Chapter 9

RemoveScan: the algorithm

RemoveScan operates on the same set of cells as FindScan. Its task is to remove the scan chains from the set of cells. By removing we mean removal of functionality which was only present for the scan chains, not the normal functionality. We will not alter cells already in the set of cells. When chains have to be removed from a cell $C \in CELL$, we will generate a cell C', performing the same task as C, but without the scan functionality.

A scan-chain should only be removed when it is used in a higher-level scan chain. So actually scan chain elements are removed (by replacing the associated instances), which implies the parent chain will be removed. When a cell C' uses an instance of cell C, with C containing n chains, only those chains which are used in a chain of C' must be removed (by replacing that instance),

For each leaf cell, we must *define* another cell which replaces it when a set of its chains has to be removed. For non-leaf cells this replacement can then be constructed.

9.1 RemoveScan for leaf scan chains

Let $C \in CELL$ be a leaf cell. Suppose $CHAIN(C)$ contains n scan chains. The interconnection of each instance of C determines which of these chains is used as a chain element in a higher lever chain. There are thus 2^n possible replacement cells for C.

An example is an n-input multiplexer. Such a multiplexer contains n scan chains by definition, selected by $n_s = \lceil \log_2 n \rceil$ selection inputs.

9.1.1 Examples of leaf cells

Inverter (INV), Buffer (BUF), and D-flipflop (DFF)

These cells all contain one chain. Because these chains in fact overlap with the normal functionality of the cell, replacement is not necessary. The fact that these cells don't have enable pins also implies this. One could say that these cells replace themselfs.

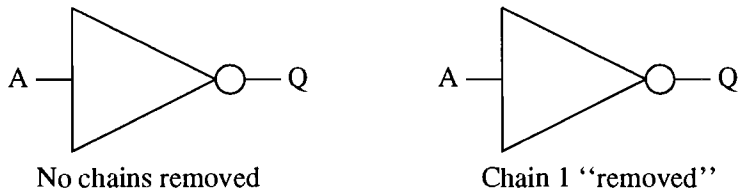

Figure 9.1: Replacement of an inverter

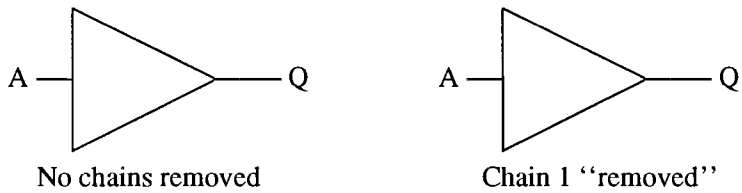

Figure 9.2: Replacement of a buffer

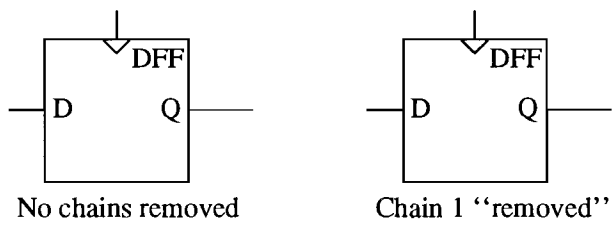

Figure 9.3: Replacement of a D flipflop

Two-input multiplexer

Now consider the case of a two-input multiplexer. Removing one chain results in a cell in which the other data input is directly connected to the output, with the selection input removed. Removing both chain will result in an empty cell.

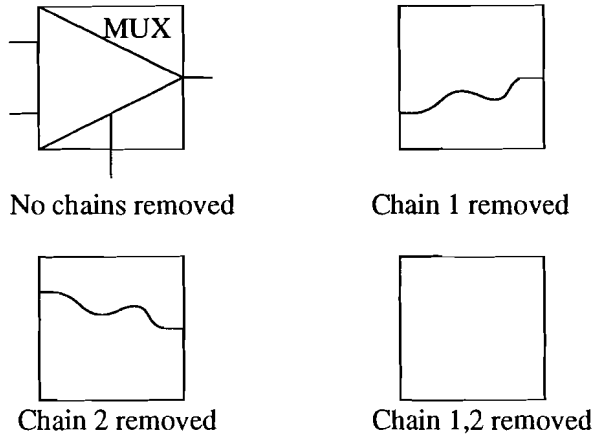

Figure 9.4: Replacement of a multiplexer

Scannable flipflop (SFF)

The scannable flipflop is a concatenation of a two-input multiplexer, and a D-flipflop. Only one of the two inputs of the multiplexer is allowed in a scan chain. Therefore there are two possibilities, either the chain is removed, resulting in a D-flipflop, or it isn't. In figure 9.5 these two possibilities are depicted.

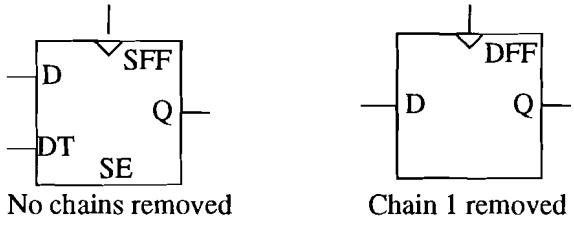

Figure 9.5: Replacement of a scannable D flipflop

9.1.2 Automatic match-rule generation for combinatorial cells

Let C be a cell, owning a combinatorial output port op, depending on input ports $ip_1, ip_2, \ldots ip_n$. This dependence can be denoted by a boolean function:

$$f_{op}(ip_1, ip_2, \ldots, ip_n)$$

The input ports include possible scan-input ports, as well as enable ports.

FindScan reports which scan chains of an instance of C are part of a higher-level scan chain. Removing these chains means simplifying the circuit, given the fact that the enable-values of scan-enable ports will never occur. Thus for the set of certain vectors of the form

$$(ip_1, ip_2, \ldots, ip_n),$$

which all enable one of the chains, the output value is a don't care. This can be used to simplify the expression.

Lets look for example at a two-input multiplexer, named MUX, with inputs A, B, selection port S, and output Q. The function f_Q is:

$$f_Q = A\overline{S} + BS$$

Using a truth-table, the function looks like:

A	B	S	Q
0	X	0	0
1	X	0	1
X	0	1	0
X	1	1	1

Suppose only the chain from A to Q is used in a higher-level chain. The truth-table for the scan-free variant then becomes:

A	B	S	Q
0	X	0	X
1	X	0	X
X	0	1	0
X	1	1	1

This table can be reduced to

A	B	S	Q
X	X	0	X
X	0	1	0
X	1	1	1

Which can be realised by the function

$$f_Q = B$$

The multiplexer is thus replaced by a cell, containing one wire which connects B and Q. Input ports A and S are not used. We name it MUX'.

Let C be a cell containing I, an instance of MUX. Let C' be the same cell, but with I replaced by an instance of MUX'. Because MUX' does not use port A and S, they can be deleted from their nets in C'. This may lead to nets with only one port or one port reference. The algorithm of figure 9.6 simplifies a cell C, by using this information about nets.

$IsUsed(C, I) \equiv$ **exists** $p \in OP(cell(I))$ **such that** $used_C((I, p))$

$RemoveInstance(C, I)$
{
    $C.I \leftarrow C.I \setminus \{I\}$
    forall $n \in C.N$
    {
        $n.PR \leftarrow n.PR \setminus \{(I, p) \mid p \in PORT\}$
        if $n = (\emptyset, \emptyset)$
            $C.N \leftarrow C.N \setminus \{n\}$
    }
}

$SimplifyCell(C)$
{
    while exists $I \in C.I$ **such that** ¬IsUsed(I)
    {
        forall $I \in C.I$ **such that** ¬IsUsed(I)
            $RemoveInstance(C, I)$
    }
    return S
}

Figure 9.6: The RemoveScan algorithm

Chapter 10

The NDS environment

NDS stands for "Network Data Structure". It comprises storage of network data, and several functions to operate on the data. Data structures and functions are written using C++. The NDS user manual [Philips ED&T 93] states:

> NDS, the Network Data Structure, provides the program developer with a general means for the storage of network data and a set of functions that operate on that data. To facilitate the use of NDS, include and library files are provided, besides this document.
>
> In principle, NDS is netlist language independent. Moreover, it is very apt to use NDS to write netlist format conversion software. However, an NDL reader and writer are provided within NDS for convenience. They are small and fast pieces of software handling a very compact and readable language. An Edif reader is provided separately.
>
> NDS is written in C++, it is assumed that the application developer is familiar with the concepts of C++...

10.1 NDS classes

Being based upon C++, NDS uses the *class* concept. A class is a description of structure and behaviour of a something. An *object* is an instantiation of a class: a thing with the structure and behaviour as described in the class.

There are six classes which are the backbone of NDS. Four of them are the basic building blocks that convey network structure information. Two additional classes allow to add other information about the network.

The basic classes are:

Design: This is a class that groups blocks together. It is the NDS equivalent of our formal "set of cells".

Block: This class is used to model our formal cells, as well as our formal instances of cells. Therefore there are two kinds of block: declaration blocks (our cells) and

instantiation block (our instances).

Pin: The Pin class represents our formal ports, as well as our formal port references. Therefore there are two kinds of pins: declaration pins (our ports) and instantiation pins (our port references).

Net: This class represents connection of ports, being declaration ports of a cell, and/or instantion pins of that cells' children. It represents our formal net.

The additional classes are:

Property: Almost all NDS objects can have one or more properties attached to them. These properties may contain extra information, which could not be specified using the structural description, for example the fan-out of an output port.

Pin2Pin: This object refers to two pins, one being the FromPin, the other the ToPin. Attaching properties to such an object can be used to specify inter-pin relation, e.g., delay.

