Hardware Design Based on Verilog HDL

Gordon J. Pace
Balliol College

Oxford University Computing Laboratory
Programming Research Group

Trinity Term 1998

Thesis submitted for the degree of Doctor of Philosophy at the University of Oxford

Abstract

Up to a few years ago, the approaches taken to check whether a hardware component works as expected could be classified under one of two styles: hardware engineers in the industry would tend to exclusively use simulation to (empirically) test their circuits, whereas computer scientists would tend to advocate an approach based almost exclusively on formal verification. This thesis proposes a unified approach to hardware design in which both simulation and formal verification can co-exist.

Relational Duration Calculus (an extension of Duration Calculus) is developed and used to define the formal semantics of Verilog HDL (a standard industry hardware description language). Relational Duration Calculus is a temporal logic which can deal with certain issues raised by the behaviour of typical hardware description languages and which are hard to describe in a pure temporal logic. These semantics are then used to unify the simulation of Verilog programs, formal verification and the use of algebraic laws during the design stage. A simple operational semantics based on the simulation cycle is shown to be isomorphic to the denotational semantics. A number of laws which programs satisfy are also given, and can be used for the comparison of syntactically different programs.

The thesis also presents a number of other results. The use of a temporal logic to specify the semantics of the language makes the development of programs which satisfy real-time properties relatively easy. This is shown in a case study. The fuzzy boundary in interpreting Verilog programs as either hardware or software is also exploited by developing a compilation procedure to translate programs into hardware. Hence, the two extreme interpretations of hardware description languages as software, with sequential composition as the topmost operator (as in simulation), and as hardware with parallel composition as the topmost operator are exposed.

The results achieved are not limited to Verilog. The approach taken was carefully chosen so as to be applicable to other standard hardware description languages such as VHDL.

All of a sudden, it occurred to me that I could try again in a different way, more simple and rapid, with guaranteed success. I began making patterns again, correcting them, complicating them. Again I was trapped in this quicksand, locked in this maniacal obsession. Some nights I woke up and ran to note a decisive correction, which then led to an endless series of shifts. On other nights I would go to bed relieved at having found the perfect formula; and the next morning, on waking, I would tear it up. Even now, with the book in the galleys, I continue to work over it, take it apart, rewrite. I hope that when the volume is printed I will be outside it once and for all. But will this ever happen?

The Castle of Crossed Destinies
Italo Calvino, 1969

Acknowledgements

First of all, I would like to thank my supervisor, Jifeng He, for frequent and fruitful discussions about the work presented in this thesis. I would also like to thank all those in Oxford with similar research interests, discussions with whom regularly led to new and interesting ideas.

Thanks also go all my colleagues at the Department of Computer Science and Artificial Intelligence at the University of Malta, especially Juanito Camilleri, for providing a wonderful working environment during this past year.

Needless to say, I would like to thank all my friends, too numerous to list, who helped to make the years I spent in Oxford an unforgettable period of my life. Also, thanks go to my parents whose constant support can never be fully acknowledged.

Finally, but certainly not least, thanks go to Sarah, for encouragement, dedication and patience, not to mention the long hours she spent proof-reading this thesis.

I also wish to acknowledge the financial help of the Rhodes Trust, who partially supported my studies in Oxford.

Contents

Chapter 1

Introduction

1.1 Aims and Motivation

1.1.1 Background

System verification is one of the main raisons d'être of computer science. Various different approaches have been proposed and used to formally define the semantics of computer languages, and one can safely say that most computer scientists have, at some point in their research, tackled this kind of problem or at least a similar one. One particular facet of this research is hardware verification. Before discussing the possible role of computer science in hardware, it may be instructive to take a look at how hardware is usually developed in practice.

In contrast to software, checking hardware products by building and then testing on a number of 'typical' inputs and comparing the outputs with the expected results, can be prohibitively expensive. Building a prototype circuit after every iteration of the debugging process involves much more resources, ranging from time to money, than those involved in simply recompiling the modified source code.

This gave rise to the concept of simulation of hardware using software. If hardware can be simulated by software efficiently and correctly, then the expense of developing hardware can be reduced to that of software and, similarly, the order of the speed of development of hardware can be pushed up to be on the same level as that for software

This idea can be pushed further. Since the comparison of input and output pairs can be a tiresome and hence error prone process, given a description of a hardware product, one can build a shell around it to successively feed in a number of inputs and compare them to the expected outputs. Hence, simply by giving a table of results, an engineer can design and use a testing module which would protest if the given circuit failed any of the tests. This makes the debugging process easier, and furthermore the results are more easily reproducible.

But certain devices can be described more easily than by giving a whole table of input, output pairs. For some, a simple equation can suffice, while for others it may be necessary to use an algorithm. Hence, if the simulator is also given the ability to parse algorithms, one can check circuits more easily. For example, given a circuit and a claim that it computes the nth Fibonacci number (where n is the input), we can easily test the circuit for some inputs by comparing its output to the result of a short program evaluating Fibonacci numbers using recursion or iteration. If the algorithmic portions in the simulator can be somehow combined with the hardware description parts, one can also start with an algorithmic 'specification' and gradually convert it into a circuit.

This is the main motivation behind the development and design of hardware description languages. Basically, they provide a means of describing hardware components, extended with algorithmic capabilities, which can be efficiently simulated in software to allow easy and cheap debugging facilities for hardware.

As in software, hardware description languages (henceforth HDLs) would benefit greatly if research is directed towards formalising their semantics.

1.1.2 Broad Aims

In the hardware industry, simulation is all too frequently considered synonymous with verification. The design process usually consists of developing an implementation from a specification, simulating both in software for a number of different inputs and comparing the results. Bugs found are removed and the process repeated over and over again, until no new bugs are discovered. This procedure, however, can only show the presence of errors, not their absence.

On the other hand, formal methods cannot replace existing methods of hardware design overnight. Figure 1.1 proposes one possible framework via which formal methods may be introduced in hardware design. The approach is completely built upon formal techniques but includes simulation for design visualisation and development. Formal laws helping hardware engineers to correctly transform specifications into the implementation language are also included. The result is more reliability within an environment which does not require a complete revolution over current trends.

- Both the specification and implementation languages are formally defined within the same mathematical framework. This means that the statement: 'implementation I is a refinement of specification S' is formally defined and can be checked to be (or not to be) the case beyond any reasonable doubt.

- Design rules which transform a code portion from one format into another are used to help the designer transform a specification into a format more directly implementable as a hardware component. These rules are verified to be correct within the semantic domain, and hence the implementation is certain to be reliable.

- The relation between the simulator semantics and the semantics for the implementation language can also be shown. By formally defining the simulation cycle, we can check whether the semantics of a program as derived from the implementation language match the meaning which can be inferred from the simulation of the program.

The synthesis process is effectively a compilation task, which can be verified to be correct — that the resultant product is equivalent to (or a refinement of) the original specification code. However, since the design rules may not be complete, leaving situations unresolved, the inclusion of simulation within the formal model is used to help in the design process. This is obviously the area in which formal tools fit best. A verified hardware compiler which transforms high level HDL code into a format which directly represents hardware is one instance of such a tool.

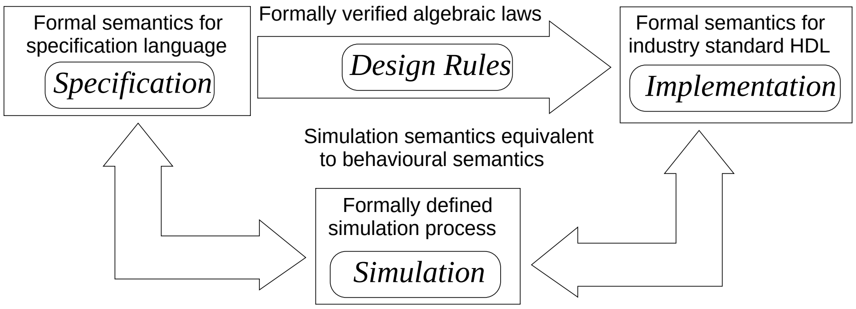


Figure 1.1: Simulation and verification: a unified approach

Since the design procedure is not always decidable, it may leave issues open, which it hands over to the designer to resolve. The simulator may then be used to remove obvious bugs, after which, it can be formally verified.

This is not the first time that a framework to combine simulation and formal verification techniques has been proposed. Other approaches have been proposed elsewhere in the literature.

In view of these facts, a formal basis for an HDL is of immediate concern. As in the case of software, one would expect that, in general, verifying large circuits would still be a daunting task. However, it should hopefully be feasible to verify small sensitive sub-circuits. This indicates the importance of a compositional approach which allows the joining of different components without having to restart the verification process from scratch.

Finally, another important aim is to analyse the interaction between software and hardware. A circuit is usually specified by a fragment of code which mimics the change of state in the circuit. Since most HDLs include imperative language constructs, a study of this feature will help anyone using an HDL to consider the possibility of implementing certain components in software. At the opposite end of the spectrum, software designers may one day include HDL-like constructs in their software languages to allow them to include hardware components within their code.

The aim of this research is thus twofold:

1. To investigate the formal semantics of a hardware description language
2. To study possible algebraic methods and formal hardware techniques which can aid the process of verification and calculation of products. These techniques will ideally be of a general nature and the ideas they used would be applicable to other HDLs.

1.1.3 Achievements

The original contributions presented in this thesis can be split into a number of different areas:

- At the most theoretical level, we present a variant of Duration Calculus, which can deal with certain situations which normal Duration Calculus fails to handle. Since the conception of Relational Duration Calculus, a number of similar extensions to Duration Calculus have sprung up independently. Despite this we still believe that Relational Duration Calculus can express certain constraints much more elegantly than similar calculi. A justification of this statement is given in chapter 3.

- Relational Duration Calculus is used to specify a semantics of a subset of Verilog. We do not stop at giving the semantics to this subset, but investigate other Verilog instructions where the more complex semantics expose better the inherent assumptions made in the exposition of the initially examined constructs.

- Algebraic laws allow straightforward syntactic reasoning about semantic content and are thus desirable. The semantics of Verilog are used to derive a suite of useful algebraic laws which allow comparison of syntactically different programs.

- The semantics are also used to transform specifications in different styles into Verilog programs. One of the two styles, real-time specifications, has been, in our opinion, largely ignored in formal treatments of HDLs. The other style shows how one can define a simple specification language and give a decision procedure to transform such a specification into Verilog code. Some examples are given to show how such a specification language can be used.

- Finally, the two opposing aspects in which one can view HDLs, as software or as hardware, is investigated in the last few chapters. On one hand, Verilog is simulated in software, essentially, by reducing it into a sequential program. On the other hand, certain Verilog constructs have a direct interpretation as hardware.

  - We start by showing that the denotational semantics of Verilog are sound and complete with respect to a simple version of Verilog simulation cycle semantics (given as an operational semantics). Essentially, this allows us to transform Verilog programs into sequential programs.
  - At the other end of the spectrum, we define, and prove correct, a compilation procedure from Verilog programs into a large collection of simple programs running in parallel. Most of these programs can be directly implemented in hardware.

  This allows a projection of Verilog programs in either of the two different directions.

As can be deduced from this short summary, the main contribution of this thesis is to give a unified view of all these issues for Verilog HDL. Some of the aspects treated in this thesis had previously been left unexplored with respect to HDLs in general.

In exploring these different viewpoints, we have thus realised a basis for the formal framework shown in figure 1.1 and discussed in the previous section.

1.1.4 Overview of Existent Research

Frameworks similar to the one proposed have been developed and researched time and time again for software languages. However, literature on the joining of the two, where hardware simulation and verification are put within the same framework, is relatively sparse [Cam90, Poo95, Tan88, Mil85b, Bry86, Pyg92] and such approaches with an industry standard HDL are practically non-existent.

Most research treating hardware description languages formally tends to concentrate on academically developed languages, such as CIRCAL and RUBY. When one considers the complexities arising in the semantics of real-life hardware description languages, this is hardly surprising.

Formal studies of industry standard HDLs have been mostly limited to VHDL[1] [BGM95, BSC$^+$94, BSK93, Dav93, Goo95, KB95, Tas90, TH91, Tas92, WMS91]. As with hardware models, the approaches used are extremely different, making comparison of methods quite difficult. Since the semantics of the language is intimately related to the simulation cycle, which essentially sequentialises the parallel processes, most of the approaches are operational. These approaches usually help by giving a formal documentation of the semantics of the simulation language, but fail to provide practical verification methods.

Another popular standard HDL is Verilog[Ope93, Tho95, Gol96]. Although it is as widespread as VHDL, Verilog has not yet attracted as much formal work [Bal93, Gor95, GG95, Pro97, PH98]. Paradoxically, this fact may indicate a more promising future than for VHDL. Most formal work done on VHDL refers back to the informal official documentation. In Verilog, a paper addressing the challenges posed by the language and giving a not-so-informal description of the semantics was published before a significant amount of work on the language appeared [Gor95]. Hopefully, researchers will use this as a starting point or at least compare their interpretation of the semantics with this. Also, the research carried out in developing formal techniques for the treatment of VHDL is almost directly applicable to Verilog. If researchers in Verilog stick to the clearer semantics presented in [Gor95] rather than continuously referring back to the language reference manual, and follow guidelines from the lessons learnt from VHDL, formal research in this area may take a more unified approach.

Practical verification methods and verified synthesis procedures, which transform specifications into a form directly implementable as hardware, are probably the two main ultimate goals of the research in this field. Currently, we are still very far from these ultimate targets, and a lot of work still has to be done before positive results obtained will start to change the attitude of industries producing hardware with safety-critical components towards adopting these methods.

1.1.5 Choosing an HDL

One major decision arising from this discussion, is what HDL to use. We can either use an industry standard language or use one which has been developed for academic use or even develop one of our own.

The first choice is whether or not to define an HDL from scratch. The advantages of starting from scratch are obvious: since we are in control of the design of the language, we can select one which has a relatively simple behavioural and simulation semantics. On the other hand, this would mean that less people (especially from industry) would make an effort to follow the work and judge whether the approach we are advocating is in fact useful in practice. Defining our own language would also probably mean that certain features commonly used by hardware engineers may be left out for the sake of a simple formalism, with the danger that this choice may be interpreted as a sign that the formal

[1] VHSIC Hardware Description Language, where VHSIC stands for 'Very High Speed Integrated Circuits'[Per91, LMS86]

approach we are advocating is simply not compatible with real-life HDLs. In view of these facts it is probably more sensible to work with an already existent HDL.

However, we can choose to avoid the problem by using an already existent academically developed HDL such as RUBY or CIRCAL. This is quite attractive since the formal semantics of such HDLs have already been defined, and it is usually the case that these semantics are quite elegant. In comparison, industrially used HDLs tend to be large and clumsy, implying involved semantics which are difficult to reason with and not formally defined anywhere. Despite these advantages, the difficulty in getting industry interested in the techniques developed for non-standard HDLs is overwhelming.