10.2 LDS classes

LDS stands for "Library Data Structure". Just like NDS it is written in C++, and consists of classes with functions to operate on a library and its elements. In fact, the LDS classes are derived from the NDS classes, using the C++ inheritance feature.

There are four classes within LDS, all of which are derived from NDS classes:

Library: A collections of cells. It is derived from an NDS design.

Cell: A design block having ports attached to it. An LDS cell represents our formal leaf cell.

Port: A port of a cell having characteristics. Although an LDS port is named differently from an NDS pin, they both represent our formal port.

Port2Port: The combination of two ports, in a directed way.

LDS provides the user with many functions calls, which can be used to retrieve functional, and behavioural information about an LDS object.

Chapter 11

SDS

SDS stands for "Scan Data Structure". It was developed to be able to represent scan chains, and to be able to manipulate these scan chains. SDS is a set of C++ classes, and accompanying functions, all derived from the NDS class "Network". This was done in order to profit from the already implemented data structures which are implemented inside NDS (e.g. linked list). Another advantage is that the developer using SDS can, like when using NDS, attach properties to SDS objects.

SDS is a separate toolkit however, which can perfectly be used to write scanchain manipulating tools. It has its own representation language: RPL, the Routing Plan Language.

Because scan chains *are* related to a design, SDS contains function calls which can handle references of SDS objects to NDS objects, i.e., scan chain object can be related to design object. Ones these relation have been established, function calls exist such as "given this NDS declaration Block, what is the list of scan chains" (returning an SDS Macro).

11.1 SDS classes

SDS consists of seven different classes:

RoutingPlan: This is a class that groups Macros together. This class is the entry point for an SDS structure. Only via such an object can an SDS structure be accessed. It is related to the NDS Design class.

Macro: A macro is related to an NDS block, or LDS cell. It contains all scan chains through that block/cell. In our formal model we speak of a set of chains of a cell.

ClockDomain: A clockdomain groups chains of a cell together. Two chains are located in the same clockdomain, if they are clocked by the same set of clock pins.

ClockPin: This class models the combination: "pin with a polarity". It is an NDS Pin, extended with a polarity. In our formal model this was called a clock port.

Chain: This class represents scan chains. In case of a higher-level chain, its elements are specified. This represents our formal scan chain.

ChainPin: This class is almost identical to ClockPin. It consists of an NDS Pin, extended with a value. It represents the enable port of our formal model. It is also used to represent the scan-out port, in combination with the inversion property of the chain.

ChainElement: This class models a piece of scan chain. It is used to describe hierarchy found in a scan chain, and refers to a lower-level Chain. It corresponds with our formal chain element.

11.1.1 The SDS object hierarchy

In figure 11.1 the hierarchy of an SDS structure is depicted. Each arrow denoted that the above object owns a set of the object below.

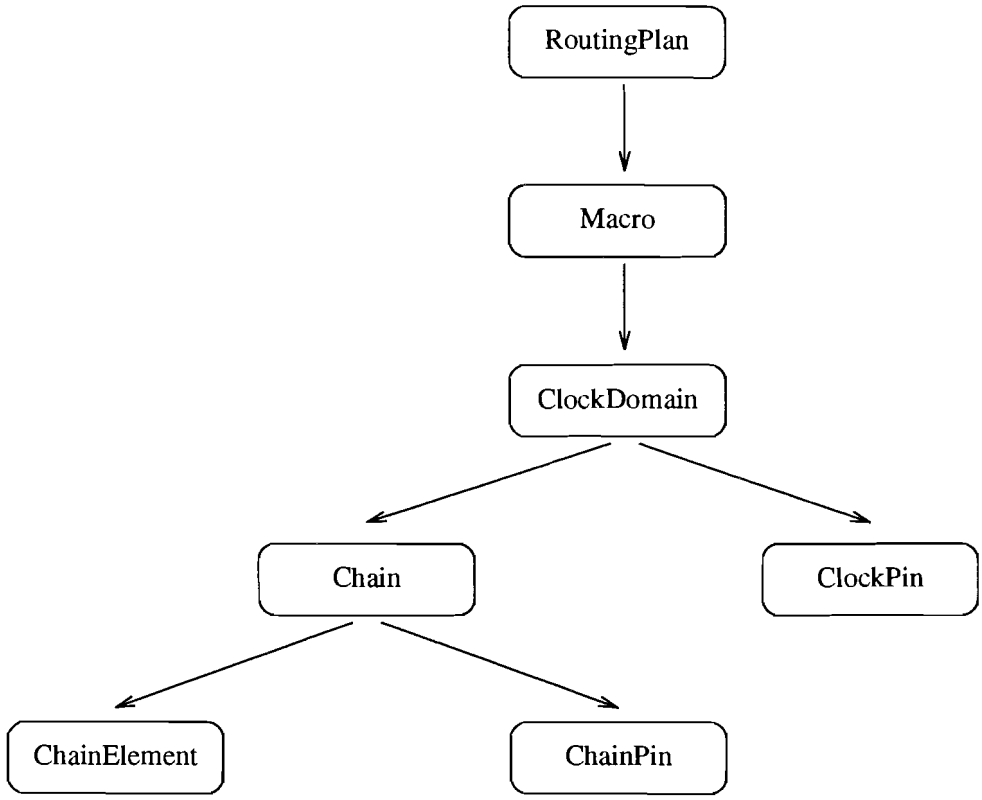

Figure 11.1: Ownership of SDS objects

11.1.2 Relation between SDS and NDS classes

When associating scan chains with a design, all SDS objects refer to a NDS counterpart. This relation in given in figure 11.2.

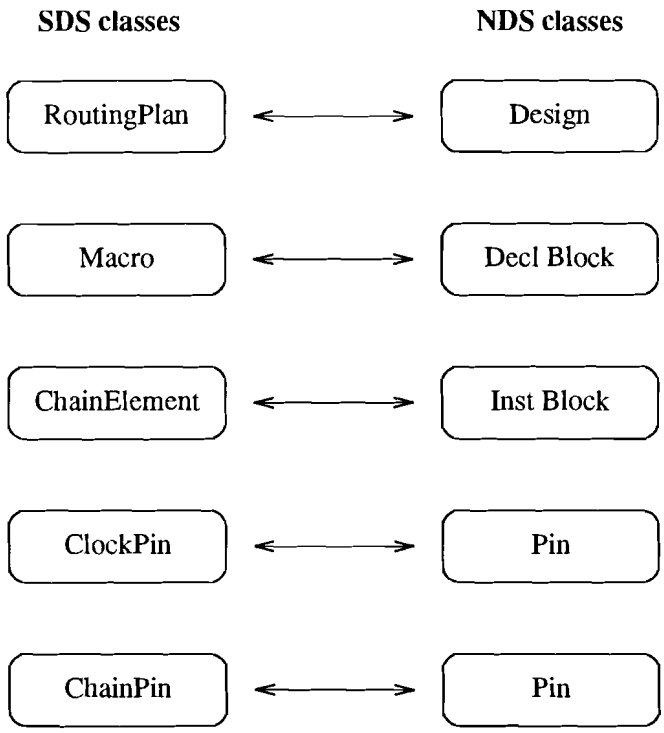

Figure 11.2: Relations between SDS and NDS classes

11.1.3 The RoutingPlan class

A RoutingPlan object is the only object through which the structure can be accessed. It gives access to all macros that are defined in the routing plan. A routingplan is a collection of macros, and functions exist to add, remove, and retrieve Macro objects.

11.1.4 The Macro class

A macro is an important entity in SDS. It is used to describe all chains that run through a block in your design. As said, each macro coincides with a declaration block in the design.

The chains of a macro are grouped together in their clock domains. All chains of one clockdomain are clocked by the same set of clock pins.

SDS was initially intended as a function library for InScan, a scan chain insertion program. InScan consists of two programs, PrepScan and ScanIt. The input of PrepScan is a design, with no or only partial scan chains. This design is evaluated, and a *routing plan* is generated, which describes which instances should be replaced by others, to transform the design into a full scannable design. ScanIt then performs this replacement/extension. InScan is discussed in [Voort 93].

Because of this, a macro 'M' is said to be `DerivedFrom` a declaration Block 'B'. In case of a library cell 'C', 'M' is said be `ReplacedFor` this cell. In this report, SDS and RPL

are used to describe scan chain which are already present in a given design. There are no "original" blocks or cells from which block are derived, or for which they have been replaced.

11.1.5 The ClockDomain class

In our formal model, each chain contains a set of clock ports, specifying the ports which (simultaneously) clock that scan chain. The restrictions drawn upon the construction of non-leaf scan chain guaranty that if two chains are used as chain elements in a higher chain, the two elements must have compatible clock ports.

In SDS, chains of a macro are grouped into *clock domains*. The clock pins of the chains are specified in the clockdomain, so chains of one clockdomain are compatible which each other (as far as clock pins are considered). A clockdomain states: All these clock pins are clocked simultaneously, i.e., it is assumed that they are connected properly at some higher level in the design hierarchy.

11.1.6 The ClockPin class

A ClockPin refers to a NDS Pin object, and contains a polarity field. The polarity denoted whether a rising (positive polarity), or a falling edge (negative polarity) on this pin is interpreted as being a clock pulse.

11.1.7 The Chain class

A chain object, which resides in macro 'M', refers to a scan chain which resides in the declaration block to which 'M' refers. If this block is a non-leaf block, the chain object may be describes by a list of chain elements, which described the internal structure. In case of a library cell, the chain will not contain chain elements.