This means that the only way of ensuring that the aims of the approach proposed are feasible, would be to grit our teeth and use an industry standard HDL, possibly restricting the language reasonably in order to have simpler semantics. The main aim of this thesis is to show how industrial methods such as simulation can be assisted (as opposed to just replaced) by formal methods. Working with a subset of a industry standard HDL means that there is a greater chance of persuading hardware engineers to look at the proposed approach and consider its possible introduction onto their workbench.

The final problem which remains is that of choosing between the standard industrial HDLs available. This choice can be reduced to choosing between the two most popular standards: Verilog or VHDL. These are the HDLs which can best guarantee a wide hardware engineer audience.

1.1.6 Verilog or VHDL?

It is important to note that this is a relatively secondary issue. The main aim of this thesis is not to directly support a particular HDL, but to present a framework within which existent HDLs can be used. We will avoid lengthy discussions regarding particular features of a language which are not applicable to other standard HDLs. Having said this, it is obviously important to choose one particular language and stick to it. If not, we might as well have developed our own HDL.

When choosing an HDL to work on, the first that tends to spring to mind is VHDL. It is usually regarded as *the* standard language for hardware description and simulation. A single standard language avoids repetition of development of libraries, tools, etc. This makes VHDL look like a very likely candidate. Furthermore, quite a substantial amount of work has also been done on the formalisation of the VHDL simulation cycle and the language semantics which can help provide further insight into the language over and above the official documentation.

Another choice is Verilog HDL. Verilog and VHDL are the two most widely used HDLs in industry. Verilog was developed in 1984 by Phil Moorby at Gateway Design Automation as a propriety verification and simulation product. In 1990, the IEEE Verilog standards body was formed, helping the development of the language by making it publicly available. Unlike VHDL, which is based on ADA, Verilog is based on C. It is estimated that there are as many as 25,000 Verilog designers with 5,000 new students trained each year[Gor95]. This source also claims that Verilog has a 'wide, clear range of tools' and has 'faster simulation speed and efficiency' than VHDL.

Research done to formalise Verilog is minimal. However, in [Gor95], M.J.C. Gordon informally investigated the language and outlined a subset of the language called V. Although the approach is informal, the descriptions of the simulation cycle and language semantics are articulated in a clearer way than the official language documentation. If this description is accepted as a basis for the language by the research community, it would be easier to agree on the underlying semantics. The sub-language, V, is also an advantage. In research on VHDL, different research groups came up with different subsets of the language. V provides researchers working on Verilog with a small manageable language from which research can start. Although, the language may have to be even further reduced or enhanced to make it more suited to different formal approaches, the common sub-language should help to connect work done by different research groups.

Finally, Verilog provides all the desirable properties of an HDL. It provides a structure in which one can combine different levels of abstraction. Both structural and behavioral constructs are available, enabling design approaches where a behavioral definition is gradually transformed into a structural one which may then be implemented directly as hardware.

The competition seems to be quite balanced. VHDL, being more of an acknowledged standard, is very tempting. However, Verilog is widely used and is (arguably) simpler than VHDL. The definition of a sub-language of Verilog and a not-so-informal definition of the semantics, before any substantial research has been carried out are the decisive factors. They solve a problem encountered by many VHDL researchers by attempting to firmly place a horse before anyone even starts thinking about the cart.

Most importantly, however, I would like to reiterate the initial paragraph and emphasise that the experience gained in either of the languages can usually be applied to the other. Thus, the application of the semantics of these languages and techniques used to describe the semantics is where we believe that the main emphasis should be placed.

1.2 Details of the Approach Taken

1.2.1 General Considerations

The first decision to be taken was that of selecting the kind of approach to be used in defining the semantics of the language. Most interpretations of the VHDL semantics have been defined in terms of an operational semantics. Although the definitions seem reasonably clear and easy to understand, in most cases the result is a concrete model which can make verification very difficult. Operational semantics are implementation-biased and cannot be used to link a design with its abstract specification directly. A more abstract approach seems necessary for the model to be more useful in practice.

1.2.2 Temporal Logic

Previous research has shown the utility of temporal logic in specifying and verifying hardware and general parallel programs in a concise and clear manner [Boc82, HZS92, MP81, Mos85, Tan88]. Other approaches used include, for example, Higher Order Logic (HOL) [CGM86], relational calculus, petri nets, process algebras such as CSP or CCS, etc. Temporal logics appear to be more suitable for our purpose of defining the semantics of Verilog. It avoids the explicit use of time variables, has a more satisfactory handling of time than CSP or CCS and handles parallel processes in an easier way than pure Relational Calculus.

A deeper analysis of Verilog shows that standard temporal logics are not strong enough to define the semantics of the language completely. The mixture of sequential zero-delay and timed assignments cannot be readily translated into a temporal logic. Zero time in HDLs is in fact interpreted as an infinitesimally small (but not zero) delay. Standard temporal logics give us no alternative but to consider it as a zero time interval (at a high level of abstraction), which can cause some unexpected and undesirable results. One solution was to upgrade the temporal logic used to include infinitesimal intervals. This is, in itself, a considerable project. A less drastic but equally effective solution is to embed a relational calculus approach within a temporal logic.

The next decision dealt with the choice of temporal logic to use. Interval Temporal Logic [Mos85] and Discrete Duration Calculus [ZHR91, HO94] seemed to be the most appropriate candidates. Eventually, Discrete Duration Calculus was chosen because of its more direct handling of clocked circuits and capability of being upgraded to deal with a dense time space if ever required.

1.3 Overview of Subsequent Chapters

The thesis has been divided into four main parts, each of which deals with a different aspect of the formalisation of the semantics of Verilog HDL.

Part I: This part mainly serves to set the scene and provide the background and tools necessary in later parts.

Chapter 2 gives a brief overview of Verilog, concentrating on the subset of the language which will be subsequently formalised.

Chapter 3 starts by describing vanilla Duration Calculus as defined in [ZHR91]. It is followed by an original contribution — an extension to the calculus to deal with relational concepts: Relational Duration Calculus, which will be necessary to define the formal semantics of Verilog.

Expressing real-time properties in Duration Calculus can sometimes yield daunting paragraphs of condensed mathematical symbols. But abstraction is what makes mathematics such a flexible and useful tool in describing the real world. **Chapter 4** gives a syntactic sugaring for Duration Calculus to allow concise and clear descriptions of real-time properties, based on the work given in [Rav95].

Part II: We now deal with the formalisation of Verilog HDL.

Chapter 5 gives a formal interpretation of the semantics of a subset of Verilog using Relational Duration Calculus, and discusses how this subset can be enlarged.

Algebraic reasoning is one of the major concerns of this thesis. **Chapter 6** gives a number of algebraic laws for Verilog based on the semantics given in the previous chapter.

Part III: Two case studies are given to assess the use of the semantics.

In **chapter 7** we specify a rail-road crossing (as given in [HO94]) and transform the specification into a Verilog program which satisfies it.

A different approach is taken in **chapters 8** and **9**. In this case, a simple hardware specification language is defined, which can be used to state some basic properties of circuits. Using this language, a number of simple circuits are specified. A number of algebraic laws pertaining to this specification language are then used on these examples to show how it can be used in the decomposition of hardware components and their implementation in Verilog.

Part IV: Finally, the last part investigates the relationship between hardware and software inherent in Verilog specifications.

Chapter 10 looks at the conversion of Verilog code into a smaller subset of Verilog which has an immediate interpretation as hardware. Essentially, the transformations proved in this chapter provide a compilation process from Verilog into hardware.

Chapter 11 shows how a simplification of the Verilog simulation cycle, interpreted as an operational semantics of the language, guarantees the correctness of the semantics given earlier.

Lastly, the twelfth and final chapter gives a short resumé and critical analysis of the main contributions given in this thesis and investigates possible extensions which may prove to be fruitful.

Part I

This part of the thesis sets the scene for the rest of the work by introducing the basic theory and notation necessary. Chapter 2 describes the subset of the Verilog HDL which will be formalised. Chapter 3 introduces the Duration Calculus and Relational Duration Calculus which will be used to define the semantics of Verilog. Finally, chapter 4 defines a number of real-time operators which will be found useful in later chapters.

Chapter 2

Verilog

2.1 Introduction

This chapter gives an informal overview of Verilog. Not surprisingly, this account is not complete. However, all the essentials are treated in some depth, and this obviously includes the whole of the language which will be treated formally later. For a more comprehensive description of the language a number of information sources are available. [Tho95, Gol96] give an informal, almost tutorial-like description of the language, while [Ope93, IEE95] are the standard texts which one would use to resolve any queries. Finally, a concise, yet very effective description of a subset of Verilog and the simulation cycle can be found in [Gor95].

2.1.1 An Overview

A Verilog program (or specification, as it is more frequently referred to) is a description of a device or process rather similar to a computer program written in C or Pascal. However, Verilog also includes constructs specifically chosen to describe hardware. For example, in a later section we mention wire and register type variables, where the names themselves suggest a hardware environment.

One major difference from a language like C is that Verilog allows processes to run in parallel. This is obviously very desirable if one is to describe the behaviour of hardware in a realistic way.

This leads to an obligatory question: How are different processes synchronised? Some parallel languages, such as Occam, use channels where the processes run independently until communication is to take place over a particular channel. The main synchronisation mechanism in Verilog is variable sharing. Thus, one process may be waiting for a particular variable to become true while another parallel process may delay setting the particular variable to true until it is out of a critical code section.

Another type of synchronisation mechanism is the use of simulation time. At one level, Verilog programs are run under simulators which report the values of selected variables. Each instruction takes a finite amout of resource time to execute and one can talk about speed of simulators to discuss the rate at which they can process a given simulation. On the other hand, Verilog programs also execute in simulation time. A specification may be delayed by 5 seconds of simulation time. This does not mean that simulating the module will result in an actual delay of 5 seconds of resource time before the process resumes, but that another process which takes 1 second of simulation time may execute 5 times before the first process continues. This is a very important point to keep in mind when reading the following description of Verilog. When we say that 'statement so-and-so takes no time to execute' we are obviously referring to simulation time — the simulator itself will need some real time to execute the instruction.

2.2 Program Structure

A complete specification is built from a number of separate modules. Each module has a number of input and output ports to enable communication with the outside world. The module body relates the inputs and outputs. The top level module then specifies a complete system which can be executed by a Verilog simulator. Two types of module body descriptions can be used: structural and behavioural.

A structural description of a module contains information about how the wires in the system are connected together, possibly instantiating a number of modules in the process. Essentially, a structural description serves as the interface of the module. A simple example of a structural description of a half adder is:

```
module HALF_ADDER_STRUCT(in1, in2, cout, sout);
   input in1, in2; output cout, sout;

   AND AND_1(in1, in2, cout);
   XOR XOR_1(in1, in2, sout);

endmodule
```

The meaning of the module is self-evident: A half adder module has two inputs and two outputs. An `AND` module and a `XOR` module are instantiated for every half-adder, naming them `AND_1` and `XOR_1` respectively. The inputs and outputs of the half-adder are then 'rewired' into these gates.

On the other hand, a behavioural description of a module describes what the module actually does — how the output is calculated from the input. A behavioural description of a half-adder may look like:

```
module HALF_ADDER_BEH(in1, in2, cout, sout);
   input in1, in2; output cout, sout;

   assign cout = (in1+in2) / 2;
   assign sout = (in1+in2) % 2;

endmodule
```

The assign statements make sure that whenever the right hand side expression changes value so does the variable being assigned to.

Obviously, since structural modules are built of modules themselves, this nesting must end at some level. Verilog, however, provides a number of standard modules which provide a convenient level at which to stop describing structure.

The description of these in-built modules is beyond the scope of this chapter and will not be described any further.

It is also interesting to note that in engineering circles, modules are sometimes described in both ways. Both modules are then combined into a single module together with a test module which feeds a variety of inputs to both modules, and compares the outputs. Since behavioural modules are generally easier to understand and write than structural ones, this is sometimes considered to be an empirical basis for 'correctness'.

2.3 Signal Values

Different models allow different values on wires. The semantic model given in this thesis deals with a simple binary (`1` or `0`) type. This is extendable to deal with two extra values `z` (high impedance) and `x` (unknown value). Other models allow signals of different strengths to deal with ambiguous situations.

Sometimes it is necessary to introduce internal connections to feed the output of a module into another. Verilog provides two types of signal propagation devices: wires and registers. Once assigned a value, registers keep that value until another assignment occurs. In this respect, registers are very similar to normal program variables. Wires, on the other hand, have no storage capacity, and if undriven, revert to value `x`.

In this chapter we will be discussing only register variables. For a discussion of wire type variables one can refer to any one of the main references given at the beginning of the chapter.

2.4 Language Constructs

Behavioural descriptions of modules are built from a number of programming constructs some of which are rather similar to ones used in imperative programming languages. Four different types of behavioural modules are treated here:

Continuous assignments: A module may continuously drive a signal on a variable. Such a statement takes the form:

```
assign v=e
```

This statement ensures that variable `v` always has the value of expression `e`. Whenever the value of a variable in expression `e` changes, an update to variable `v` is immediately sent.

Delayed continuous assignments: Sometimes, it is desirable to wait for the expression to remain constant for a period of time before the assignment takes place. This is given by:

```
assign v = #n e
```

If the variables of `e` change, an assignment is scheduled to take place in `n` units time. Furthermore, any assignments scheduled to take place *before* that time are cancelled. Note that if `v` can be written to by only one such statement, the effect is equivalent to making sure that after the variables in `e` change, they must remain constant for at least `n` time units if the value of `v` is to be updated. This type of delay is called *inertial delay.*

One-off modules: It is sometimes desirable to describe the behaviour of a module using a program. The following module starts executing the program P as soon as the system is started off. Once P has terminated, the module no longer affects any wires or registers.

```
initial P
```

Looping modules: Similarly, it is sometimes desirable to repeatedly execute a program P. In the following module, whenever P terminates, it is restarted again:

```
forever P
```

The different modules are executed in 'parallel' during simulation. Hence, we can construct a collection of modules which outputs `a_changed`, a variable which is true if and only if the value of `a` has just changed:

```
assign a_prev    = #1 a
assign a_changed = a xor a_prev
```

2.4.1 Programs

The syntax of a valid program is given in table 2.1. As can be seen, a program is basically a list of semi-colon separated instructions. If the sequence of instructions lies within a `fork` ... `join` block, the instructions are executed in parallel, otherwise they are executed in sequence, one following the other. These two approaches can be mixed together. Note that if a parallel block is put into sequential composition with another block, as in `fork P;Q join; R`, R starts only after *both* P and Q have terminated.