Seen externally, a chain consists of several pins. These pins are represented by ChainPins:

ScanIn: This pin received the test data during test. Each chain has exactly one ScanIn Pin. Its polarity must be positive.

ScanOut: This pin emits test data during test. A chain may own several ScanOut pins. Its polarity denotes whether the chain inverts data or not.

ScanEnable: This pin must carry its specified value, to enable test mode. A chain may own several ScanEnable pins.

NormEnable: This pin must carry its specified value, to enable normal mode. A chain may own several NormEnable pins.

HoldEnable: This pin must carry its specified value, to enable hold mode. A chain may own several HoldEnable pins.

In our formal model, we've only used enable ports. These ports are represented by ScanEnable pins here. NormEnable and HoldEnable pins were not used in the formal model, since they are not necessary to recognise and remove scan chains.

Besides its pins, a scan chain has two more characteristics: a length, indicating how many clock cycles it takes for data at the input to appear at the output, and a boolean property, indicating whether the chain inverts this data or not. The latter property is stored in the polarity of the scan-out pin of a chain, the first property is stored separately.

11.2 The RoutingPlan Language

The routingplan language, RPL, is used to read and write scan chain hierarchy descriptions. A routing plan file corresponds to a RoutingPlan object. The file contains a description of the underlying structure, describing macros, clock domains, chains, etc.

We will use the RPL language as input and output file format in the NDS implementation. The complete syntax of the RPL language can be found in appendix B.

Chapter 12

VERA

VERA stands for "VERification Assistant". Is was developed to be able to write applications which could be used for the verification trajectory, which was discussed in chapter 2. The VERA manuals [Lynch 93], [Kostelijk 93], [Kuppen 93], state:

> VERA is a rule-based tool intended for use in the structural verification of IC designs. The rule-based approach results in a flexible tool which can be used for a variety of applications. It can be used to check the presence of facts, check the absence of a given set of errors, to modify a network description and to gain statistics on a design.
> ...
> The most common usage of VERA is *hierarchy-extraction*. The extraction of a circuits hierarchy can be seen as the means with which the structure of a circuit design can be verified. That circuit design will generally have been made in a top-down fashion, working successively from higher to lower level structures in the hierarchy. The verification process thus consists of ensuring that all expected modules are present and properly connected at each level in the circuit design hierarchy, by re-creating that hierarchy in a bottom-up fashion. This verification will generally start therefore on the (flat) netlist which has been extracted from the layout resulting from the overall design.

VERA is written in Common Lisp. Because of its flexibility, it was decided to investigate the performance of VERA in applications, where for now, only the C/C++ language was used.

12.1 Introduction

Vera operates on a network (our formal non-leaf cell). It uses the following inputs:

Element-type descriptions: These store the different characteristics of types including their possible expansion in terms of lower-level elements. They represent our formal cells.

Network description: This is a (flat) netlist, representing the circuit.

Knowledge-base: This contains definitions of rules: what they do with (or to) the current network.

Control file: This contains the succession of commands which are to be carried out during the session. In other words, this is the *program*.

Just like programs using NDS will be written in C++ (like NDS), programs using VERA will be written in Lisp (like VERA). The control file needed by VERA contains VERA-commands. However, the control file may contains normal Lisp commands, since VERA-commands are just extensions to the Lisp language.

Some of the possible control commands are:

(`load` *file*): This Lisp command will load and execute the given *file*. It is used for instance in case of batch-control.

(`load-kb` *file*): This commands load the knowledge-base *file* into VERA. The knowledge-base contains all rules and actions defined. Rules and actions will be discussed in section 12.2.

(`load-nd` *file*): This is the command required to load a network-description into VERA.

(`load-td` *file*): This command must be executed before the network-description is loaded, i.e., before a (`load-nd` *file*)-call. It causes the type-descriptions to be loaded.

(`load-wp` *file*): This command is intended to allow the user the option of loading in certain additional (or changing existing) properties of nodes or elements.

The above mentioned commands all load information into VERA. The most important command however is the

(`activate` *rule*)

command. Its causes the given rule, or match, named *rule*, to be "activated". Rules and matches will be discussed in the next section.

12.2 Rules, matches, and actions

Rules, matches, and actions are loaded into VERA with the (`load-kb` *file*)-command. A rule in general consists of a *match* and an *action* part. If the match-part of a rule can be satisfied, the actions of the action-part will be undertaken.

A rule is called a *match* if it consists of a match-part only. An *action* is a rule without a match-part. VERA has built-in actions and matches. Some examples of these, so-called *primitives*, are:

Recognize: A primitive match for finding a given structure in the circuit under verification.

Abstract: A primitive action for removing a structure found, and replacing it with a single higher-level element.

Test: A primitive match which can be used to check whether certain information found conforms to a given criterion.

Message: A primitive action which will print a user-defined message to a given file for every match accepted.

Two rules are used in the implementation of VERA FindScan: the primitive rule "test", and the "path" rule. Both will be discussed in the next sections. These, and all other rules are described in detail in the Vera reference manual [Kostelijk 93].

12.2.1 The "test" rule

The syntax of the test rule is given in figure 12.1. This match-call tests the given *function*, given its arguments. The match is accepted when the function-result is not `nil`, i.e., when *true* is returned. The test-rule allows the user to write Lisp functions, and use them as a match-rule.

```
(test (match-parameter-part)
      (function
       argument*
      )
)
```

Figure 12.1: Syntax of the VERA *test* rule

12.2.2 The "path" rule

The syntax of the path rule is given in figure 12.2. The given expressions have the following meaning:

start, *finish*: Indicate the beginning and end of the path.

nn-ne-en-ee: Indicates whether the *start* (*finish*) is a node or an element.

terminal-type-or-terminal-class: Indicates whether the terminal-lists are about terminal-classes or terminal-types.

node-to-element-terminal-list: This list determines which terminals of elements are allowed in a path, when a node to element boundary is crossed.

element-to-node-terminal-list: This list determines which terminals of elements are allowed in a path, when a element to node boundary is crossed.

excluded-element-list: This list determines which elements *are not allowed* in a path.

excluded-node-list: This list determines which nodes *are not allowed* in a path.

included-element-list: This list determines which elements *are mandatory* in a path.

included-node-list: This list determines which nodes *are mandatory* in a path.

maximum-path-length: The number of objects (being either elements or nodes) in a path will not exceed two times this values plus one.

maximum-number-of-paths: This is the maximum number of paths which will be returned by this primitive.

```
(path (match-parameter-part)
      (start
       finish
       nn-ne-en-ee
       terminal-type-or-terminal-class
       node-to-element-terminal-list
       element-to-node-terminal-list
       excluded-element-list
       excluded-node-list
       included-element-list
       included-node-list
       maximum-path-length
       maximum-number-of-paths
      )
)
```

Figure 12.2: Syntax of the VERA *path* rule

12.3 ND: the Network Description language

The network-description or ND-file contains the circuit which will be processed. The VERA netlist format is in Lisp-list form. Its syntax is specified in figure 12.3.

Certain requirements are necessary for each element specification:

1. The first entry, *type*, refers to an *already loaded* type-description.
2. The second entry, *element*, specifies the (unique) name of the elements itself, which can be a name or number.
3. The next entries, $term_1$, ..., $term_x$, are the terminal names of the elements. These should correspond to the terminal specifications in the type description *type*.

($element_1$
$element_2$
$\vdots$
$element_n$
)

where $n \geq 0$, and

$element_i = (type\ element\ term_1\ term_2\ \ldots term_x\ attrib_1\ attrib_2\ \ldots attrib_y)$, $x, y \geq 0$.

Figure 12.3: ND: the VERA netlist format

4. The last entries, $attrib_1$, ..., $attrib_x$, specify *attributes* of the element. The number and order of these attributes is again determined by the corresponding type description. A type description may have default values for an attribute. An element may use these default values (specify '!'), overrule these values (specify another value), or specify an attribute to be unknown/irrelevant (specify '?').

12.4 TD: the Type Description language

The type description or TD-file contains the information relevant to each different type of element which might occur in the network. Just like the network description, the TD-file is specified in Lisp-list form. Its syntax is specified in figure 12.4.

(td_1
td_2
$\vdots$
td_n
)

where $n \geq 0$, and

$td_i = (type\ (entry_1\ val_1)\ (entry_2\ val_2) \cdots (entry_m\ val_n)$, $m \geq 0$.

Figure 12.4: TD: the VERA type description format

The name of an element is specified in the *type* field, which is followed by a list of several properties of that element. The following two properties are mandatory:

`terminal-names` (*terminal-name**): a list of the terminal-names of the element.

`terminal-classes` (*terminal-class**): a list of the terminal-classes of the element. Ter-

minal which have identical classes, may be interchanged, without changing the functionality (e.g. the input terminals of a NAND)

Optionally the following properties may be specified:

`terminal-types` (*terminal-type**): A list of the terminal-types of the element. This property indicates how information flows, i.e., *in*, *out*, or *io*.

`terminal-groups` (*terminal-groups**): A list of the terminal-groups of the element. Whereas terminal-classes indicate interchangeability of individual terminals, terminal-groups indicate interchangeability of groups of terminals.

`attribute-names` (*attribute-name**): The attribute-names determine which attributes an element-type possesses, and the order they must have in a network description.

`attribute-tolerances` (*attribute-name**): The attribute-tolerances determine, for numeric attribute only, the tolerance allowed for matching this element with a "matching" value.

`default-attribute-values` (*attribute-value**): These attribute-values are used as default in case an element in a network specifies '!' for this attribute.

`global-nodes` (*node**): A list of global nodes, such as supply, ground and clock lines. This is useful when "abstracting" higher level elements.

`network` *network-description*: Describes this element-type in terms of lower-level element-types.

`restrictions` (*restriction*): Restrictions are used with the *recognize* rule.

Chapter 13

NDS: implementation of the algorithms

13.1 FindScan

As discussed earlier, FindScan needs a design and a set of leaf scan chains as input. It then produces a description of the chains found in the design.

13.1.1 File formats

Design: The design may be described in several formats. Among others, NDL and Edif can be read and written. A design can also refer to blocks, which are not specified within the design. These blocks should be found in the library.

Library: The library contains leaf blocks (cells), which may be used in a design. the library may be specified using the NDL, EDT or Edif format.

Leaf scan chains: The leaf scan chains are specified by their external properties: pins, length, and possible inversion. Since these leaf chains are consistent with the SDS definition of chains, they can be specified using the RPL file-format.

Scan chains: After processing of the input data, FindScan will generate a list of the scan chains found in the design. Of course this can (and will) be written out using the RPL file format.

13.1.2 Information flow

The described information flow of FindScan is depicted in figure 13.1.

13.1.3 Generating of the leaf scan chains file with GenRpl

In our algorithm, hence in the NDS implementation, we have assumed that a list of leaf scan chains is available. Since we use the RPL language to represent these scan chains,

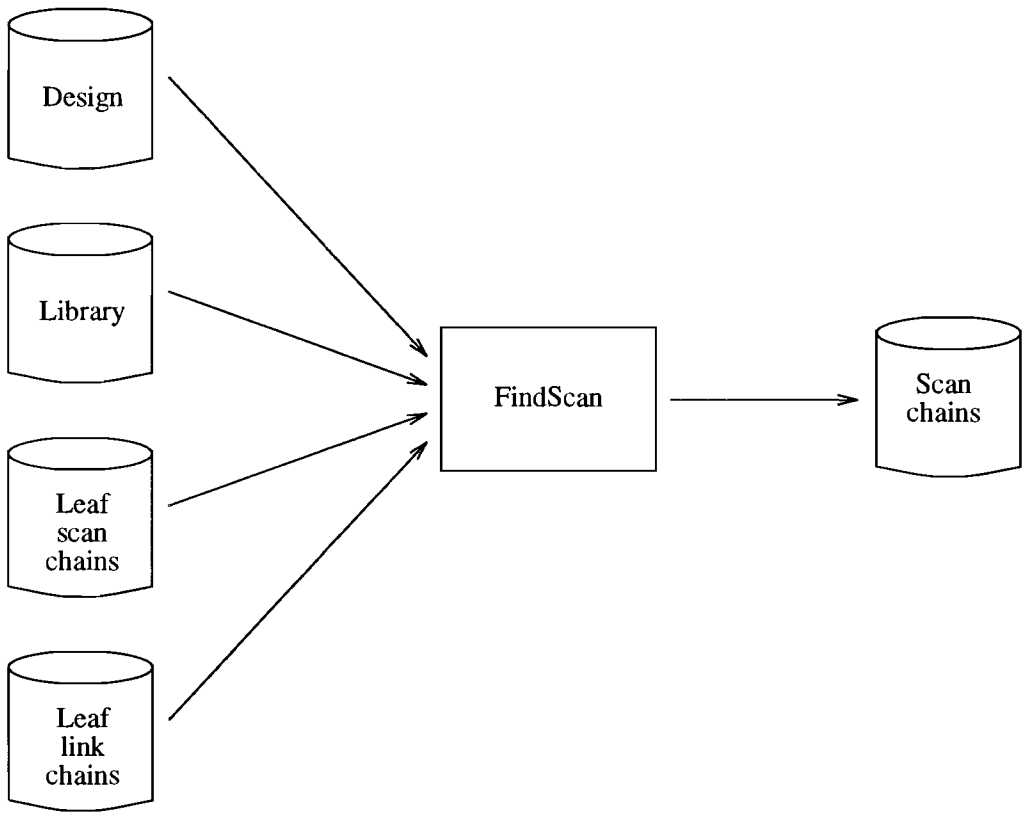

Figure 13.1: Information flow during a FindScan session

this file could be manually generated.

However, the LDS toolkit provides us with many function calls to retrieve information about library cells. We can use this information to automatically generate the RPL file. If `cell` is a `CellHandle`, the following calls (among other) are available:

`cell->IsBuf( )`: returns true iff this cell has one input port, and one outport that just buffers the input.

`cell->IsInv( )`: returns true iff this cell has one input port, and one outport that inverts the input.

`cell->IsPrimFF( )`: returns true iff this cell is a normal D-flipflop, with or without an extra inverting output.

`cell->IsSFF( )`: returns true iff this cell is a scannable flipflop.

A tool, GenRpl, has been developed, which uses the information from the library file to generate an RPL file containing leaf scan chains.

For a buffer `buf` with input `A` and output `X` the following RPL text is generated:

```
Macro buf
  ClockDomain no_clock NotDriven
    Chain chain1
      ScanIn  A
      ScanOut Q
      Length  0
    EndChain { chain1 }
  EndClockDomain { no_clock }
EndMacro { buf }
```

For an inverter `inv` with input `A` and output `Q` the text is almost identical. Only an inversion sign '-' is added:

```
Macro inv
  ClockDomain no_clock NotDriven
    Chain chain1
      ScanIn  A
      ScanOut -Q
      Length  0
    EndChain { chain1 }
  EndClockDomain { no_clock }
EndMacro { inv }
```

For a D-flipflop `dff` with input `D`, output `Q`, inverting output `QB`, and clocked by port `CK` we have:

```
Macro dff
  ClockDomain the_clock DrivenBy ( CK )
    Chain chain1
      ScanIn  D
      ScanOut ( Q -QB )
      Length  1
    EndChain { chain1 }
  EndClockDomain { the_clock }
EndMacro { dff }
```

For a scannable flipflop `sff` with scan data input `DT`, output `Q`, inverting output `QB`, enabled by `SE`, and clocked by port `CK` we have:

```
Macro sff
  ClockDomain the_clock DrivenBy ( CK )
    Chain chain1
      ScanIn  DT
      ScanOut ( Q -QB )
      Length  1
    EndChain { chain1 }
  EndClockDomain { the_clock }
EndMacro { sff }
```

Manual generation/extension of the RPL files

In LDS one can retrieve information about a cell. A cell is either combinatorial or sequential. A combinatorial cell may be further categorised, being a buffer, an inverter, a NAND, etc. A sequential cell may be a D-flipflop, a scannable flipflop, etc.

Because these properties are specified for the entire *cell*, instead of for each individual *port*, GenRpl will not find all possible leaf scan chains. The global cell approach fails for instances with the following configuration: A cell with inputs `A` and outputs `Q1` and `Q2`, for which:

$$Q1 = A,\ Q2 = \overline{A}$$

This cell thus contains one buffer, and one inverter. However, the entire cell cannot be seen a "a buffer" or "an inverter". Hence LDS can only characterise this cell as being combinatorial. In cases like this one, the user may manually extend the RPL file with a macro description for such a cell.

Extending an RPL file manually does not involve must work, since the RPL language is humanly readable. With the use of an arbitrary editor the user can add/delete macros and chains to an RPL file. This can be done for two reasons:

1. The library contains cells which were not recognised by GenRpl. They may have

properties unknown to GenRpl, or be of the form sketched above.

2. The scan chains of a non-leaf block of the design are known by the user. FindScan will interpret the existence of a macro for a non-leaf block as: "this block has already been processed, and contains this set of chains". This prevents FindScan of examining the structure of that block. This can also be used to specify scan chains which are not covered by our scan chain definition.

13.1.4 Tracer

Most of the functionality of FindScan is contained in the Tracer toolkit. It is a collation of C++ classes and associated functions, which are capable of *tracing* scan chains. The *LinkTracer* can be used to retrieve the driving pin of a clock or enable pin of a scan chain element. The *ScanTracer* is capable of recognising entire scan chains.

These two classes are capable of retrieving scan chain of a design. Together with design and library parsers, a routing plan reader and writer, they form the FindScan application.

13.2 RemoveScan

As discussed earlier, RemoveScan works on the same design and library as FindScan. It also receives the detected scan chains of FindScan, and a math-rule file. After processing this data it produces a scan-free version of the design.

13.2.1 File formats

Design: The design can again be described using the NDL or Edif.

Library: The library can again be specified using the NDL, EDT or Edif format.

Scan chains: These will be specified using RPL.

Match-rule file: The match-rule file contains replacements for blocks, i.e., it has a design structure. Therefore it can be represented in NDL or Edif.

13.2.2 Information flow

The described information flow of FindScan is depicted in figure 13.2.

13.2.3 Generating the match-rule file

In our algorithm, hence in the NDS implementation, we have assumed that a match-rule file is available, which describes how cells, which are part of a scan chain, should be replaced.