Instruction blocks can be joined together into a single instruction by the use of `begin` and `end`. This allows, for example, the embedding of sequential programs in parallel blocks as shown in the following program:

```
fork begin P;Q end; R join
```

In this case, Q must wait for P to terminate before executing. On the other hand, R starts at the same time as P.

The instructions we will be expounding can be split into 3 classes: guards, assignments and compound statements.

Guards

Timed guards: `#n` stops the current process for `n` simulation time units after which control is given back.

Value sensitive guards: A program can be set to wait until a particular condition becomes true. `wait e`, where `e` is an expression, does just this.

Edge sensitive guards: The processing of a process thread can also be paused until a particular event happens on one of the variables. There are three main commands: `@posedge v` waits for a transition from a value which is not 1 to 1 on `v`; `@negedge v` waits for a falling edge on variable `v`; and `@v` which waits for either event.

Complex guards: A guard can also be set to wait until either of a number of edge-events happen. `@(G1 or G2 or ... Gn)`, waits until any of `@G1` to `@Gn` is lowered.

$\langle prog \rangle$	$::=$	$\langle instr \rangle$
	$\mid$	$\langle instr \rangle$ `;` $\langle prog \rangle$
$\langle edge \rangle$	$::=$	$\langle var \rangle$
	$\mid$	`posedge` $\langle var \rangle$
	$\mid$	`negedge` $\langle var \rangle$
	$\mid$	$\langle edge \rangle$ `or` $\langle edge \rangle$
$\langle guard \rangle$	$::=$	$\#\ \langle number \rangle$
	$\mid$	`wait` $\langle var \rangle$
	$\mid$	`@` $\langle edge \rangle$
$\langle inst \rangle$	$::=$	`begin` $\langle prog \rangle$ `end`
	$\mid$	`fork` $\langle prog \rangle$ `join`
	$\mid$	$\langle guard \rangle$
	$\mid$	$\langle var \rangle = \langle expr \rangle$
	$\mid$	$\langle var \rangle = \langle guard \rangle\ \langle expr \rangle$
	$\mid$	$\langle guard \rangle\ \langle var \rangle = \langle expr \rangle$
	$\mid$	$\langle var \rangle <= \langle guard \rangle\ \langle expr \rangle$
	$\mid$	`if` $(\langle expr \rangle)\ \langle inst \rangle$
	$\mid$	`if` $(\langle expr \rangle)\ \langle inst \rangle$ `else` $\langle inst \rangle$
	$\mid$	`while` $(\langle expr \rangle)\ \langle inst \rangle$
	$\mid$	`do` $(\langle expr \rangle)$ `while` $\langle inst \rangle$
	$\mid$	`forever` $\langle inst \rangle$
	$\mid$	`fork` $\langle prog \rangle$ `join`

Table 2.1: The syntax of a subset of Verilog

Assignments

Immediate assignments: Assignments of the form `v=e` correspond very closely to the assignment statements normally used in imperative programming languages like C and Pascal. The variable `v` takes the value of expression `e` without taking any simulation time at all.

Blocking guarded assignments: There are two types of blocking assignment statements: `g v=e` and `v=g e`, where `g` is a guard. Their semantics are quite straightforward — `g v=e` behaves just like `g; v=e` and `v=g e` behaves like `v'=e; g; v=v'` (where `v'` is a fresh variable, unused anywhere else).

Non-blocking guarded assignments: The assignment `v<=g e` acts just like `v=g e`, but does not block the execution of any code appearing after it. Thus, for any program `P`, `v<=g e; P` acts like:

`fork v=g e; P join`[1]

Each of these types of assignments can be used to assign a number of variables in parallel. For example, `v1, v2, ...vn = e1, e2, ...en` is interpreted as the assignment which starts by calculating the values of expressions `e1` up to `en` (using the old values of variables `v1` to `vn`) and then assigning these values to the variables at one go. Hence, after executing `v=0 ; v,w=1,v`, `v` and `w` have the values `1` and `0` respectively.

Constructs

Conditional: The conditional statements `if (b) P` and `if (b) P else Q` act just like their counterpart in imperative programming languages like C and Pascal. The value of `b` is evaluated and, if it is evaluated to `1`, execution follows through the first branch, otherwise through the second branch.

Loops: `while (b) P` corresponds exactly once again to its counterpart in imperative programming. `b` is evaluated and, if true, `P` is executed. At the end of the execution, control is placed once again at the beginning of the `while` loop. When `b` is evaluated and is found to be `0`, the loop terminates.

`do P while (b)` acts in a similar fashion, but checks the value of the expression at the end of the executions, rather than at the start.

`forever P` is used for non-terminating loops, acting just like `while (1) P`.

There is obviously much more to Verilog than this. However, this probably constitutes the main core of ideas behind the whole language. The informal presentation of the semantics in this section, still leaves a wide range of possible behaviours. We will reduce this by giving the *simulation cycle* semantics of the language. This will, essentially, describe how the language is executed on a simulator and will therefore remove most of the issues still unclear up to this point.

[1] Note that this is only meant as an informal description of the semantics. If we were to use this definition, one of the conditions would have to be that `P` is the rest of the sequential program. Otherwise, we would have situations where `(v<=g e; P); Q` does not act like `v<=g e;(P;Q)`, which is undesirable, since it would mean that sequential composition is not associative.

2.4.2 Example Programs

A computer science or programming background gives the necessary intuition to deal with the procedural (imperative) subset of Verilog. The continuous assignments and the concept of simulation time, on the other hand, correspond very closely to ideas in hardware. The main source of confusion one usually encounters is when the two meet. This section gives a small example to help clarify these ideas. For the sake of a high interest-to-size ratio, the example will use integer variables.

Any self-respecting programmer can produce a solution to the calculation of the Nth Fibonacci number in imperative Verilog:

```
prev=1; curr=1; n=2;
while (n<N)
   prev, curr, n = curr, curr+prev, n+1;
```

Since, the answer may be needed by another module, an engineer may choose to package the procedure differently. If we call the above program P:

```
wait start;
P
finish = #1 1;
finish<= #1 0
```

The procedure now waits for the signal `start` to become true before it starts its execution. Upon termination, it delivers a short high signal on variable `finish`. This allows other procedures to know when the value of the Nth fibonacci number is available.

What about another module which needs the seventh fibonacci number?

```
N=7;
start = #1 1;
start<= #1 0;
wait finish;
Q
```

This program writes 7 to variable `N` and sends a start signal. It then waits for signal `finish` to turn to true, which signifies that the other module is done processing. This module may now safely run a program Q which uses the seventh fibonacci number.

Note that these programs work no matter how long the first program takes to calculate the seventh (or indeed any) fibonacci number.

2.5 The Simulation Cycle

The most obvious way of reducing the ambiguity in the above description of Verilog constructs is by explaining how the constructs are interpreted by a simulator. The description given here is based on [Ope93] and is also very similar to the one given is [Gor95]. In particular, the introduction of the guards $\Delta\langle edge\rangle$ and a new assignment statement `v←#n e` are taken directly from the latter reference.

The state of the execution consists of the current simulation time, a function giving the current values of registers and two sets of threads. All threads consist of the code they execute (together with a marker where the next instruction to be executed lies), their status and possibly, a pending assignment. The status of a thread can be one of:

- Enabled
- Delayed until t
- Guarded by g
- Finished

The two sets of threads are called statement threads and update threads.

2.5.1 Initialisation

Initially, the variables are all set to the unknown state `x` and the simulation time is reset to 0. Each module is placed in a different statement thread with the execution point set at the first instruction and with its status enabled.

Simulation Cycle

We start by describing what the execution of a statement in a thread does. This obviously depends on the instruction being executed.

Guards: For each type of guard, we specify the actions taking place. We will add a new type of guard Δ`v`, where `v` is an edge. The reason will become apparent later.

- `#n`: The status of the thread is set to: *delayed until t,* where t is `n` more than the current simulation time.
- `wait v`: If `v` is true, the thread remains enabled and the execution point is advanced. Otherwise, the thread becomes blocked by a guard which is lowered when `v` becomes true.
- Δ`e`: changes the status to *guarded by* `e`, where `e` is the guard at the execution point.
- `@v`, `@posedge v`, `@negedge v` and `@(g1 or g2 ... or gn)`: behave just like the guards Δ`v; #0`, Δ`posedge v; #0`, Δ`negedge v; #0` and Δ`(g1 or g2 ... or gn) ;#0` respectively. The implications of this interpretation are discussed in more detail at the end of this section, once the whole simulation cycle has been described.

Assignments: We consider the different type of assignments separately:

- `v=e`: `e` is evaluated and the resulting value is written to variable `v`.
- `g v=e`: is treated just like `g; v=e`.
- `v=g e`: `e` is evaluated to α and the thread is set guarded by `g`. A pending assignment `v=`α is also added to the thread.
- `v<=g e`: `e` is evaluated to α and a new update thread is created guarded by `g` and with an assignment statement `v=`α as its only code.

After this, the execution point of the code is advanced if there are any further instructions. Otherwise, the status of the thread is set to *finished.*

Compound Statements:

- `if (b) P`: `b` is evaluated and the execution point is moved to `P` if true, but moved to the end of the whole instruction otherwise.
- `if (b) P else Q`: `b` is evaluated and the execution point is advanced into `P` or `Q` accordingly.
- `while (b) P`: is replaced by `if (b) begin P; while (b) P end`.
- `forever P`: is replaced by `while (1) P`
- `fork P join`: Creates as many threads as there are statements in `P` and sets them all as enabled.
- `begin P end`: is simply replaced by `P`.

The complete simulation cycle can now be described:

1. If there are any enabled statement threads, the simulator starts by picking one. If it has a pending assignment it is executed and the pending assignment is removed, otherwise a step is made along the statement code. If a change which has just occurred on a variable lowers other threads' guards, these threads have their status modified from *guarded* to *enabled,* and the simulator goes back to step 1.
2. If there are any threads delayed to the current simulation time they are enabled and the simulator restarts from step 1.
3. If any update threads are enabled, they are carried out in the order they were created and the threads are deleted. If the updates have lowered guards on some threads, these have their status reset to *enabled.* Control returns back to step 1.
4. Finally, we need to advance time. The simulator chooses the lowest of the times to which statement threads are delayed. The simulation is set to this time and all threads enabled to start at this time are enabled. Control is given back to step 1.

Now, the difference between `@e` and Δ`e` should be clearer. In (Δ`e;` P), P becomes enabled immediately upon receiving the required edge `e`. On the other hand, in (`@e;` P), P is only enabled once the required edge is detected and all other enabled statements in parallel threads are exhausted.

2.5.2 Top-Level Modules

This description of the simulation cycle is limited to programs. How are modules first transformed into programs?

Modules in the form `initial P` are transformed to `P`, and modules in the form `always P` are transformed to `forever P`.

Intuitively, one would expect that `assign v=e` to be equivalent to the module `always @(v1 or ... or vn) v=e` where `v1` to `vn` are the variables used in expression `e`. Reading the official documentation carefully, however, shows that this is not so. When the `assign` statement is enabled (by a change on the variables of `e`), the assignment is executable immediately. In the other situation, after lowering the guard `@(v1 or ... vn)`, one still has to wait for all other enabled statements to be executed before the assignment can take place. Thus, the continuous assignment is treated in a way analogous to:

$$\texttt{always } \Delta\texttt{(v1 or ... vn) v=e}$$

This leaves only delayed continuous assignments to be dealt with. The instruction `always v=#n e` is equivalent to:

$$\texttt{always } \Delta\texttt{(v1 or ... vn) v}\leftarrow\texttt{ \#n e}$$

`v`$\leftarrow$`#n e` is another type of assignment acting as follows: the current value of `e` is evaluated (α) and a new statement thread, delayed to `n` time units more than the current simulation time, is created. This new thread has the assignment `v=`α as its code. Furthermore, all similar statement threads with code `v=`α and which are delayed to earlier times are deleted (hence achieving the inertial effect).

2.6 Concluding Comments

We have presented the informal semantics of Verilog in two distinct ways. In our opinion, neither presentation explains the language in a satisfactory manner. The first presentation lacks enough detail and is far too ambiguous to be used as a reference for serious work. The second is akin to presenting the semantics of C by explaining how a C compiler should work. This is far too detailed, and despite the obvious benefits which can be reaped from this knowledge, one can become a competent C programmer without knowing anything about its compilation. Having more space in which to expound their ideas, books about Verilog usually take an approach similar to our first presentation but give a more detailed account. They still lack, however, the elegance which can regularly be found in books expounding on 'cleaner' languages such as C and Occam. The main culprits are shared variables and certain subtle issues in the simulation cycle (such as the separate handling of blocking and non-blocking assignments). This indicates that the language may need to be 'cleaned up' before its formal semantics can be specified in a useful manner. This will be discussed in more detail in chapter 5.

Chapter 3

Relational Duration Calculus

3.1 Introduction

Temporal logics are a special case of modal logic [Gol92], with restrictions placed on the relation which specifies when a state is a successor of another. These restrictions produce a family of states whose properties resemble our concept of time. For example, we would expect transitivity in the order in which events happen: if an event A happens before another event B which, in turn, happens before C, we expect A to happen before C.

Applications of temporal logics in computer science have been investigated for quite a few years. The applications include hardware description, concurrent program behaviour and specification of critically timed systems. The main advantage over just using functions over time to describe these systems is that temporal logics allow us to abstract over time. We can thus describe systems using more general operators which makes specifications shorter, easier to understand and hence possibly easier to prove properties of. They have been shown to be particularly useful in specifying and proving properties of real-time systems where delays and reaction times are inherent and cannot be ignored.

The temporal logic used here is the Duration Calculus. It was developed by Chaochen Zhou, C.A.R. Hoare and Anders P. Ravn in 1991 [ZHR91]. As with other temporal logics, it is designed to describe systems which are undergoing changes over time. The main emphasis is placed on the particular state holding for a period of time, rather than just for individual moments. Several applications of Duration Calculus (henceforth DC) and extensions to the calculus have been given in the literature. The interested reader is referred to [Zho93, ZHX95a, ZRH93, ZX95, Ris92, HRS94, HO94, HH94, HO94, MR93] to mention but a few. [HB93] gives a more formal (and typed) description of DC than the one presented here where the typed specification language Z [Spi92] is used for the description.

3.2 Duration Calculus

3.2.1 The Syntax

We will start by describing briefly the syntax of the DC. The natural and real numbers are embedded within DC. Then, we have a set of *state variables*, which

$$
\begin{array}{lcl}
\langle\text{natural number}\rangle & & \\
\langle\text{state variables}\rangle & & \\
\langle\text{state expression}\rangle & ::= & \langle\text{state variables}\rangle \\
 & | & \langle\text{state expression}\rangle \wedge \langle\text{state expression}\rangle \\
 & | & \langle\text{state expression}\rangle \vee \langle\text{state expression}\rangle \\
 & | & \langle\text{state expression}\rangle \Rightarrow \langle\text{state expression}\rangle \\
 & | & \langle\text{state expression}\rangle \Leftrightarrow \langle\text{state expression}\rangle \\
 & | & \neg\langle\text{state expression}\rangle \\
\langle\text{duration formula}\rangle & ::= & \int\langle\text{state expression}\rangle = \langle\text{natural number}\rangle \\
 & | & \lceil\langle\text{state expression}\rangle\rceil \\
 & | & \langle\text{duration formula}\rangle \wedge \langle\text{duration formula}\rangle \\
 & | & \langle\text{duration formula}\rangle \vee \langle\text{duration formula}\rangle \\
 & | & \langle\text{duration formula}\rangle \Rightarrow \langle\text{duration formula}\rangle \\
 & | & \langle\text{duration formula}\rangle \Leftrightarrow \langle\text{duration formula}\rangle \\
 & | & \neg\langle\text{duration formula}\rangle \\
 & | & \langle\text{duration formula}\rangle \mathbin{\textbf{;}} \langle\text{duration formula}\rangle \\
 & | & \exists\langle\text{state variable}\rangle \cdot \langle\text{duration formula}\rangle
\end{array}
$$

Table 3.1: Syntax of the Duration Calculus

are used to describe the behaviour of states over time. The constant state variables **1** and **0** are included.

The state variables may be combined together using operators such as $\wedge$, $\vee$ and $\neg$ to form state expressions. These *state expressions* are then used to construct *duration formulae* which will tell us things about time intervals, rather than time points (as state variables and state expressions did). Table 3.1 gives the complete syntax.

We will call the set of all state variables $\mathcal{SV}$ and the set of all duration formulae $\mathcal{DF}$.

3.2.2 Lifting Predicate Calculus Operators

Before we can start to define the semantics of Duration Calculus, we will recall the *lifting* of a boolean operator over a set. This operation will be found useful later when defining DC.

Since the symbols usually used for boolean operators are now used in DC (see table 3.1), we face a choice. We can do either of the following:

- overload the symbols eg $\wedge$ is both normal predicate calculus conjunction and the symbol as used in DC, or
- use alternative symbols for the boolean operators

The latter policy is adopted for the purposes of defining the semantics of DC so as to avoid confusion. The alternative symbols used are: $\sim$ is *not*, $\cap$ is *and*, $\cup$ is *or*, $\rightarrow$ is *implies* and $\leftrightarrow$ is *if and only if*.

Given any n-ary boolean operator $\oplus$, from $\mathbb{B}^n$ to $\mathbb{B}$, and an arbitrary set S, we can define the lifting of $\oplus$ over S, written as $\overset{S}{\oplus}$.

$\overset{S}{\oplus}$ is also an n-ary operator which, however, acts on functions from S to $\mathbb{B}$, and returns a function of similar type.

$$LIFT == S \longrightarrow \mathbb{B}$$
$$\overset{S}{\oplus} :: LIFT^n \longrightarrow LIFT$$

Informally, the functional effect of lifting an operator $\oplus$ over a set S is to apply $\oplus$ pointwise over functions of type $LIFT$. In other words, the result of applying $\overset{S}{\oplus}$ to an input $(a_1, a_2, \ldots a_n)$ is the function which, given $s \in S$ acts like applying each a_i to s and then applying $\oplus$ to the results.

$$\overset{S}{\oplus} (a_1 \ldots a_n)(s) \stackrel{def}{=} \oplus (a_1(s) \ldots a_n(s))$$

For example, if we raise the negation operator $\sim$ over a set $\mathbb{T}$ we get an operator $\overset{\mathbb{T}}{\sim}$ which given a function from time ($\mathbb{T}$) to boolean values ($\mathbb{B}$), returns its pointwise negation: for any time t, $\overset{\mathbb{T}}{\sim} P(t) = \sim P(t)$.

This concept is being defined generally, rather than just applying it whenever we need, because we can easily prove that lifting preserves certain properties of the original operator such as commutativity, associativity, idempotency, etc. This allows us to assume these properties whenever we lift an operator without having to make sure that they hold each and every time.

3.2.3 The Semantics

Now that the syntax has been stated, we may start to give a semantic meaning to the symbols used. So as to avoid having to declare the type of every variable used, we will use the following convention: n is a number, X and Y are state variables, P and Q state expressions and D, E and F duration formulae. Given an interpretation $\mathcal{I}$ of the state variables, we can now define the semantics of DC.

State Variables

State variables are the basic building blocks of duration formulae. State variables describe whether a state holds at individual points in time. We will use the non-negative real numbers to represent time.

$$\mathbb{T} == \mathbb{R}^+ \cup \{0\}$$

Thus, an interpretation of a state variable will be a function from non-negative real numbers to the boolean values 1 and 0. For any state variable X, its interpretation in $\mathcal{I}$, written as $\mathcal{I}(X)$, will have the following type:

$$\mathcal{I}(X) :: \mathbb{T} \longrightarrow \{0, 1\}$$

The semantics of a state variable under an interpretation $\mathcal{I}$ is thus easy to define:

$$[\![X]\!]_{\mathcal{I}} \stackrel{def}{=} \mathcal{I}(X)$$

Any interpretation should map the state variables **1** and **0** to the expected constant functions:

$$\begin{aligned} \mathcal{I}(\mathbf{1}) &= \lambda t : \mathbb{T} \cdot 1 \\ \mathcal{I}(\mathbf{0}) &= \lambda t : \mathbb{T} \cdot 0 \end{aligned}$$

State Expressions

State expressions have state variables as the basic building blocks. These are then constructed together using symbols normally used for propositional or predicate calculus. We interpret the operators as usually used in propositional calculus but lifted over time.

$$
\begin{array}{rcl}
[\![X]\!]_{\mathcal{I}} & \stackrel{def}{=} & \mathcal{I}(X) \\
[\![P \wedge Q]\!]_{\mathcal{I}} & \stackrel{def}{=} & [\![P]\!]_{\mathcal{I}} \stackrel{\mathbb{T}}{\cap} [\![Q]\!]_{\mathcal{I}} \\
[\![P \vee Q]\!]_{\mathcal{I}} & \stackrel{def}{=} & [\![P]\!]_{\mathcal{I}} \stackrel{\mathbb{T}}{\cup} [\![Q]\!]_{\mathcal{I}} \\
[\![P \Rightarrow Q]\!]_{\mathcal{I}} & \stackrel{def}{=} & [\![P]\!]_{\mathcal{I}} \stackrel{\mathbb{T}}{\rightarrow} [\![Q]\!]_{\mathcal{I}} \\
[\![P \Leftrightarrow Q]\!]_{\mathcal{I}} & \stackrel{def}{=} & [\![P]\!]_{\mathcal{I}} \stackrel{\mathbb{T}}{\leftrightarrow} [\![Q]\!]_{\mathcal{I}} \\
[\![\neg P]\!]_{\mathcal{I}} & \stackrel{def}{=} & \stackrel{\mathbb{T}}{\sim} [\![P]\!]_{\mathcal{I}}
\end{array}
$$

where the operators used are the lifting of the normal propositional calculus operators over time as defined earlier.

Duration Formulae

We now interpret duration formulae as functions from time intervals to boolean values. We will thus be introducing our ability to discuss properties over time intervals rather just than at single points of time. We will only be refering to closed time intervals usually written in mathematics as $[b, e]$, where both b and e are real numbers. This is formally defined as follows:

$$
\begin{array}{rcl}
[\cdot, \cdot] & :: & \mathbb{R} \times \mathbb{R} \to \mathbb{P}\mathbb{R} \\
[b, e] & \stackrel{def}{=} & \{r : \mathbb{R} \mid b \leq r \leq e\}
\end{array}
$$

Interval is the set of all such intervals such that b is at most e:

$$
Interval \stackrel{def}{=} \{[b, e] \mid b, e \in \mathbb{R},\ b \leq e\}
$$

We can now specify the type of duration formulae interpretation:

$$
[\![D]\!]_{\mathcal{I}} :: Interval \longrightarrow \{0, 1\}
$$

$\int P = n$ holds if and only if P holds exactly for n units over the given interval. More formally:

$$
[\![\textstyle\int P = n]\!]_{\mathcal{I}}[b, e] \stackrel{def}{=} \int_b^e [\![P]\!]_{\mathcal{I}} dt = n
$$

$\lceil P \rceil$ is true if the state expression P was true over the whole interval being considered, which must be longer than zero:

$$
[\![\lceil P \rceil]\!]_{\mathcal{I}}[b, e] \stackrel{def}{=} (\int_b^e [\![P]\!]_{\mathcal{I}} dt = e - b) \text{ and } (b < e)
$$

Now we come to the operators $\wedge$, $\vee$, $\neg$, $\Rightarrow$ and $\Leftrightarrow$. Note that these are *not* the same operators as defined earlier for state expressions. Their inputs and outputs are of different types from the previous ones. However, they will be defined in a very similar way:

$$
\begin{array}{rcl}
[\![D \wedge E]\!]_{\mathcal{I}} & \stackrel{def}{=} & [\![D]\!]_{\mathcal{I}} \overset{Interval}{\cap} [\![E]\!]_{\mathcal{I}} \\
[\![D \vee E]\!]_{\mathcal{I}} & \stackrel{def}{=} & [\![D]\!]_{\mathcal{I}} \overset{Interval}{\cup} [\![E]\!]_{\mathcal{I}} \\
[\![D \Rightarrow E]\!]_{\mathcal{I}} & \stackrel{def}{=} & [\![D]\!]_{\mathcal{I}} \overset{Interval}{\rightarrow} [\![E]\!]_{\mathcal{I}} \\
[\![D \Leftrightarrow E]\!]_{\mathcal{I}} & \stackrel{def}{=} & [\![D]\!]_{\mathcal{I}} \overset{Interval}{\leftrightarrow} [\![E]\!]_{\mathcal{I}} \\
[\![\neg D]\!]_{\mathcal{I}} & \stackrel{def}{=} & \overset{Interval}{\sim} [\![D]\!]_{\mathcal{I}}
\end{array}
$$

These definitions may be given in a style which is simpler to understand by defining the application of the function pointwise as follows:

$$
[\![D \wedge E]\!]_{\mathcal{I}}[b,e] \quad \stackrel{def}{=} \quad [\![D]\!]_{\mathcal{I}}[b,e] \cap [\![E]\!]_{\mathcal{I}}[b,e]
$$

We now come to the *chop* operator $;$, sometimes referred to as *join* or *sequential composition*. By $D \,;\, E$ we will informally mean that the current interval can be chopped into two parts (each of which is a closed, continuous interval) such that D holds on the first part and E holds on the second. Formally, we write:

$$
[\![D \,;\, E]\!]_{\mathcal{I}}[b,e] \stackrel{def}{=} \exists m \in [b,e] \cdot [\![D]\!]_{\mathcal{I}}[b,m] \cap [\![E]\!]_{\mathcal{I}}[m,e]
$$

Existential Quantification

Standard DC allows only quantification over global variables whereas in Interval Temporal Logic the quantifier can be used over both global variables and state variables. We adopt state variable quantification since it will be found useful later.

Existentially quantifying over a state variable is defined simply as existentially quantifying over the function:

$[\![\, \exists X \cdot D]\!]_{\mathcal{I}}$ is true if there exists an alternative interpretation $\mathcal{I}'$ such that:

$$
\begin{array}{rcl}
\mathcal{I}'(Y) & = & \mathcal{I}(Y) \text{ if } Y \neq X \\
\text{and } [\![D]\!]_{\mathcal{I}'} & = & \text{true}
\end{array}
$$

3.2.4 Validity

A duration formula is said to be valid with respect to an interpretation $\mathcal{I}$ if it holds for any prefix interval of the form $[0,n]$. This is written as $\mathcal{I} \vdash D$.

A formula is simply said to be *valid* if it is valid with respect to any interpretation $\mathcal{I}$. This is written as $\vdash D$.

For example, we can prove that for any interpretation, the chop operator is associative. This may be written as:

$$
\vdash (D \,;\, (E \,;\, F)) \Leftrightarrow ((D \,;\, E) \,;\, F)
$$

3.2.5 Syntactic Sugaring

At this stage, anybody familiar with other temporal logics may wonder when certain features and operators common to these logics will be presented. Operators such as *always* and *sometimes* can, in fact, be defined as syntactic equivalences to operators already described. This section will deal with the definition of some of these useful operators:

- l, read as *length* gives the length of the interval in question. It may be defined simply as the integral of the function $\mathbf{1}$ (which always returns true) over the current interval:

  $$l \stackrel{def}{=} \int \mathbf{1}$$

  Note that, since we have only defined the meaning $\int P = n$, we can only use the length operator to make sure that the interval is in fact exactly n units long, that is, in the form $l = n$. Later we will define constructs to allow us to state that the length of the interval is at least, or at most, n.

- $\lceil\rceil$, read as *empty*, states that the interval under consideration is empty. Recall that $\lceil P \rceil$ is always false for an empty interval. Also note that $\lceil \mathbf{1} \rceil$ is always true for non-empty intervals.

  $$\lceil\rceil \stackrel{def}{=} \neg\lceil \mathbf{1} \rceil$$

- We can now define a construct to make sure that a state expression is true throughout the construct. Unlike $\lceil P \rceil$, however, we will consider an empty interval to satisfy this constraint.

  $$\lfloor P \rfloor \stackrel{def}{=} \lceil\rceil \vee \lceil P \rceil$$

- As in the case of boolean states, we would like a duration formula which is always true. This will be aptly named **true**:

  $$\mathbf{true} \stackrel{def}{=} \lceil\rceil \vee \lceil \mathbf{1} \rceil$$

  false may now be defined simply as a negation of **true**:

  $$\mathbf{false} \stackrel{def}{=} \neg\mathbf{true}$$

- For any duration formula D, $\Diamond D$, read as *sometimes* D, holds if and only if D holds over some sub-interval of the one being currently considered. $\Diamond D$ is itself a duration formula. The formal definition is:

  $$\Diamond D \stackrel{def}{=} \mathbf{true} \;;\; D \;;\; \mathbf{true}$$

- We define $\Box$ as the dual of *sometimes*. $\Box D$, read as *always* D, holds whenever D holds for any subinterval of the current one. Alternatively, it may be seen as: there is no subinterval over which $\neg D$ holds. The definition is given in terms of *sometimes*:

  $$\Box D \stackrel{def}{=} \neg(\Diamond(\neg D))$$

  From this definition we can immediately infer that:

  $$\Diamond D = \neg(\Box(\neg D))$$

- Similar to $\Diamond D$ we can define $\Diamond_i D$, read as *initially, sometimes* D, which holds if there is a prefix of the current interval over which D holds:

  $$\Diamond_i D \stackrel{def}{=} D \;;\; \mathbf{true}$$

- As before, we can define $\Box_i D$, read as *initially, always D*, as follows:

$$\Box_i D \stackrel{def}{=} \neg\Diamond_i\neg D$$

- Recall that we only defined $\int P = n$ as a duration formula. It is usually the case that we desire to check conditions such as $\int P \leq n$ or $\int P \geq n$. These may be defined as:

$$\begin{array}{lcl} \int P \geq n & \stackrel{def}{=} & \int P = n \ ; \mathbf{true} \\ \int P \leq n & \stackrel{def}{=} & \neg(\int P \geq n) \vee (\int P = n) \end{array}$$

  Note that now the expressions $l \geq n$ and $l \leq n$ are duration formulae.