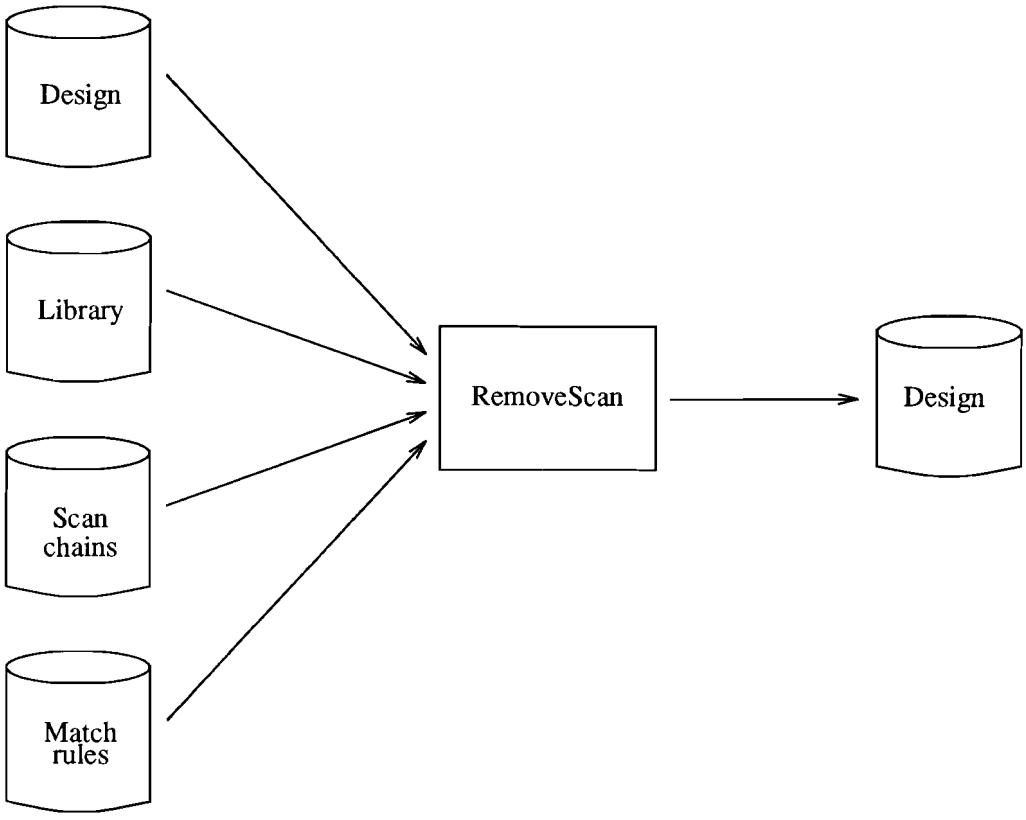

Figure 13.2: Information flow during a RemoveScan session

Scan chains which do not have enable ports, share the normal and scan functionality. Examples of such cells are buffers, inverters and the D-flipflop. So if these cells are parts of a scan chain, they do not have to be replaced.

A multiplexer and a scannable flipflop do have enable ports. Removing scan functionality means assuming the cells will never be enabled. In case of leaf scan chains with *one* enable port, we can assume that this port always carries the not-enabled value.

Using NDL, the math-rule file of a two-input multiplexer, named `MUX2` looks like:

```
      MACRO
MACRO MUX2_1 I(A,B,S) O(Q)
Inst1 BUF    I(B)     O(Q)
      MEND

      MACRO
MACRO MUX2_2 I(A,B,S) O(Q)
Inst1 BUF    I(A)     O(Q)
      MEND

      MACRO
MACRO MUX2_3 I(A,B,S) O(Q)
      MEND
```

Buffer `BUF` is used to model to connection between an input and an output, since NDL is not capable of describing nets containing more than one external port.

There are three different replacements for `MUX2`. `MUX2_1` must be used when the chain from `A` to `Q` is used in a scan chain, and the chain from `B` to `Q` isn't. `MUX2_2` must be used is the opposite case, i.e., only the chain from `B` to `Q` is used in a scan chain. Finally `MUX2_3` must be used is both chains are used in a scan chain.

The match-rule file of a scannable flipflop, named `SFF`, looks like:

```
      MACRO
MACRO SFF_1  I(CK,D,DT,SE) O(Q)
Inst1 DFF    I(CK,D)       O(Q)
      MEND
```

A scannable flipflop is thus simply replaced by a D-flipflop.

Chapter 14

VERA: implementation of the algorithms

14.1 FindScan

As discussed earlier, FindScan needs a design and a set of leaf scan chains as input. It then produces a description of the chains found in the design.

14.1.1 File formats

Design: The design may be described in several formats. Edif and of course the VERA TD/ND format can be read. Other file formats must first be converted.

Leaf scan chains: The leaf scan chains are specified by their external properties: pins, length, and possible inversion. Since the leaf chains are consistent with the SDS definition of chains, they can be specified using the RPL file-format. A tool has been developed to transform RPL-input into a Lisp-like language, suitable for the VERA environment.

Scan chains: After processing the input data, FindScan will generate a list of the scan chains found in the design. Just like the C++ version it will be written out using the RPL file format.

14.1.2 Information flow

The described information flow of FindScan is depicted in figure 14.1.

14.1.3 Generation a leaf scan chains file

Since GenRpl (discussed in the NDS implementation chapter) automatically generates a leaf scan chains file in the RPL format, we could use this file to import the leaf chains. However, this would require a RPL-parser to be written in Lisp.

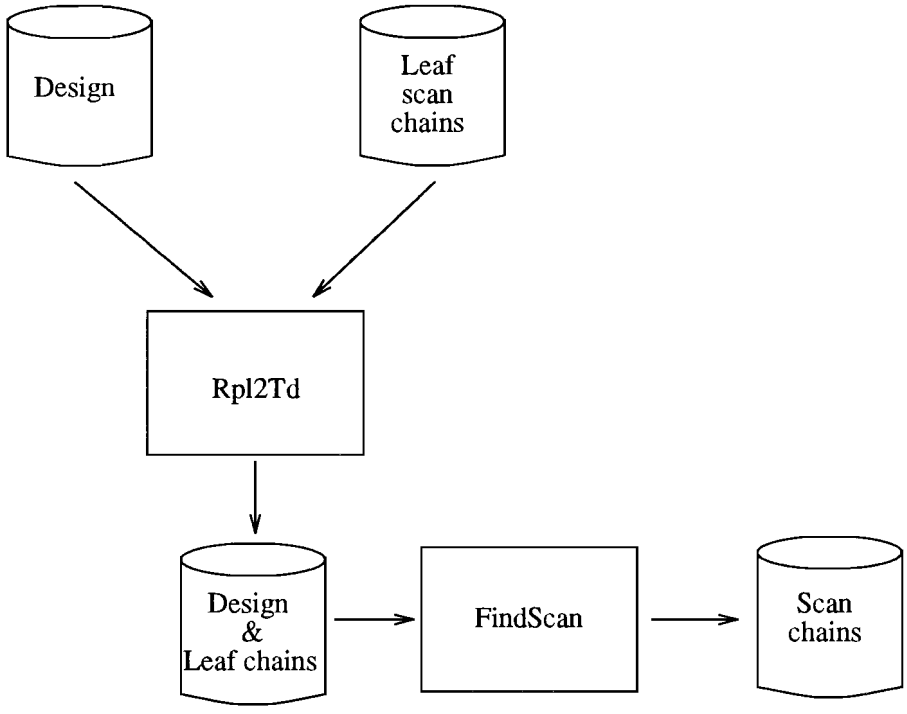

Figure 14.1: Information flow during a FindScan session

Writing this parser would result in two parsers, one in C++ and one in Lisp. In that case two parsers had to be supported. Therefore another approach has been taken. We've developed a tool which parses a given RPL file, and produces a semantical identical file, using another syntax. This syntax is the Lisp list format, which can easily be read into VERA. The tool is called *Rpl2TD*. It reads a design, library and leaf scan chains file (in RPL format), and writes and a TD-file, containing the design, and information about the leaf scan chains.

14.1.4 Rpl2TD

As discussed in chapter 12, the TD-format consists of a Lisp list, containing *type descriptions*. The Rpl2TD tool reads a design, a library, and a RPL file, stores them into NDS, LDS, and SDS structures (using the parsers of NDS and SDS), and converts this set of data structures into a TD-file.

The conversion is done by two programs:

Rpl2TD: This program is written using the NDS, LDS and SDS toolkits. It converts the input data into an Edif design/library file, and additionally generates a VERA batch file. This batch file has to be executed, using VERA.

VERA batch file: This program (generated by Rpl2TD) reads the Edif design/library file, and adds the leaf scan information to each type description. Finally the extended

TD-file is written out.

The generated VERA batch file is of the form shown in figure 14.2.

```
(load "rpl2td_init.lsp")
(load-edif "design_name.edif_out")

(add-attribute-to-td '(type-1) 'scan-chains '( ... )
(set-terminal-type 'type-1 'term1 'sin)
(set-terminal-type 'type-1 'term2 'sout)

 ...

(add-attribute-to-td '(type-n) 'scan-chains '( ... )
(set-terminal-type 'type-n 'term1 'sin)
(set-terminal-type 'type-n 'term2 'sout)

(save-td "design_name.td")
```

Figure 14.2: The VERA batch file generated by Rpl2TD

The "`rpl2td_init.lsp`" file contains the implementation of the "`set-terminal-type`" function. Its function is to change the terminal-type of the given terminal to the given type. Scan input and output terminals are given the type `sin` and `sout` respectively, clock ports are given the type `cin`.

The final action of the batch file is to write the type-description file. It contains all necessary information for FindScan: the design and library, and the leaf scan chains.

The only information not explicitly present in the TD-file is a list of leaf link chains. These are however *implicitly* present. Link chains can been seen as a subset of scan chains: the chains without clock or enable terminals.

14.1.5 Match-rules used to recognise scan chains

The definition of a scan chain is given in a form, very much like the VERA approach. First a set of possible scan chain is defined. Then several restrictions are given, which define subsets of the larger set. The combination of all restrictions results in the set of scan chains.