3.3 Least Fixed Point

In expressing certain situations, it may be useful to be able to recursively define temporal behaviour. In [PR95] P.K. Pandya and Y.S. Ramakrishna introduce the use of the μ notation in the Mean-Value Calculus, a variant of Duration Calculus. This section presents part of this work with respect to the Duration Calculus. The resulting calculus allows for finitely many free variables, and enables us to use finite mutual recursion.

3.3.1 Open Formulae

A duration formula is said to be open if contains a positive number of free variables. One such formula is $l < 5 \vee (l = 5; X)$. We will write such formulae as $F(\overline{X})$, where $\overline{X}$ is the list of free variables in the formulae.

Given an interpretation $\mathcal{J}$ of open variables, we can also give the meaning of open formulae:

$$[\![X]\!]_{\mathcal{I}}^{\mathcal{J}} \stackrel{def}{=} \mathcal{J}(X)$$

For any other duration formula D, the extended semantics act just as before:

$$[\![D]\!]_{\mathcal{I}}^{\mathcal{J}} \stackrel{def}{=} [\![D]\!]_{\mathcal{I}}$$

3.3.2 Fixed Points

Now consider the recursive equation $X = F(X)$. An interpretation j of variable X is a solution (for a particular $\mathcal{I}$ and $\mathcal{J}$) if:

$$j \ = \ [\![F(X)]\!]_{\mathcal{I}}^{\mathcal{J}[X \leftarrow j]}$$

where $\mathcal{J}[X \leftarrow j]$ represents the interpretation identical to $\mathcal{J}$ except that X is interpreted as j. The formula given does not necessarily have exactly one solution. Of these, we will be interested in the least fixed point, which we denote by $\mu X \cdot F(X)$.

The concept of *least* can be expressed informally as 'the interpretation which assigns 1 the least possible number of times'. More formally, we say that an interpretation (of a single variable) i is at least as large as another interpretation j (which we will write as $i \leq j$), if any interval for which i returns 1, so does j:

$$i \leq j \stackrel{def}{=} i \stackrel{Interval}{\rightarrow} j$$

.

3.3.3 Monotonicity

We say that an interpretation $\mathcal{J}$ is not larger than $\mathcal{J}'$, with respect to a variable X if:

- $\mathcal{J}(Y) = \mathcal{J}'(Y)$ for any $Y \neq X$, and
- $\mathcal{J}(X) \leq \mathcal{J}'(X)$

We write this as $\mathcal{J} \leq_X \mathcal{J}'$.

A formula F is said to be monotonic if:

$$\mathcal{J} \leq_X \mathcal{J}' \Rightarrow [\![F]\!]_{\mathcal{I}}^{\mathcal{J}} \leq [\![F]\!]_{\mathcal{I}}^{\mathcal{J}'}$$

3.3.4 Semantics of Recursion

[PR95] goes on to prove a number of useful results. The ones which are of interest here are:

- If, in a formula F, X always appears within the scope of an even number of negations, F is monotonic.
- If F is monotonic in X, then the semantics of $\mu X \cdot F$ are given by:

  $$\bigcap^{Interval} \{i \mid [\![F]\!]_{\mathcal{I}}^{\mathcal{J}[X \leftarrow i]} \leq i\}$$

  In other words, it is true for an interval if and only if *all* interpretations in V are true over that interval.
- The least fixed point satisfies the following laws:

  $$\begin{array}{rcl} \mu X \cdot F(X) & = & F(\mu X \cdot F(X)) \\ F(D) \Rightarrow D & \vdash & \mu X \cdot F(X) \Rightarrow D \end{array}$$

This construct can, for example, be used to specify a clock signal with period $2n$ in such a way as to expose its iterative nature:

$$\begin{array}{rcl} CLOCK_1 & = & (l < n \wedge \lfloor C \rfloor) \ \vee \ (l = n \wedge \lceil C \rceil); CLOCK_0 \\ CLOCK_0 & = & (l < n \wedge \lfloor \neg C \rfloor) \ \vee \ (l = n \wedge \lceil \neg C \rceil); CLOCK_1 \end{array}$$

These can be combined into:

$$CLOCK \quad = \quad \mu X \cdot \left(\begin{array}{ll} & (l < n \wedge \lfloor C \rfloor) \\ \vee & (l = n \wedge \lceil C \rceil); (l < n \wedge \lfloor \neg C \rfloor) \\ \vee & (l = n \wedge \lceil C \rceil); (l = n \wedge \lceil \neg C \rceil); X \end{array} \right)$$

3.4 Discrete Duration Calculus

In [HO94] Jifeng He and Ernst-Rüdiger Olderog reduce the general duration calculus to a discrete time domain. This serves to simplify the calculus whenever continuous time is not required, such as in the description of clocked circuits. The modifications are now described.

The calculus will stand unchanged except for the following assumptions:

1. The state functions may only change values at integer times. In other words, any state function has to be constant over open intervals of the form $(n, n+1)$ where n is a natural number.

2. Only intervals with integer start and end points are allowed.

3. Because of the previous condition, validity is now reduced to prefix intervals of the form $[0, n]$ where n is a natural number.

It will also be useful to introduce a unit interval operator similar to $\lceil P \rceil$ but also implying that we are considering a unit length interval:

$$\lceil\lceil P \rceil\rceil \stackrel{def}{=} \lceil P \rceil \wedge l = 1$$

Since the boolean states are now discrete, rather than continuous, it now makes more sense to introduce a past, or history operator, which uses past values of the state variables or expressions. For a state expression P and natural number n, $n \gg P$, read as P *shifted by* n, is defined as P but with time shifted n units back. Since P is a function defined on the non-negative real numbers, $n \gg P$ has to be arbitrarily defined on the first n time units. As a convention, we choose $n \gg P$ to be false for this period. Time shift may be formally defined as:

$$(n \gg P)\; t = \begin{cases} 0 & \text{if } t < n \\ P(t-n) & \text{otherwise} \end{cases}$$

Specifying the clock signal is now possible in a different style, emphasising better the alternating value of the clock:

$$CLOCK \quad = \quad \Box(l = 1 \Rightarrow \lceil\lceil n \gg C = \neg C \rceil\rceil)$$

3.5 Relational Duration Calculus

3.5.1 Introduction

The Problem

As has been shown elsewhere, duration calculus (DC) and other temporal logics are very useful in describing systems where timing is inherent. When describing a system using DC we can usually take safe estimates about timings. However, certain systems we would like to describe using temporal logics include a non-timed subset which is very difficult to describe using a standard temporal logic. Problems usually arise when we would like to describe strings of consecutive zero time transitions.

When defining the semantics of most hardware description languages in terms of some temporal logic, this is one of the main obstacles which have to be overcome. Most HDLs (such as Verilog HDL and VHDL) have both timed and zero delay transitions (assignments). The non-zero delay transitions are quite readily described using any temporal logic. However, the zero delay assignments prove to be a problem since we expect these zero length transitions to happen consecutively.

Zero delays are best described as infinitesimal delays. We may thus fit any finite number of 'zero delays' into any finite interval. Momentary intermediate values taken by variables during these infinitesimal transitions are ignored by the model as a whole, but may still affect other variables.

For example, in the series of assignments: `i:=1; j:=i; i:=0`, the temporary value 1 of variable `i` will not be captured by the complete system. Still, its side-effects should be visible, and we expect `j` to be set to 1 after the series of assignments.

The Solution

The approach used here is to define an extension of DC, where these zero time transitions are treated in a relational way to capture temporary and carried over values. Hence the name *relational duration calculus*.

Despite the problems faced by pure DC, most specifications are only interested in the stable states of the system. This means that the specification of such a system can usually be given in pure DC. An implementation may, however, require the use of the relational extension presented here. The embedding of DC within our system would thus mean that, in effect, such pure DC specifications can be directly verified.

The problem of where to introduce this necessary effect now arises. The solution presented here introduces a new chop operator. The standard chop operator (;) presented earlier causes problems when dealing with zero delay transitions. One example where the result is not what we expect in this new domain of application is:

$$(\lceil\rceil \wedge A) \,;\, (\lceil\rceil \wedge B) \equiv (\lceil\rceil \wedge A \wedge B) \equiv (\lceil\rceil \wedge B) \,;\, (\lceil\rceil \wedge A)$$

The whole idea of sequence is thus lost.

3.5.2 The Syntax and Informal Semantics

Relational duration calculus is built over discrete duration calculus by adding a number of new operators to handle immediate changes over state variables.

Pre and Post Values of State Variables

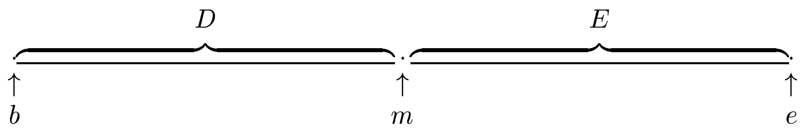
We will need a way of examining the value of a state variable just before the current interval begins. This is, in effect, similar to reading an input. $\overleftarrow{v}$ will stand for this value.

Similarly we will need to examine or set values of state variables at the end of the current interval. Again, this has a similar connotation to the output of a process. $\overrightarrow{v}$ is the end value of state variable v.

As a simple example, we may now specify the real meaning of an assignment `v := w` as follows: $\lceil\rceil \wedge \overrightarrow{\mathtt{v}} = \overleftarrow{\mathtt{w}}$.

A New Chop Operator

As already discussed, we will need an alternative chop operator. Given two relational DC formulae D and E, we can informally describe the meaning of D followed by E where the post-values of the state variables var in D match their pre-value in E. This is written as $D \overset{var}{\fatsemi} E$ and is informally described as follows:



$D \overset{var}{\fatsemi} E$ holds over an interval $[b, e]$ if we are able to split the interval into two parts at a point m such that:

- D holds over $[b, m]$ except for variables overwritten by E at m
- E holds over $[m, e]$ with the inputs being the outputs of D

A more formal definition will be given later.

The Syntax of Relational Duration Calculus

Now that an informal description of the new operators has been given, we may give a complete description of the relational duration calculus syntax. This is given in table 3.2.

We will refer to the set of all duration formulae in the Relational Duration Calculus as $\mathcal{RDF}$.

3.5.3 The Semantics

As the syntax table shows, relational duration calculus has a large subset which is simply duration calculus. The semantics of that subset are identical to those given for the duration calculus in section 3.2.3 and are not redefined here.

Input and Output Values

The input value of a variable v over an interval $[b, e]$, denoted by $\overleftarrow{v}$ is the value of v just before time b. If b is zero, it will automatically be false. This may be defined as:

$$(\overleftarrow{v} = x)[b, e] \overset{def}{=} \begin{cases} \textbf{false} & \text{if } b = 0 \\ \lim\limits_{t \to b^-} v(t) \text{ is equal to } x & \text{otherwise} \end{cases}$$

$\overrightarrow{v}$ is the 'dual' of $\overleftarrow{v}$. It is defined as the right limit of v at the end point of the interval:

$$(\overrightarrow{v} = x)[b, e] \overset{def}{=} \lim_{t \to e^+} v(t) \text{ is equal to } x$$

We may now define equality between two edge values. If e_1 and e_2 are edge values (that is, of the form $\overleftarrow{v}$ or $\overrightarrow{v}$):

$$e_1 = e_2 \overset{def}{=} \exists x : \mathbb{B} \cdot (e_1 = x \wedge e_2 = x)$$

$\langle$boolean$\rangle$		
$\langle$natural number$\rangle$		
$\langle$state variables$\rangle$		
$\langle$state expression$\rangle$	::=	$\langle$state variables $\rangle$
	|	$\langle$state expression$\rangle \wedge \langle$state expression$\rangle$
	|	$\langle$state expression$\rangle \vee \langle$state expression$\rangle$
	|	$\langle$state expression$\rangle \Rightarrow \langle$state expression$\rangle$
	|	$\langle$state expression$\rangle \Leftrightarrow \langle$state expression$\rangle$
	|	$\neg\langle$state expression$\rangle$
$\langle$edge value$\rangle$	::=	$\overleftarrow{\langle\text{state variable}\rangle}$
	|	$\overrightarrow{\langle\text{state variable}\rangle}$
$\langle$duration formula$\rangle$	::=	$\langle$edge value$\rangle = \langle$boolean$\rangle$
	|	$\int\langle$state expression$\rangle = \langle$natural number$\rangle$
	|	$\lceil\langle$state expression$\rangle\rceil$
	|	$\langle$duration formula$\rangle \wedge \langle$duration formula$\rangle$
	|	$\langle$duration formula$\rangle \vee \langle$duration formula$\rangle$
	|	$\langle$duration formula$\rangle \Rightarrow \langle$duration formula$\rangle$
	|	$\langle$duration formula$\rangle \Leftrightarrow \langle$duration formula$\rangle$
	|	$\neg\langle$duration formula$\rangle$
	|	$\langle$duration formula$\rangle \overset{var}{\fatsemi} \langle$duration formula$\rangle$
	|	$\langle$duration formula$\rangle$ **;** $\langle$duration formula$\rangle$
	|	$\exists\langle$state variable$\rangle \cdot \langle$duration formula$\rangle$

Table 3.2: Syntax of the Relational Duration Calculus

Relational Chop

The relational chop operator will be defined in stages. To satisfy $D \overset{var}{\fatsemi} E$, we must first find a midpoint m at which to split the interval. We must then rename all variables var in D to fresh variables. Obviously, the inputs of the renamed variables are the same as those of the original ones. Also, we need to store the final value of the variables at the end of D since they may be used in E:

$$(D[var/var_D] \wedge \bigwedge_{v \in var} (\overleftarrow{v}_D = \overleftarrow{v} \wedge v' = \overrightarrow{v}_D))[b, m]$$

Similarly, variables of E will be renamed to fresh variables. We must also make sure that the input of E is the same as the output of D:

$$(E[var/var_E] \wedge \bigwedge_{v \in var} \overleftarrow{v}_E = v')[m, e]$$

Finally, we must define the values of the actual variables. For every variable v in var:

$$\begin{aligned}\bigwedge_{v \in var} \quad & \lfloor v \Leftrightarrow v_D \rfloor [b, m] \text{ and} \\ & \lfloor v \Leftrightarrow v_E \rfloor [m, e] \text{ and} \\ & \overrightarrow{v} = \overrightarrow{v_E}[m, e] \text{ and} \\ & \overleftarrow{v} = \overleftarrow{v}_D[b, m]\end{aligned}$$

The complete definition of the relational chop operator may now be given as the conjunction of the above expressions. If the set of variables var is $\{v_1, v_2, \ldots v_n\}$:

$$\begin{aligned}D \overset{var}{\fatsemi} E \overset{def}{=} \quad & \exists \quad m : \mathbb{N}, var' : \mathbb{B}^* \cdot \\ & \exists \quad var_D, var_E \cdot \\ & \qquad (Exp_1 \wedge Exp_2 \wedge Exp_3)\end{aligned}$$

where Exp_1, Exp_2 and Exp_3 are the three above expressions. Moving around the existential quantification, we can define the relational chop in terms of the normal chop operator:

$$\begin{aligned}& D \overset{var}{\fatsemi} E \overset{def}{=} \\ & \quad \exists var' : \mathbb{B}^* \cdot \ \exists var_D, var_E \cdot \\ & \quad\ (D[var/var_D] \wedge \bigwedge_{v \in var} (\overleftarrow{v_D} = \overleftarrow{v} \wedge \lfloor v \Leftrightarrow v_D \rfloor \wedge v' = \overrightarrow{v_D}))\ ; \\ & \quad\ (E[var/var_E] \wedge \bigwedge_{v \in var} (\overleftarrow{v_E} = v' \wedge \lfloor v \Leftrightarrow v_E \rfloor \wedge \overrightarrow{v} = \overrightarrow{v_E}))\end{aligned}$$

Note that we are hereby defining a family of chop operators — one for every set of state variables var. Note that when var is chosen to be the empty set, the relational chop reverts back to the normal DC chop operator.

To go back to the original example of showing that the visible behaviour of `v:=1; w:=v; v:=0` is the same as that of `v,w:=0,1`, we can use this new chop construct to deduce that:

$$\begin{pmatrix} \lceil\rceil & \wedge & \overrightarrow{v} = 1 \\ & \wedge & \overrightarrow{w} = \overleftarrow{w} \end{pmatrix} \overset{\{v,w\}}{\fatsemi} \begin{pmatrix} \lceil\rceil & \wedge & \overrightarrow{w} = \overleftarrow{v} \\ & \wedge & \overrightarrow{v} = \overleftarrow{v} \end{pmatrix} \overset{\{v,w\}}{\fatsemi} \begin{pmatrix} \lceil\rceil & \wedge & \overrightarrow{v} = 0 \\ & \wedge & \overrightarrow{w} = \overleftarrow{w} \end{pmatrix}$$
$$\equiv \begin{pmatrix} \lceil\rceil & \wedge & \overrightarrow{v} = 0 \\ & \wedge & \overrightarrow{w} = 1 \end{pmatrix}$$

3.6 Laws of the Relational Chop Operator

The relational chop operator, together with the pre- and post-value operators, act very much like assignment and sequential composition. A number of laws they obey are, in fact, very similar to the laws of assignment given in [H+85]. Some such laws include:

$$\begin{array}{lcll} (\lceil\rceil \wedge \overrightarrow{v} = \alpha) \overset{W}{\fatsemi} (\lceil\rceil \wedge \overrightarrow{v} = \beta) & = & (\lceil\rceil \wedge \overrightarrow{v} = \beta) & \text{provided that } v \in W \\ (\lceil\rceil \wedge \overrightarrow{v} = \alpha) \overset{W}{\fatsemi} (\lceil\rceil \wedge \overrightarrow{w} = f(v)) & = & (\lceil\rceil \wedge \overrightarrow{w} = f(\alpha)) & \text{provided that } w \in W \end{array}$$

where α and β are constants. Note that in the second law, the intermediate value of v was lost since it was not propagated in the second formula. These and other similar laws can be deduced from the following laws:

$$\begin{array}{rcl} (D \wedge \overrightarrow{v} = \alpha) \overset{W}{\fatsemi} E(\overleftarrow{v}) & = & D \,;\, E(\alpha) \\ D \overset{W}{\fatsemi} E(\overleftarrow{v}) & = & \exists \alpha \cdot (D \,;\, E(\alpha)) \end{array}$$

where both laws hold provided that:

- $v \in W$,
- if $\overrightarrow{w}$ appears in D, then $w \notin W$ and
- if $\overleftarrow{w}$ appears in E and w is not v, then $w \notin W$

Both laws are direct consequences of the definition of relational chop. Using induction, these laws can be generalised to:

$$\begin{array}{rl} & (D \wedge \overrightarrow{v}_1 = \alpha_1 \wedge \ldots \wedge \overrightarrow{v}_n = \alpha_n) \overset{W}{\fatsemi} E(\overleftarrow{v}_1, \ldots, \overleftarrow{v}_n, \ldots, \overleftarrow{v}_{n+k})) \\ = & \exists \alpha_{n+1}, \ldots \alpha_{n+k} \cdot (D \,;\, E(\alpha_1, \ldots, \alpha_{n+k}) \end{array}$$

provided that:

- for all i $(1 \leq i \leq n+k)$, $v_i \in W$,
- if $\overrightarrow{w}$ appears in D, then $w \notin W$ and
- if $\overleftarrow{w}$ appears in E and w is not one of the v_i's, then $w \notin W$

3.7 Conclusions

Another temporal logic similar to Duration Calculus is Interval Temporal Logic [Mos85]. The main difference between the two is the time domain — Duration Calculus has a dense time domain as opposed to the discrete time domain used in Interval Temporal Logic. Since asynchronous systems can be expressed more naturally using a continuous time domain, Duration Calculus (together with its discrete variant) allows immediate comparison of, for instance, the interaction between a clocked circuit and a continuous specification of its environment. On the other hand, Interval Temporal Logic allows quantification over state variables which can be convenient to in temporal abstraction.

Alternative approaches to solve the problem of sequentiality of zero time intervals could have been taken. In [ZH96a], Zhou Chaochen and Michael Hansen propose a similar extension to Duration Calculus to handle immediate assignments. The method they use defines the relational chop operator by creating a small interval between the two over which the two duration formulae can 'communicate'. This extra interval is then abstracted over.

[Qiw96] proposes another similar approach which uses a different interpretation for state functions. The state functions associate two values for each state variable at each and every time point which are then used in a very similar fashion to the pre- and post-values described here. The result is used to derive an extension to Hoare logic for reasoning about sequential programs which may use DC formulae for time consuming (continuous) sections. The model is not used to handle parallel processes and the elegance of the proofs would probably be somewhat reduced for such cases.

The main difference between the calculi presented in the above two papers [Qiw96, ZH96a] and Relational DC as presented here is the fact that our chop operator has a related type. In the other work, the chop operator unifies all the state variables. In other words, whereas we define a family of chop operators: $\overset{W}{\fatsemi}$, they have only one chop operator behaving like $\overset{var}{\fatsemi}$, where var is the set of all the state variables in the system. This does not reduce expressivity of the other calculi, but it can make certain properties more difficult to state. For example, in our case it is quite straightforward to specify a number of duration formulae, each of which controls a number of state variables and which may interact (by, for instance, the assignment of a variable to the value of another variable controlled by a different formula). These two specifications can usually easily be stated using Relational DC, such that their conjunction corresponds to the behaviour of the composition of the threads obeyed at the same time.

Consider, for example, the statement that w is assigned to 1 after which it is assigned to the value of $w \wedge v$:

$$(\lceil\rceil \wedge \overrightarrow{w} = 1) \overset{\{w\}}{\fatsemi} (\lceil\rceil \wedge \overrightarrow{w} = (\overleftarrow{w} \wedge \overleftarrow{v}))$$

In relational DC this reduces to: $\overrightarrow{w} = \overleftarrow{v}$.

However, if we can use only a universal relational chop ($\overset{\{v,w\}}{\fatsemi}$ in this case), the above statement changes its meaning to: $\exists b : \mathbb{B} \cdot \overrightarrow{w} = b$ (which is equivalent to **true**).

As already stated, this does not mean that the other calculi are less expressive. There still are ways to state this same property. Obviously, we would rather not place any constraints on v in this formula (since we might want to give a separate duration formula describing the behaviour of v). So what can be done if we want to describe multiple variables whose behaviour is defined by different conjuncts? The solution proposed in [ZH96a] uses traces of the shared variables. As expected, however, this introduces an extra level of complexity to the specification, which Relational DC manages without. It is for this reason that we believe that for the kind of problems treated in this thesis, Relational DC is a better candidate than the other similar DC variants.

[BB93] shows how infinitesimal time durations can be included within a temporal process algebra. It would be an interesting project to attempt to integrate the ideas presented in this paper into duration calculus.

Another possible approach investigated elsewhere is the use of timing diagrams [SD97] where the semantics of such an instantaneous change are defined over the timing diagrams of the function.

One method not explicitly investigated elsewhere is the use of a state based discrete temporal logic where time is simply considered to be another (non-decreasing) variable. One would then define an abstraction over the steps which do not increase the value of time to derive a direct relationship between the sequence of states and time steps.

To conclude, the relational duration calculus described in this chapter is one possible way of defining the semantics of languages which combine both timed and immediate constructs which seems to work reasonably well. The complexity of the definitions may be initially off-putting. However, as in the case of duration calculus, these definitions should ideally be used to verify a number of algebraic laws, which can then be used in larger proofs. This way, the ability to simply specify certain systems in Relational DC is not spoilt by the complex underlying semantics.

Chapter 4

Real Time Specifications

This short chapter introduces a number of standard forms for the duration calculus which can be used to simplify and standardise specifications.

These standard forms help to describe certain commonly encountered real-time properties. A number of laws will accompany the standard forms so as to reduce (or altogether remove) the need to revert back to duration calculus to prove results.

These operators were originally introduced in [Rav95, MR93] where the laws we state are also proved.

To illustrate the use of the operators, we will state properties of a simple gas-burner system with three boolean states. *Gas* will be boolean state true when gas is being released in the system, *Flame* is true if and only if the gas is ignited, and finally *Alarm* will represent the state of an alarm, which is to be turned on if too much unburnt gas is released into the system.

4.1 The Real Time Operators

4.1.1 Invariant

A boolean expression is said to be an *invariant* of the system if it is true at any time. This will be written as $\surd P$ and defined by:

$$\surd P \stackrel{def}{=} \lfloor P \rfloor$$

In our example system, one would expect that there is never a flame without gas: $\surd(\neg Gas \Rightarrow \neg Flame)$

Laws of *Invariant*

Invariants are monotonic on their only operand:

Law 1: Monotonic
If $P \Rightarrow Q$ and $\surd P$ then $\surd Q$.

Conjunction distributes in and out of invariants:

Law 2: Conjunctive
$\surd P \wedge \surd Q = \surd(P \wedge Q)$

On the other hand, disjunction distributes in, but not out, of invariants:

Law 3: Disjunctive
If $\surd P \vee \surd Q$ then $\surd(P \vee Q)$.

Formulae of the form $\surd(P \Rightarrow Q)$, allow us to replace P by Q in monotonic contexts:

Law 4: Replacement
If $\surd(P \Rightarrow Q)$ and $F(P)$, where F is monotonic[1], then $F(Q)$.

4.1.2 Followed By

It is usually convenient to specify that whenever a duration formula D holds over an interval, another formula E will initially hold on non-zero intervals starting at the end of the current one.

$D \longrightarrow E$ is read as D *is always followed by* E and defined as follows:

$$D \longrightarrow E \stackrel{def}{=} \forall r : \mathbb{R} \cdot \Box(((D \wedge l = r) \,;\, l > 0) \Rightarrow (l = r \,;\, E \,;\, \mathbf{true}))$$

One may insist that the alarm is to be turned on if there is a period of more than 7 time units during which unburnt gas is being released: $((l \geq 7 \wedge \lceil Gas \wedge \neg Flame \rceil) \longrightarrow \lceil Alarm \rceil$.

Laws of *Followed By*

Conjunction outside of *followed by* expressions to a common antecedent distribute to disjunction inside the operator:

Law 1: Distributive (I)
$D_1 \longrightarrow E$ and $D_2 \longrightarrow E$ if and only if $(D_1 \vee D_2) \longrightarrow E$.

On the other hand, disjunction can be distributed inside as conjunction:

Law 2: Distributive (II)
If $D_1 \longrightarrow E$ or $D_2 \longrightarrow E$ then $(D_1 \wedge D_2) \longrightarrow E$.

4.1.3 Leads To

Extending the idea of the *followed by* operator is the *leads to* operator. This operator will be used to express progress in the system being described. $D \stackrel{\delta}{\longrightarrow} E$ will signify that after duration formula D holds for δ time units, duration formula E will then hold. It is read as D *leads to* E *after holding for* δ.

$$D \stackrel{\delta}{\longrightarrow} E \stackrel{def}{=} (D \wedge l = \delta) \longrightarrow E$$

In most cases, D and E will be formulae of the form $\lceil P \rceil$. To avoid repetition, we will overload this operator with one which takes two state expressions:

$$P \stackrel{\delta}{\longrightarrow} Q \stackrel{def}{=} \lceil P \rceil \stackrel{\delta}{\longrightarrow} \lceil Q \rceil$$

The statement previously given about when the alarm should be turned off can now be rephrased as: $Gas \wedge \neg Flame \stackrel{7}{\longrightarrow} Alarm$

[1] F is said to be monotonic if $P \Rightarrow Q$ implies that $F(P) \Rightarrow F(Q)$

Laws of ***Leads To***

The following laws pertain to the *leads to* operator with state expressions as operands.

The *leads to* operator is monotonic on all three of its operands:

Law 1: Monotonic on delay
If $P \xrightarrow{\delta} Q$ and $\Delta \geq \delta$ then $P \xrightarrow{\Delta} Q$.

Law 2: Monotonic on first operand
If $P \xrightarrow{\delta} Q$ and $R \Rightarrow P$ then $R \xrightarrow{\delta} Q$.

Law 3: Monotonic on second operand
If $P \xrightarrow{\delta} Q$ and $Q \Rightarrow R$ then $P \xrightarrow{\delta} R$.