The (large) set of possible scan chains, called *scan chain candidates* is retrieved by using the `find-scanchain-candidate` rule defined below:

```
(define-rule FIND-SCANCHAIN-CANDIDATES (p)(start end)
    ()
    (
     (path (p) (start end 'node-node 'terminal-type '( sin ) '( sout )
                '() '() '() '() 'max 'max ))
```

```
    )
    ()
)
```

This rule returns all scan chain candidates, using the path-rule, discussed in chapter 12. The candidates found have to be checked, whether they form a valid chain or not. First the length of the path is checked. This length *is not* the length of the chain as defined in our scan chain definition, but the VERA interpretation of the length of a path.

The length of a path

The length of a path is the sum of the number of nodes and the number of elements of that path. Since two terminal of a type cannot be connected directly in VERA, a chain must contains at least two nodes, and one element. Because the number of elements is one less than the number of nodes, the length of the path must be $3, 5, 7, \ldots$

The following match filters out the path which have an incorrect length:

```
(define-match FILTER-OUT-INCORRECT-LENGTHS (p)(start end)
    ()
    (
     (test (p) ('length-of-scanchain-is-valid (instance-of p)))
    )
    ()
)
```

where

```
length-of-scanchain-is-valid (p)
```

is a Lisp-function returning true iff the length of the given path is $3, 5, 7, \ldots$

The elements of a path

The scan chain candidates found using the path-rule are concatenations of elements, entering via a terminal with type `sin`, and leaving via a terminal with type `sout`. This will lead to invalid elements in case of elements which contain two or more scan chains. The type shown in figure 14.3 contains two separate chains. For clarity, the clock and enable terminal are left out.

Suppose an element of the type shown in figure 14.3 is used in another type. The terminal-type of terminal A,B is `sin`, that of Q1,Q2 is `sout`.

The path rule only uses these terminal-types. It has no knowledge about which scan input and scan output terminal pairs are part of the same scan chain. Therefore it may come up with the non scan chain elements "A to Q2" or "B to Q1". The following rule will recognise these invalid elements, and remove the corresponding "scan chains".

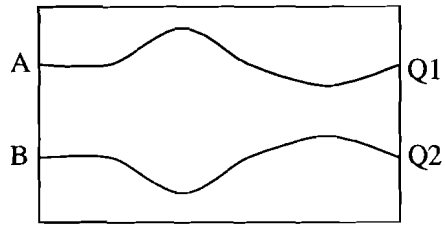

Figure 14.3: A type containing two separate chains

```
(define-match FILTER-OUT-INCORRECT-ELEMENTS (p)()
    ()
    (
     (test (p) ('elements-of-scanchain-are-valid (instance-of p)))
    )
    ()
)
```

where

```
elements-of-scanchain-are-valid (p)
```

is a Lisp-function returning true iff all elements of the given path correspond to true scan chain elements.

Each type contains a list of its scan chains, therefore the Lisp-function only has to check whether the element-type contains a scan chain starting and ending at the terminals found by the path-rule.

The clocks of a chain

Since the above rules/matches have filtered out incorrect concatenation of elements, the chains that are left only contain elements which refer to scan chains, i.e., they contain *scan chain elements.*

The scan chains with incorrect clock nodes are discarded by the following rule:

```
(define-match FILTER-OUT-INCORRECT-CLOCKS (p)()
    ()
    (
     (test (p) ('clocks-of-scanchain-are-valid (instance-of p)))
    )
    ()
)
```

where

```
clocks-of-scanchain-are-valid (p)
```

is a Lisp-function returning true iff the elements of the given path are clocked by terminals via link chains, and are compatible among each other.

A function has been written, just like in the C++ linktracer toolkit. Given a node, it returns the node which drives the given node through a link chain:

```
driving-node-of-node (node)
```

This function is based on the *find-linkchains* rule, which is similar to the *find-scanchains* rules. The difference is that the find-linkchains rule prohibits chains with clock or enables terminals.

The enables of a chain

The scan chain with incorrect enable nodes are discarded by the following rule:

```
(define-match FILTER-OUT-INCORRECT-ENABLES (p)()
    ()
    (
     (test (p) ('enables-of-scanchain-are-valid (instance-of p)))
    )
    ()
)
```

This rule is identical to the clock checking rule. There are separate rules in case clocks and enable should be treated differently in future versions of FindScan.

14.1.6 Determining the scan chains of a design

As explained in chapter 8, the order in which the different types can be processed is restricted. This restriction can be derived from the descendence graph of the design. In VERA a separate toolkit, the *TD-monitor* is present, which can be used to retrieve such hierarchy information.

TD-monitor

One of the command the type-description monitor recognises is the *show-hierarchy* command. It is used by FindScan:

```
(td-monitor 'show-hierarchy '! T)
```

This call returns a list of types, and their *level* in the hierarchical structure. We use this information to choose a valid processing order for the types present in a design.

The program

FindScan itself is implemented as a Lisp function. A type-description file must have been loaded, containing information about the leaf scan chains. FindScan first uses the TD-monitor to determine the order in which to process the types.

Before a type is processed, its network is loaded into VERA. Then the match-rules described earlier are used to find the scan chains present in the type. After this, the chains that were found are added to the type-description of the type under examination.

Chapter 15

Comparing NDS and VERA

In this chapter we will discuss some difference between NDS and VERA, using FindScan as a test application.

15.1 Design manipulation

There is a big difference in the way designs can be manipulated in the NDS or VERA environment.

NDS, being a Network Data Structure, has many function calls which allow the user to inspect and/or change a loaded design. For instance, given a block, one can retrieve its children, its parent, its nets, etc. The NDS toolkit consist of almost all such basic manipulations. Using these functions, the user can build an application performing a specific task.

Writing FindScan using NDS required a tool which could recognise chains of scan chain elements, the Tracer. The Tracer needs a list of possible scan chain elements. The SDS toolkit was used as a suitable data structure. The Tracer supplies functions to retrieve driving pins of clock and enable ports, as well as functions to search for complete scan chains.

VERA was written for verification purposes. It contains rules, matches, and actions, which can be activated. These rules inspect and/or change a loaded network. The user may create new rules, using existing rules.

FindScan is not a verification application, but it does need structure recognition. We've used the path-rule to recognise a group of possible scan chains. Because the path-rule has no knowledge of which scan input belongs to what scan output, it may come up with many invalid chain elements and therefore with many invalid chains.

To speed up the execution time, as well as reducing the memory requirements, we could modify the path-rule. This modified path-rule should be able to find only strings of scan chain elements, i.e., be able to match input and output terminals of a type. Modifying the path-rule would however require more information about the internal Lisp structures used in VERA.

15.2 Implementation effort

Comparing the implementation effort needed in both world is rather difficult. We first developed the C++ implementation. This was done by creating a data structure toolkit for scan chains, SDS, and developing a toolkit capable of tracing chains of elements, Tracer.

FindScan merely consists of a small program, using the facilities offered by NDS, SDS, and the Tracer toolkits. The implementation effort of FindScan using NDS therefore includes the effort of implementing SDS and Tracer. It must be noted that a parser for the routing plan language was already available.

After the NDS version of FindScan and RemoveScan was finished, the implementation using VERA was started. To avoid an extra parser for the RPL language, written for VERA, a tool was developed which converted RPL. Macro information was added as an attribute to the types (cells) to which the macros referred.

The tracer equivalent software was written using the path-rule built into VERA. Several extra rules were written in Lisp, using the test-rule. Finally an RPL writer function was developed, to be able to compare the results of the NDS and VERA versions.

15.3 Run-time and memory requirements

The two FindScan implementations are both executed on a number of designs. Both returned a list of the scan chains found, using RPL. The MatchRpl tool has been used to compare the results of the two programs. As expected, they returned the same set of chains in every tested case.

The run-time needed by the two implementations was measurement on a HP9000/735 platform. The result are listed in figure 15.1.

Name	depth	largest cell	NDS run-time	VERA run-time
d2r	1	11291	5436	2278
estimator	4	1170	123	5953
pdemxmj	2	127	2.12	1.76
melm	2	1374	4.98	–
timer	6	616	1.67	3.68
tdma	6	185	1.91	783.63
tutor	5	8	0.23	1.02

Figure 15.1: Run-time results in seconds of FindScan on a HP9000/735 platform

The VERA implementation of FindScan almost always requires more memory, due to the fact that it starts with a large set of scan chain candidates. The NDS version only stores valid scan chains, as soon as a scan chain is found invalid the search is stopped, and the traversed path is discarded.

15.4 Reliability

By reliability we mean the confidence one has in an implementation, given the fact that it worked on several test cases. In other words, which parts of the software are used when FindScan is executed on a particular design.

The advantage of the VERA implementation is the fact that the user does not have to worry about exceptions such as NULL-pointers, illegal input etc. The user can rely on that fact that these exceptions are handled by VERA. This implies that executing the software on an average design will test almost all functions of it.

15.5 Flexibility

Flexibility stands for the ease with which the program can be adapted to a (slightly) different definition of a scan chain. both version are flexible in the sense that they allow different leaf scan chains to be added.

A modification of the algorithm would be necessary for instance if the definition of enable ports would be extended. In the present definition only elements with one input and one output terminal are allowed in the chain of elements which drive an enable port. If we allow multi-input elements like ANDs and ORs, the enable ports is not driven by one port, but by as set of ports.

The Tracer toolkit could be extended to search for such paths. Extending the VERA implementation would, because of memory and run-time requirements, almost certainly require the modification of the path-rule mentioned earlier. Using the regular path-rule would result in an enormous amount of possible paths, which then have to be restricted using additional rules.

Chapter 16

Possible improvements

Based upon the given definition of scan chains two algorithms have been specified and implemented using NDS and VERA. Both implementations used structure recognition to identify scan chains in a design.

16.1 Extension of the scan chain definition

The definition of scan chains contained a restriction on clock and enable ports. The clock and enable ports of scan chain elements had to be *linked* to ports of the higher level cell. A port is linked by another port, if the latter one drives the first one, possibly through a chain of buffer and/or inverters.

Sometimes the clock is not only driven through buffers or inverters. It's possible that the clock is generated by a special clock generator cell. An extension of the definition should make it possible to specify an output of such a cell as being a valid clock generator.

The enable ports of a chain may also be driven by other logic than buffer or inverter. For instance, it's possible that a (former unused) combination of input ports of a cell is used to enable a scan chain. This combination has to be detected by a logical subcircuit.

The same solution as for clock ports could be used, i.e., specifying an output of a specific cell as being a valid enable generator. Another solution would be to allow other (multi-input) combinatorial cell is the enable path.

16.2 Structure versus expressions

Another approach of scan chain recognition and removing is to use expressions instead of structure. A toolkit, XDS, exists which can transform structure into an equivalent expression, and vice versa. The implementation of RemoveScan using expressions will briefly be discussed in the following section.

16.2.1 Implementing RemoveScan using expressions

As already mentioned, removing a scan chain means *not enabling* it. Using the EDS toolkit, which is capable of manipulating expressions, an algorithm could be developed, which would recognise unused logic due to scan chain removal. This logic could be discarded, which is equivalent to removing the scan functionality.

16.2.2 Automatic generation of the Match-rule file

We noticed that the number of possible replacement of a block was 2^n, with n being the number of possible scan chains through that block. Theoretically it's possibly to have 2^n different instances of the block, which all must be replaced by another block.

In practice most instances need the same replacement. Therefore the match-rule file does not have to contains all 2^n replacements. The match-rule file could be generated "on request", using the results of findScan to determine which replacement blocks are needed.

Another problem is finding such a replacement block. For combinatorial cells the expression approach will be suitable. After simplifying the expressions, another block (library cell) must be found which matches the simplified version generated.

Chapter 17

Conclusions

In this report algorithms for finding and removing of scan chains have been discussed. The implementation of these algorithms is used to compare two different programming environments, NDS and VERA.

The NDS environment has been extended by a scan data structure, SDS, and a toolkit to trace scan chains and clock/enable lines, the Tracer. With use of these tools, an implementation of FindScan has been realised.

After the NDS version of FindScan was finished, the VERA implementation was started. Similar data structures to the SDS structures were implemented, including appropriate functions to operate on these structures. The Tracer toolkit was realised using primitive rules of VERA, with use of additional rules.

As expected, both versions of FindScan return the same set of scan chains, when given the same design. The VERA implementation however needs more run-time, as well as more memory. This results from the fact that VERA first recognises a set of possible scan chains, and later reduces this set to the set of real scan chains.

To speed up the VERA implementation, as well as reducing its memory requirements, the path-rule could be extended. A flexible solution would be for the user to be able to pass a Lisp-function, which will be used by the path-rule to determine valid paths. Writing such a function however requires inside information on the path-rule, so this would only be of help to experienced users.

Without this improved path-rule, we choose NDS as the better environment, because modification of the algorithm can be done, without needing change to the current environment. If other users of VERA also would require a more flexible rule for path-searching, resulting in an implementation of such a rule, VERA could be used as well.

The routing plan language, used to represent scan chains, could also use a small extension. It has been noticed that many scan chains do have multiple outputs, i.e., other flipflops besides the last one are reachable from the outside. In the current version of RPL, such a chain has to be represented by a set of chains, all starting at the same scan input pin. RPL could be extended to contain a list of additional output pins, all labeled with their specific position in the chain (specified by the number of clock cycles, and the inversion property).

Another option is to execute a task using both worlds. One could decide which parts of a task would be best suited to solve with one of the environments, and split up the job. To simplify this process, NDS has been extended to be able to output VERA-readable netlists. VERA has been extended by functions which could inspect properties of such a netlist. To export information from VERA to NDS, Edif could be used.

Bibliography

[Woudsma 90] WOUDSMA, R., F.P.M. BEENKER, J. VAN MEERBERGEN, C. NIESSEN [1990], *PIRAMID: an Architecture-Driven Silicon Compiler for Complex DSP Applications*, Proceedings ISCAS Conference, New Orleans, May.

[Beenker 90] BEENKER, F.P.M., M. VAN DER STAR, R.W.C. DEKKER, R.J.J. STANS [1990], *Implementing Macro Test in Silicon Compiler Design*, IEEE Design & Test of Computers, April.

[Claasen 89] CLAASEN, T.A.C.M., F.P.M. BEENKER, J. JAMIESON, R.G. BENNETS [1989], *New Directions in Electronic Test Philosophy, Strategy and Tools*, Proceedings of the 1st European Test Conference , Paris

[Baker 90] BAKER, K., S. VERHELST [1990], *IDDQ testing because "zero defects isn't enough": a Philips perspective*, Proceedings International Conference, Washington, p. 253-254

[Bennets 85] BENNETS, R.G. [1984], *Design of testable logic circuits*, Addison Wesley

[Fujiwara 85] FUJIWARA, H. [1985], *Logic testing and design for testability*, The MIT Press

[Bennets 93] BENNETS, R.G., F.P.M. BEENKER [1993], *Partial Scan: what Problem does it Solve ?*, Proceedings European Test Conference, Rotterdam, The Netherlands, April 19-22, p. 99-106

[EIA 87] EIA [1987], *Electronic Design Interchange Format version 2 0 0*, Electronic Industries Association, Engineering Department, 2001 Eye Street, N.W. Washington, D.C. 20006

[Goel 81] GOEL, P., B.C. ROSALES [1981], *PODEM-X: An Automatic Test Generation System for VLSI Logic Structures*, Proceedings 18th Design Automation Conference, p. 260-268, June

[Fujiwara 83] FUJIWARA, H., T. SHIMONO [1983], *On the Acceleration of Test Generation Algorithms*, IEEE Transactions on Computers, Vol. C-32, No. 12, p. 1137-1144, December

[Philips ED&T 93] Philips Electronics N.V [1993], *NDS v2.1 User Manual*

[Philips ED&T 93] Philips Electronics N.V [1993], *LDS v2.0 User Manual*

[Lynch 93] Lynch B.F., A.P. Kostelijk, P.A. Kuppen [1993], *VERA user manual*, Version 3.0

[Kostelijk 93] Kostelijk A.P., B.F. Lynch, P.A. Kuppen [1993], *VERA reference manual*, Version 3.0

[Kuppen 93] Kuppen P.A., A.P. Kostelijk, B.F. Lynch [1993], *VERA rule manual*, Version 3.0

[Voort 93] Voort, T.A.F.M. van de [1993], *InScan, specification and implementation of a scan chain inserter*, Philips ED&T internal report

Appendix A

Mathematical notation

This chapter describes the mathematical notations used in this report. Many readers will already be familiar with most of these notations. Some new notations are defined in this section.

A.1 Abbreviations

iff: *iff* stands for "if, and only if". It is used to state that two statements are equivalent. It is only used in text, in formulas we will use the equivalence-symbol '$\equiv$'.

A.2 Sets and tuples

Set: A *set* S is a collection of zero or more entities. Either an entity is part of the set, or it isn't. For example:

$$S = \{1, 2, 5, 7, 8, 9\}$$

is a set of natural numbers.

Power set: The *powerset* of a set S (denoted by $\mathcal{P}(S)$) is the set which contains all subsets of S, including $\emptyset$ and S itself:

$$\mathcal{P}(S) = \{T | T \subseteq S\}$$

Tuple: A *tuple* is a finite, nonempty, ordered set. It is denoted by

$$(el_1, el_2, el_3, \ldots, el_n)$$

A.3 Predefined sets

Set of natural numbers: $\mathbb{N}$ denotes the *set of natural numbers*:

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$$

Set of positive natural numbers: $\mathbb{N}^+$ denotes the *set of positive natural numbers*:

$$\mathbb{N}^+ = \{1, 2, 3, \ldots\}$$

Set of booleans: $\mathbb{B}$ denotes the *set of booleans*:

$$\mathbb{B} = \{false, true\}$$

A.4 Operators

For all: $\forall$ denotes the *for all* operator. It is used to denote the fact that a certain expression holds for all elements of a certain set. For example:

$$\forall n \in \mathbb{N}^+ : \ n > 0$$

Exists: $\exists$ denotes the *exists* operator. It is used to denote the fact that a certain expression holds for at least one element of a certain set. For example:

$$\exists n \in \mathbb{N}^+ : \ n < 3$$

Exists n**:** $\exists^n \ (n \in \mathbb{N})$ denotes the *exists precisely* n operator. It is used to denote the fact that a certain expression holds for precisely n elements of a certain set. It will mainly be used to state that an expression holds for precisely one element. For example:

$$\exists^1 n \in \mathbb{N}^+ : \ 4 < n < 6$$

The dot operator, used on a tuple: To refer to an *element of a tuple*, the *dot notation* is used. For example if T is a tuple of the type

$$(a, b, c, d, e)$$

the fourth element of it will be denoted by:

$$T.d$$

This means that, if we want to use the dot notation for a tuple, we have to give a (unique) name to all elements of that tuple. In this report we will define these names along with the definition of the tuple itself.

The dot operator, used on a set of tuples: We will also define the dot operator when used on a set of tuples.