Conjunction distributes in and out of the second state expression:

Law 4: Conjunctive
$P \xrightarrow{\delta} Q_1$ and $P \xrightarrow{\delta} Q_2$ if and only if $P \xrightarrow{\delta} (Q_1 \land Q_2)$.

Leads to is transitive:

Law 5: Transitive
If $P \xrightarrow{\delta_1} Q$ and $Q \xrightarrow{\delta_2} R$ then $P \xrightarrow{\delta_1+\delta_2} R$.

4.1.4 Unless

Sometimes we want to specify that a state expression, if true, must remain so unless another state expression is true. If P and Q are state expressions, then the formula P *unless* Q is defined by:

$$P \ unless \ Q \stackrel{def}{=} \Box(\lceil P \rceil \ ; \lceil \neg P \rceil \Rightarrow \lceil P \rceil \ ; \lceil \neg P \land Q \rceil \ ; \mathbf{true})$$

The alarm must not turn off until the gas leak is fixed: $Alarm$ *unless* $\neg(Gas \land \neg Flame)$.

So as not to have the alarm turning on whenever it feels like it, we can insist that it remains off until a gas leak occurs: $\neg Alarm$ *unless* $Gas \land \neg Flame$.

Laws of ***Unless***

The *unless* operator is monotonic on its second operand:

Law 1: Monotonic
If $Q \Rightarrow R$ and P *unless* Q then P *unless* R.

Conjunction distributes into the first operand becoming disjunction:

Law 2: Distributivity
If P_1 *unless* Q and P_2 *unless* Q then $(P_1 \lor P_2)$ *unless* Q.

Conjunction distributes inside the second operand:

Law 3: Conjunctivity
If P *unless* Q_1 and P *unless* Q_2 then P *unless* $(Q_1 \land Q_2)$.

4.1.5 Stability

Certain boolean expressions must remain true for a minimum number of time units. This stability requirement is written as $\mathcal{S}(P, \delta)$ and is formally defined as:

$$\mathcal{S}(P, \delta) \stackrel{def}{=} \Box(\lceil \neg P \rceil \ ; \lceil P \rceil \ ; \lceil \neg P \rceil \Rightarrow l > \delta)$$

Finally, to ensure that the alarm does not flash on and off we insist that once it turns on it must remain so for at least 10 time units: $\mathcal{S}(Alarm, 10)$.

Laws of *Stability*

Stability is monotonic:

Law 1: Monotonic
If $\mathcal{S}(P, \Delta)$ and $\Delta \geq \delta$ then $\mathcal{S}(P, \delta)$.

4.2 Discrete Duration Calculus

Finally, we give a small number of laws which arise from these operators when we use discrete duration calculus. These laws were given in [HO94].

Law 1: Discrete leads-to
$P \stackrel{\delta}{\longrightarrow} Q = \Box(\lceil\lceil \neg Q \rceil\rceil \Rightarrow \bigvee_{0<i\leq\delta} \lceil\lceil i \gg \neg P \rceil\rceil)$

Law 2: Discrete stability (I)
$\mathcal{S}(P, \delta) = \Box(\lceil\lceil \neg P \rceil\rceil \ ; \lceil P \rceil \ ; \lceil\lceil \neg P \rceil\rceil \Rightarrow l \geq \delta + 2)$

Law 3: Discrete stability (II)
If we can find a state expression Q such that $\Box(\bigwedge_{0\leq i<n} \lceil\lceil i \gg Q \rceil\rceil \Rightarrow \lceil\lceil \neg P \rceil\rceil)$ and $\Box(\bigvee_{0\leq i<n} \neg\lceil\lceil i \gg Q \rceil\rceil \Rightarrow \lceil\lceil P \rceil\rceil)$ then $\mathcal{S}(P, n)$.

4.3 Summary

As already noted elsewhere (for example in [Man81, MP81]) and will be seen later in this thesis, using a temporal logic to model the semantics of a language eases the specification and verification of real-time requirements. For this reason, a high level real-time specification language is ideal to allow parts of the reasoning to proceed without having to fall back onto reasoning within the temporal logic. This chapter describes a number of such operators which will be used later in specifying real-time constraints and properties.

Part II

Now that all the necessary tools and notations have been exposed, we define the formal semantics of Verilog and, using these semantics, we present a number of algebraic laws which Verilog programs obey.

Chapter 5

Formal Semantics of Verilog

5.1 Introduction

5.2 Syntax and Informal Semantics of Verilog

A Verilog specification consists of a number of *modules* which run in parallel. Each module has an alphabet of local variables which it uses. In addition there are a number of global variables which act as communication ports of the complete system. Recall from chapter 2 that all the variables used will be of register type and, if not assigned a value at the current time, will keep the previous value.

We will assume that all time delays will be of integer length. If this is not so, we can always multiply all the delays by a constant, so as to satisfy this pre-condition. The original times may later be retrieved by dividing. Note that this transformation is possible since we consider only rational constant delays of which we can only have a finite number.

Each module can either (i) be a continuous assignment (`assign`), or (ii) execute the module statements endlessly (`always s`), or (iii) execute the module just once in the beginning (`initial s`).

Continuous assignments can be either immediate or delayed. Delayed continuous assignments can be either inertial or transport delayed[1].

`always` and `initial` module instructions are a sequence of statements separated by the sequential composition operator `;`. Each statement can be one of a number of types:

Skip: Unarguably the simplest instruction. Does nothing and takes no simulation time either.

Guards: These delay the execution of the statement sequence for some time. Different guard types are available. The ones we will use are:

- Time delays (`#n`): This guard blocks the statement sequence for exactly `n` time units.
- Edge sensitive delays (`@v`, `@posedge v`, `@negedge v`): These guards block the instruction sequence until an edge of the desired type is seen on variable `v`. `@posedge v` waits for a rising edge (`v` goes from 0

[1]Strictly speaking Verilog does not support transport delay. This type of delay can, however, be useful and we will see how it can be defined in terms of Verilog instructions later.

to 1), `@negedge v` for a falling edge (`v` goes from 1 to 0), and `@v` waits for either type. Note that the formal model presented here cannot handle `x` and `z` values. Section 5.5.3 discussed this in more detail.

- Value sensitive delay (`wait e`): `wait e` blocks execution until expression `e` becomes true. Note that it does not wait for a rising edge, just for the value, and hence may be enabled immediately.
- Compound guards (`g or`...`or h`): This guard blocks execution until at least one of the guards `g`...`h` is lowered. Note that this is a generalisation of Verilog compound guards which may only be of the form `@(g1 ... gn)`.

Assignments: These are the basic building blocks of the language. The language allows general concurrent assignments. Three types of assignments are identified:

- Zero-delay assignments (`v=e`): The assignment is executed without taking any simulation time at all.
- Guarded assignments (`v=g e` or `g v=e`): In the case of `v=g e`, the values of the expressions are read, after which a guard block is enacted and the assignment takes place. When the guard is lowered, the variables are assigned to the values previously read. Once the assignment takes place, the next statement (if any) following the assignment is enabled. In the case of `g v=e`, the guard is enacted and the assignments take place the moment it is lowered.
- Non-blocking assignment(`v<=g e`): This acts in a similar way as guarded assignments. However, any statements following the assignment do not wait for the assignment to terminate to be executed. In other words, sequential composition following this type of statement is not treated like sequential composition at all, but more like parallel composition.

Compound statements: The statements take one or more programs and construct more complex programs out of them.

- `if b then P else Q`: is the same as normal alternation. If `b` is true enable `P`, otherwise enable `Q`. Upon termination hand control over to the next statement, if any. The special case with no `else` clause is also allowed.
- `while b do P`: Again, this is the normal iteration construct. If `b` is currently true execute `P`, after which return control to the start of the `while` statement. A similar instruction is the `do P while b` construct. In this case, the boolean check is performed at the end of the execution of `P`, not at the beginning. A special case where `P` is executed repeatedly, without checking for any condition, can also be used: `forever P`.
- A number of instructions can also be executed concurrently using the `fork ... join` construct. Every instruction in `P` is executed concurrently, and execution ends once all individual threads end. In Verilog the individual instructions would be separated by semicolons. To avoid possible confusion, we will allow only binary parallel composition, written as `fork P || Q join`, or simply as `P || Q`. Note that we can still express parallel composition of more than two programs: if parallel composition is taken to be right associative, `fork P;Q;R join` would be written as:

`fork P` $\|$ `(fork Q` $\|$ `R join) join`

- `begin P end`: This is given just as a means of collecting statements together.

Table 5.1 gives a complete overview of the language.

Not all programs generated by this syntax will be acceptable and only the semantics of a subset will be formally defined. The restrictions placed on the language correspond to good practice hardware design and simplify the treatment of the language considerably.

The restrictions are practically all related to the concept of global and local variables. We note that a complete system is made up of a number of modules running in parallel. Each of these modules is made up of sequentially composed individual statements. We will assume that each of these modules has a set of *local* variables (internal wires) which cannot be modified or referred to by any other module.

There is also a set of *global* variables (communication registers) from which all modules may read. These may be considered to be the input and output ports of the modules. The alphabet of a module is defined to be its local variables together with the global registers it writes to.

No merge on output wires: It is considered to be good hardware design practice to avoid joining together the outputs of different components to avoid problems such as short circuits. The alphabets of different modules have thus to be pairwise disjoint. Common local variables can be renamed to avoid this, but no more than one process can write to each wire.

In sections 5.4.5 and 5.5.3 we describe how this assumption can be relaxed to enable, for instance, the specification of tri-state devices.

Registers as state holders: Global variables are *always* to be assigned using a guarded assignment with a guard which always takes time to be lowered. In other words, zero delayed assignments to output ports are not allowed. This makes sense if we look at each module as a component which must have an inherent delay and at the global state to be an abstraction of a memory register. Although it limits the language, it is usually considered good design practice to build a circuit in this fashion.

Stability of outputs of combinational circuits: When `assign` statements are used to express combinational circuits, care must be taken so as not to give rise to any combinational loop by having the output of a combinational circuit connected back into itself. Again, this approach is considered to be good design practice since it avoids problems such as race-conditions. Formalisation of this check is also rather simple to perform (syntactically).

Stability of boolean tests: Blocking guards can only check conditions on environment variables (input ports). Furthermore, variables used in the boolean test of a conditional or a loop must be local (internal) variables. This allows us to look at guards as synchronisation channels between processes. Since boolean condition checking takes no time at all, the check may only be made over local wires.

No interference on shared variables: Global registers may not be read from and written to simultaneously. This avoids non-determinism and interference which can appear when we cannot be sure whether a variable value has been updated or not at the moment it is read.

```
⟨number⟩

⟨variable⟩

⟨exp⟩

⟨bexp⟩

⟨var-list⟩   ::=  ⟨variable⟩
               |  ⟨variable⟩,⟨var-list⟩
⟨exp-list⟩   ::=  ⟨exp⟩
               |  ⟨exp⟩,⟨exp-list⟩
⟨guard⟩      ::=  #⟨number⟩
               |  @⟨variable⟩
               |  @posedge ⟨variable⟩
               |  @negedge ⟨variable⟩
               |  wait ⟨variable⟩
               |  ⟨guard⟩ or ⟨guard⟩

⟨statement⟩  ::=  skip
               |  ⟨var-list⟩=⟨exp-list⟩
               |  ⟨var-list⟩=⟨guard⟩ ⟨exp-list⟩
               |  ⟨guard⟩ ⟨var-list⟩ = ⟨exp-list⟩
               |  ⟨guard⟩
               |  ⟨variable⟩<=⟨guard⟩ ⟨exp⟩
               |  if ⟨bexp⟩ then ⟨statement⟩ else ⟨statement⟩
               |  if ⟨bexp⟩ then ⟨statement⟩
               |  while ⟨bexp⟩ do ⟨statement⟩
               |  do ⟨statement⟩ while ⟨bexp⟩
               |  forever ⟨statement⟩
               |  begin ⟨stmt-list⟩ end
               |  fork ⟨statement⟩ || ⟨statement⟩ join

⟨stmt-list⟩  ::=  ⟨statement⟩
               |  ⟨statement⟩ ; ⟨stmt-list⟩

⟨module⟩     ::=  initial ⟨stmt-list⟩
               |  always ⟨stmt-list⟩
               |  assign ⟨variable⟩=⟨exp⟩
               |  assign ⟨variable⟩=#⟨number⟩ ⟨exp⟩
               |  assign_T ⟨variable⟩=#⟨number⟩ ⟨exp⟩
               |  ⟨module⟩ || ⟨module⟩
```

Table 5.1: The syntax of a subset of Verilog

No super-steps: Using loops in a timed language can be rather dangerous. One must always make sure that loops cannot block the advance of time by providing an infinite number of steps to be executed before time can be moved forward.

5.3 The Formal Semantics

5.3.1 The Techniques Used

The semantics of Verilog will be described using Relational Duration Calculus. The relational chop operator is crucial in the semantics, since it allows the simple description of zero delayed assignments.

As already discussed, a distinction has to be made between local and global variables. We will assume that different processes have disjoint local alphabets. The set of global variables is also assumed to be disjoint from the local variables. A simple syntactic check may be done to ascertain these assumptions. If any name clashes exist, renaming will solve the problem.

Here, we will use the letters `u`, `v`, $\ldots$`z` to stand for the local or global variables, and we will specify whenever a particular variable can be only local or global. `U`, `V` will be lists of variables. `e`, `f` will be used to represent expressions such as `u OR v`, and `E`,`F` lists of such expressions. `P`, `Q`, `R` will be used for variables representing arbitrary chunks of code. `b`, `c` will stand for boolean expressions. D and E will stand for arbitrary duration formulas.

We will also use a function **start**(`e`), which, given an expression `e`, will modify individual variables `v` in the expression to $\overleftarrow{\mathtt{v}}$. For example, the expression **start**(`u or v`) is $\overleftarrow{\mathtt{u}}$ `or` $\overleftarrow{\mathtt{v}}$. This function may be easily defined using primitive recursion on the structure of expressions.

$\mathsf{var}(\mathtt{e})$ is defined to be the set of variables used in expression `e`.

Finally, we note that we will be defining the meaning of primitive instructions in terms of continuation semantics and a set of variables it can write to. For statement `P` in the language, we will not define its semantics $[\![\mathtt{P}]\!]$, but $[\![\mathtt{P}]\!]_W(D)$ — the semantics of `P` writing to variables W such that after the termination of `P`, the behaviour is described by duration formula D. In other words we will define the semantics of `P` continued by D with the extra information that the variables in W are written to by P. The type of the semantics is thus:

$$[\![\cdot]\!] \quad :: \quad Prog \rightarrow \mathbb{P}\mathcal{SV} \rightarrow \mathcal{RDF} \rightarrow \mathcal{RDF}$$

The full semantics of the higher level instructions are then given in terms of the continuation semantics of the constituent instructions. Unless otherwise specified, any relational chop operator appearing in the definition of $[\![\mathtt{P}]\!]_W(D)$ will have type W. Thus, for simplicity of presentation, $\fatsemi$ will be used to stand for $\overset{W}{\fatsemi}$.