Let T be a tuple type, d an element name of T, and $ST \subseteq \mathcal{P}(T)$ be a set of such tuples. The dot operator used on such a set is defined as

$$ST.d = \bigcup_{t \in ST} t.d$$

For example if T is a tuple of the type

$$(a, b)$$

and

$$ST = \{(1,2),(3,4),(5,7),(7,2)\}$$

then

$$ST.b = \{2,4,7\}$$

Appendix B

The Routing Plan Language

A scan chain routing plan is a description of a (possible) routing of scan chains in a design. The initial description of the plan is generated by PrepScan, the designer may modify this plan or even replace it by another plan. The plan can be complete but incomplete definitions are also possible.

Note that the routing plan language is restricted to tree-constructs. This means that the description of the scan chain connections is given from the top level design down to library cell level. Per hierarchical level the routing on that particular level is indicated. Hence, it is not possible to directly specify a connection between two scan cells at different levels of hierarchy. If such a connection is required the scanin and scanout terminals have to be propagated through the hierarchy levels such that proper scanin and scanout terminals exist at the various levels of hierarchy.

B.1 Syntax

```
routing_plan    ::=     [+ macro +]

macro           ::=     'MACRO' {macro}name
                        ( [ 'DERIVEDFROM' {derivedmacro}name ] |
                        [ 'REPLACEDFOR' {replacedmacro}name ] )
                        [+ clock_domain +]
                        ( 'ENDMACRO' | 'END' ) [ ';' ]

clock_domain    ::=     'CLOCKDOMAIN' [ {domain}name ]
                        ( [ 'DRIVENBY' {clocks}clock_list ] |
                        [ 'NOTDRIVEN' ] )
                        [+ chain +]
                        ( 'ENDCLOCKDOMAIN' | 'END' ) [ ';' ]

chain           ::=     'CHAIN' [ {chain}name ]
                        [ length ]
                        [ 'SCANIN' {scanin}name ]
```

```
                        [ 'SCANOUT' {scanouts}port_list ]
                        [ 'SCANENABLE' {scanenables}port_list ]
                        [ 'NORMENABLE' {normenables}port_list ]
                        [ 'HOLDENABLE' {holdenables}port_list ]
                        [ 'ORDER'  order_list
                        ( 'ENDORDER' | 'END' ) [ ';' ] ]
                        ( 'ENDCHAIN' | 'END' ) [ ';' ]

length          ::=     'LENGTH' {length}int

clock_list      ::=     '(' [ - ] {clock_port}name [* [ , ] [ - ]
                            {clock_port}name *] ')'

port_list       ::=     [ - ] {chain_port}name |
                        '(' [ - ] {chain_port}name [* [ , ] [ - ]
                            {chain_port}name *] ')'

order_list      ::=     element [* element *]

element         ::=     {instance}name 'OFTYPE' {decl}name
                        [ [ 'CLOCKDOMAIN' {domain}name ]
                        'CHAIN' {chain}name ] [ length ]

name            ::=     '"' [* anychar *] '"' |

                        char [* char | int *]

char            ::=     'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' |
                        'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' |
                        'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
                        'y' | 'z' | '$' | '_' | '-' | '/' | '\'

anychar         ::=     any non-quoted character

int             ::=     [0-9]*
```

- Comma's, spaces, tabs and newlines are separators.
- '*', '!' and '\#' till the end of a line is comment.
- [X] means that X can occur here at most once.
- [* X *] means that X can occur here any number of times.
- [X +]+ means that X should occur here at least once.
- 'TOKEN' means that TOKEN (a string) should appear here.
- Although all tokens are printed in upper case, it is allowed to use any case for tokens.

- tag is a tag that is used in the next section for referencing to the routing plan syntax. No attention should be paid to these right now.

B.2 Semantics

The semantic tags {...} have the following meaning:

- {macro} has no semantic meaning, unless it equals {derivedmacro}/{replacedmacro} or there is no {derivedmacro}/{replacedmacro} (which means the same), in which case it refers to the declaration name of the cell in the netlist.
- {derivedmacro} and \verbreplacedmacro+ refer to the original ("template") macro for hierarchy expansion.
- {domain} refers to the clock domain. It has only a semantic meaning when used in the rule for 'instance', in which case it must be the name of a clock domain of macro {decl}.
- {scanin} refers to the name of a scan-in port of the cell {macro} in the netlist.
- {clocks} refers to a list of clock port names of the cell {macro} in the netlist.
- {scanouts}, {scanenables}, {normenables} and {holdenables} refer to lists of port names of the respective port types of the cell {macro} in the netlist.
- {clock_port} refers to the name of a clock port.
- {chain_port} refers to the name of a scan-in, scan-out, scan-enable, norm-enable or holdenable port.
- {length} refers to the length of the scan chain.
- {chain} has only a semantic meaning when used in the rule for 'instance', in which case it must be the name of a chain of clock domain {domain}.
- {instance} refers to the instance name in the {macro}.
- {decl} refers to the declaration name of the mentioned {instance}. Hierarchy is implicitly described by using some {macro} as {instance}.
- LENGTH, SCANIN, SCANOUT, SCANENABLE, NORMENABLE and HOLDENABLE may be defined in any order.

B.3 Semantic rules

Below, rules are given to which the routing plan must conform in order to represent a valid routing plan, i.e., a routing plan that corresponds with the design. These rules are an addition to the syntax, in which it is not possible to render such rules. All rules will be checked by ScanIt after reading the routing plan.

B.3.1 Rules regarding uniqueness

- All macros must be unique within the routing plan.
- All clock domains must be unique within each macro.
- All chains must be unique within each clock domain.
- All clock ports must be unique within a clock port list.

B.3.2 Rules regarding order of definition

In general, it can be stated that each entity to which is referred should be defined before this reference, so the rule 'define before use' applies. In particular, the following rules should be adhered to:

- Referred instances must be (pre) declared in the routing plan.
- Referred clock domains must be (pre) declared on the macro in the routing plan.
- Referred chains must be (pre) declared on the macro in the routing plan.

B.3.3 Rules regarding length of chains

- In principle, the `{length}` of a scan chain as is specified in the routing plan should be the sum of the lengths of its separate chain elements. However, incorrect lengths are only flagged as a warning, and correct values are calculated.

B.3.4 Rules related to the design

The following rules mention the relationships between the routing plan and the design.

- if `{macro}` equals `{derivedmacro}` or there is no `{derivedmacro}` then `{macro}` should be the name of a declaration cell in the design (or library).
- `{derivedmacro}` should be the name of a declaration cell in the design (or library).
- The ports that are mentioned in `{clocks}` must be clock ports in the design.

`{instance}` would normally be the instance name of a cell in the design that is contained in cell `{macro}`. This is however not always necessary. E.g. one could define buffers in a scan chain of the routing plan which do not yet exist in the design.

B.3.5 Other rules

- {derivedmacro} and {replacedmacro} refer to the original ("template") macro for hierarchy expansion. The 'REPLACEDFOR' construct should be used when {macro} is a leaf macro (i.e. a macro of which the chains do not have chain elements) which refers to a substitution, e.g. the line 'MACRO dff_sff REPLACEDFOR dff' states that the declaration of flip-flop 'dff' is to be replaced by 'dff_sff'. In all other cases, the 'DERIVEDFROM' construct must be chosen, unless {macro} equals {derivedmacro}, in which case the 'DERIVEDFROM' construct is optional. This construct defines a different scan chain constellation.

- A clock domain that has no clock ports must be indicated with the 'NOTDRIVEN' construct. This construct will only be used for defining a scan chain through a combinatorial element which clearly has no clock ports. Multiple clock ports are allowed with the 'DRIVENBY' construct. This means that all the ports that are mentioned in the clock port list {clocks} should be connected at a higher level of the hierarchy (compare this to "MustJoin" in Edif). Please note that this construct does not refer to multi-phase clocking.

- The scan ports that are mentioned in each chain are all optional, with the exception that each chain should have at least one scan-in and one scan-out port. There can be only one scan-in port, but the number of other ports is not restricted.

- For each {instance} its type is given. If this type information is adequate for determining the right chain (because this type of macro has only one chain) then the 'CLOCKDOMAIN' and 'CHAIN' constructs are not necessary. If there is only one clock domain but this clock domain contains more than one chain, then only the 'CHAIN' construct is necessary. If there is more than one clock domain, then both 'CLOCKDOMAIN' and 'CHAIN' are necessary.

Author : M. van Balen

Title : **Comparing the NDS and VERA environments for scan chain recognition & removal**

Copy to :

M.T.M.	Segers	WAY-3	Nat.Lab
P.W.M.	Merkus	WAY-3	Nat.Lab
J.M.C.	Jonkheid	WAY-3	Nat.Lab
K.	Kuiper	WAY-3	Nat.Lab
R.J.J.	Stans	WAY-3	Nat.Lab
R.J.G.	Morren	WAY-3	Nat.Lab
H.A.	Bouwmeester	WAY-3	Nat.Lab
S.R.	Oostdijk	WAY-3	Nat.Lab
A.	van der Veen	WAY-3	Nat.Lab
J.	Bakker	WAY-3	Nat.Lab
R.M.C.	Lenarts	WAY-3	Nat.Lab
L.A.R.	Eerenstein	WAY-3	Nat.Lab
F.G.M.	de Jong	WAY-3	Nat.Lab
F.	van der Heyden	WAY-3	Nat.Lab
A.S.	Biewenga	WAY-3	Nat.Lab
M.N.M.	Muris	WAY-3	Nat.Lab
F.G.M.	Bouwman	WAY-4	Nat.Lab
A.P.	Kostelijk	WAY-4	Nat.Lab
E.J.	Marinissen	WAY-4	Nat.Lab
C.R.	Wouters	WAY-4	Nat.Lab
F.	Hapke	Semiconductors Hamburg	