Note that the set W will be the set of local variables and global registers to which the current process can write.

This approach to giving the semantics of Verilog specifications is necessary since the semantics of certain instructions (such as non-blocking assignments) depend on what happens after the instruction is executed. Thus, in general one cannot say that $[\![P;Q]\!] = [\![P]\!];[\![Q]\!]$.

5.3.2 Syntactic Equivalences

Before we start defining the semantics of Verilog, we will introduce some syntactic equivalences for Relational Duration Calculus formulae which will make the expression of the semantics more compact and hence simpler to follow.

- Sometimes it is necessary to state that two state variables behave similarly over the current interval:

$$P \approx Q \stackrel{def}{=} \lfloor P \Leftrightarrow Q \rfloor \wedge \overleftarrow{P} = \overleftarrow{Q} \wedge \overrightarrow{P} = \overrightarrow{Q}$$

- We will need to be able to rename state variables. $D[Q/P]$ read as D with P for Q is defined as follows:

$$D[Q/P] \stackrel{def}{=} \exists P \cdot (D \wedge P \approx Q)$$

- It is frequently necessary to state that a state variable has remained constant over the current time interval.

$$\mathsf{Const}(P) \stackrel{def}{=} P \approx \mathbf{1} \vee P \approx \mathbf{0}$$

  It will be convenient to overload this operator to act on sets:

$$\mathsf{Const}(S) \stackrel{def}{=} \bigwedge_{P \in S} \mathsf{Const}(P)$$

- A conditional operator will simplify descriptions of complex systems. This will be written as $D \triangleleft E \triangleright F$, and read D *if* E *else* F. It is defined as follows:

$$D \triangleleft E \triangleright F \stackrel{def}{=} (E \wedge D) \vee (\neg E \wedge F)$$

  The algebraic laws usually associated with the similarly defined operator in programming languages also hold here. Some such laws are:

$$\begin{array}{rcll}
D \triangleleft \mathbf{true} \triangleright E & = & D & \\
D \triangleleft \mathbf{false} \triangleright E & = & E & \\
E \triangleleft D \triangleright E & = & E & \text{idempotency} \\
E \triangleleft D \triangleright (F \triangleleft D \triangleright G) & = & (E \triangleleft D \triangleright F) \triangleleft D \triangleright G & \text{associativity}
\end{array}$$

5.3.3 The Primitive Instructions

Skip

This is the simplest of instructions to define. `skip` followed by a duration formula D simply behaves like D.

$$[\![\texttt{skip}]\!]_W(D) \stackrel{def}{=} D$$

Unguarded Assignments

The informal meaning of `v=e` is that `v` takes the value of `e`. This assignment takes no detectable time. Formally, we define the meaning of `v=e` followed by a duration formula D as follows:

$$[\![\texttt{v=e}]\!]_W(D) \stackrel{def}{=} (\overrightarrow{v} = \mathbf{start}(\texttt{e}) \wedge \mathsf{Const}(W - \{v\}) \wedge \lceil\rceil) \,;\, D$$

Extending these semantics for concurrent assignments is rather straightforward. If $V = (v_1, \ldots, v_n)$ is a variable list with corresponding expression list $E = (e_1, \ldots, e_n)$, then:

$$[\![\texttt{V=E}]\!]_W(D) \stackrel{def}{=} (\bigwedge_{i=1}^{n} \overrightarrow{\mathtt{v}}_i = \mathbf{start}(\mathsf{e}_i) \wedge \mathsf{Const}(W - V) \wedge \lceil\rceil) \fatsemi D$$

Guards

Guards block a process from continuing execution until a certain event happens, or a number of time units elapse. We can divide guards into two separate groups: simple and compound. Simple guards are of the form `@v`, `@posedge v`, `@negedge v`, `wait v` or `#n`. Compound guards are of the form $\mathtt{g}_1$ `or` ... `or` $\mathtt{g}_n$ where each $\mathtt{g}_i$ is a simple guard.

The most straightforward way of defining the semantics of general guards is to divide the definition into two parts: *blocked* and *continued*. *blocked* represents the possibility of the guard not yet being lowered whereas *continued* is used to describe the behaviour of the guard once it is eventually lowered. In both cases, the variables to which the current process can write remain unchanged. The semantics of a guard can be described by:

$$\begin{aligned} [\![\mathtt{g}]\!]_W(D) \quad &\stackrel{def}{=} \quad (\mathsf{Const}(W) \wedge \mathit{blocked}) \vee \\ &\qquad (\mathsf{Const}(W) \wedge \mathit{continued}) \fatsemi D) \end{aligned}$$

The natural relation between *blocked* and *continued* can be expressed as: *continued* holds exactly the first time unit that *blocked* does not continue to hold, or, more formally:

$$\mathit{continued} \Leftrightarrow \neg\ \mathit{blocked} \wedge (\lceil\rceil \vee (\Box_i\ \mathit{blocked})\ ;\ l = 1)$$

Thus, defining *blocked* is sufficient. Also, we always expect that if *blocked* holds, it also holds for any prefix of the current interval:

$$\mathit{blocked} \Leftrightarrow \Box_i\ \mathit{blocked}$$

These properties can be verified for all guards definined in this chapter. Furthermore, we can use these properties to prove other laws we would expect to hold for *blocked* and *continued*. For example, we would expect that if a guard is *blocked*, it has not yet *continued* (and vice-versa): $\mathit{blocked} \Leftrightarrow \neg\Diamond_i\ \mathit{continued}$. This can be proved on the basis of the above two properties.

For every simple guard `g` we will give the expressions for *blocked* and *continued*. The next section will then extend this idea for compound guards.

- $[\![\texttt{\#n}]\!]_W(D)$

  This is the simplest guard. It is not lowered until `n` time units have elapsed. It allows the process to continue after precisely `n` time units elapse.

$$\begin{aligned} \mathit{blocked} \quad &\stackrel{def}{=} \quad l < n \\ \mathit{continued} \quad &\stackrel{def}{=} \quad l = n \end{aligned}$$

- $[\![\texttt{@v}]\!]_W(D)$

  This guard is lowered once a rising or falling edge is detected on variable `v`.

$$\begin{array}{rcl} \mathit{blocked} \stackrel{def}{=} & & (\lfloor v \rfloor \wedge \overrightarrow{v} = \text{true}) \\ & \vee & (\lfloor \neg v \rfloor \wedge \overrightarrow{v} = \text{false}) \\ \mathit{continued} \stackrel{def}{=} & & (\lceil v \rceil \wedge \overrightarrow{v} = \text{false}) \\ & \vee & (\lceil \neg v \rceil \wedge \overrightarrow{v} = \text{true}) \end{array}$$

- $[\![$`@posedge v`$]\!]_W(D)$

  This is similar to the previous guard but sensitive only to rising edges.

$$\begin{array}{rcl} \mathit{blocked} & \stackrel{def}{=} & \lfloor v \rfloor \vee (\lfloor v \rfloor \ ; \lfloor \neg v \rfloor \wedge \overrightarrow{v} = \text{false}) \\ \mathit{continued} & \stackrel{def}{=} & \lfloor v \rfloor \ ; \lceil \neg v \rceil \wedge \overrightarrow{v} = \text{true} \end{array}$$

- $[\![$`@negedge v`$]\!]_W(D)$

  This is identical to `posedge v` but sensitive to falling rather than rising edges.

$$\begin{array}{rcl} \mathit{blocked} & \stackrel{def}{=} & \lfloor \neg v \rfloor \vee (\lfloor \neg v \rfloor \ ; \lfloor v \rfloor \wedge \overrightarrow{v} = \text{true}) \\ \mathit{continued} & \stackrel{def}{=} & \lfloor \neg v \rfloor \ ; \lceil v \rceil \wedge \overrightarrow{v} = \text{false} \end{array}$$

- $[\![$`wait v`$]\!]_W(D)$

  `wait v` waits until `v` becomes true. The only difference between this and `posedge v` is that this does not wait for a rising edge, and may thus be activated immediately.

$$\begin{array}{rcl} \mathit{blocked} \stackrel{def}{=} & \lfloor \neg v \rfloor \wedge \overrightarrow{v} = \text{false} \\ \mathit{continued} \stackrel{def}{=} & \lfloor \neg v \rfloor \wedge \overrightarrow{v} = \text{true} \end{array}$$

Compound Guards

Having defined the semantics for simple guards, we can now easily extend the idea for a complex guard of the form:

$$(\mathtt{g}_1 \ \mathtt{or} \ \ldots \mathtt{or} \ \mathtt{g}_n)$$

From the previous section, we know $\mathit{blocked}_i$ and $\mathit{continued}_i$ for each guard $\mathtt{g}_i$. These will now be used to work out *blocked* and *continued* for the complex guard. Rather than define the semantics of a general guard with n component guards, only the special case with two component guards is considered.

A double guard ($\mathtt{g}_1$ `or` $\mathtt{g}_2$) is blocked whenever both the constituent guards are blocked. It continues once either of the guards is lowered. We must make sure that the combined guard is lowered once the *first* of the individual guards is lowered. In other words, up to one time unit ago, both guards must still have been blocked. Formally, this translates to:

$$
\begin{array}{rcl}
blocked & \stackrel{def}{=} & blocked_1 \wedge blocked_2 \\
continued & \stackrel{def}{=} & (\lceil\rceil \vee (blocked_1 \wedge blocked_2) \, ; l = 1) \wedge \\
& & (continued_1 \vee continued_2)
\end{array}
$$

When manipulating the expressions it is clear that this operation is commutative, associative and idempotent. The associativity justifies the lack of use of brackets in complex guards (without specifying whether it is left or right associative). Furthermore, the semantics of a general complex guard can be simplified to:

$$
\begin{array}{rcl}
blocked & \stackrel{def}{=} & \bigwedge_{i=1}^{n} blocked_i \\
continued & \stackrel{def}{=} & (\lceil\rceil \vee (\bigwedge_{i=1}^{n} blocked_i \, ; l = 1)) \wedge \\
& & \bigvee_{i=1}^{n} continued_i
\end{array}
$$

Guarded Assignments

Given an assignment of the form `V=g E`, where `g` is a guard, we can now easily define its semantics in terms of the semantics of the guard itself. By 'remembering' the value of `E` at the beginning, we can define the semantics of the guarded assignment statement simply as the sequential composition of the guard followed by a zero-delay assignment of the variable to the remembered value of `E`:

$$
[\![\texttt{V=g E}]\!]_W(D) \stackrel{def}{=} \exists \alpha \cdot [\![\alpha\texttt{=E ; g ; V=}\alpha]\!]_W(D)
$$

The semantics of sequential composition will be given later.

For a guarded assignment of the form `g V=E`, the semantics are even simpler:

$$
[\![\texttt{g V=E}]\!]_W(D) \stackrel{def}{=} [\![\texttt{g ; V=E}]\!]_W(D)
$$

5.3.4 Constructive Operators

Conditional

The semantics of `if-then-else` statements follow in a rather natural fashion. If the boolean guard is true at the start of the time interval the statement behaves just like the `then` branch of instruction, otherwise it behaves like the `else` branch.

$$
[\![\texttt{if b then P else Q}]\!]_W(D) \stackrel{def}{=} [\![\texttt{P}]\!]_W(D) \triangleleft \mathbf{start}(\texttt{b}) \triangleright [\![\texttt{Q}]\!]_W(D)
$$

Conditionals with no `else` clause can also be defined:

$$
\texttt{if b then P} \equiv \texttt{if b then P else skip}
$$

Note that sometimes, we will also use the shorthand notation $\texttt{P} \triangleleft \texttt{b} \triangleright \texttt{Q}$ for the Verilog program `if (b) then P else Q`.

Iteration

The `while` statement is defined using recursion.

$$[\![\texttt{while b do P}]\!]_W(D) \quad \stackrel{def}{=} \quad \mu X \cdot [\![\texttt{P}]\!]_W(X) \lhd \mathbf{start}(\texttt{b}) \rhd D$$

Sometimes, we will use `b*P` as shorthand for `while (b) P`.

The `do P while b` construct is very similar and can be viewed as a special case of the `while b do P` statement:

$$\texttt{do P while b} \equiv \texttt{P; while b do P}$$

This will sometimes be abbreviated to `P*b`.

Verilog also provides another loop statement: `forever P`. As expected, `P` is repeatedly executed forever. Its semantics may thus be defined by replacing all `forever` statements with `while` statements and using the semantics already defined.

$$\texttt{forever P} \equiv \texttt{while (true) do P}$$

Sequential Composition

Thanks to the continuation semantics used, sequential composition is trivial to define. The semantics of `P;Q` followed by D can be defined as the semantics of `P` followed by the semantics of `Q` followed by D.

$$[\![\texttt{P;Q}]\!]_W(D) \quad \stackrel{def}{=} \quad [\![\texttt{P}]\!]_W([\![\texttt{Q}]\!]_W(D))$$

An immediate conclusion we can draw from this definition is the associativity of sequential composition in Verilog.

Internal Parallel Composition

We will use the term *internal* parallel composition to refer to the `fork ... join` operator as opposed to the external parallel composition between modules.

Composing two programs in parallel will act just like running both at the same time with the execution of the continuation resuming as soon as both programs terminate. Note that the output variables W of the process will be divided between the two processes such that `P` controls W_P and `Q` controls W_Q. The variable sets must partition W ie $W = W_P \cup W_Q$ and $W_P \cap W_Q = \emptyset$.

$$[\![\texttt{fork P} \parallel \texttt{Q join}]\!]_W(D) \quad \stackrel{def}{=} \quad \left(\begin{array}{ll} & [\![\texttt{P}]\!]_{W_P}(\mathsf{Const}(W_P) \mathbin{;} D) \wedge [\![\texttt{Q}]\!]_{W_Q}(D)) \\ \vee & [\![\texttt{Q}]\!]_{W_Q}(\mathsf{Const}(W_Q) \mathbin{;} D) \wedge [\![\texttt{P}]\!]_{W_P}(D)) \end{array} \right)$$

An immediate corollary of this definition is that internal parallel composition is commutative and associative.

5.3.5 Top-Level Instructions

At the top-level, we can now specify the complete semantics of a module without needing to take into consideration what happens afterwards. These instructions may thus be interpreted as closure operators — producing a closed system from an open one.

Initial Statements

Informally `initial P` specifies that `P` will be executed once at the beginning.

$$[\![\texttt{initial P}]\!] \quad \stackrel{def}{=} \quad [\![\mathtt{P}]\!]_{\alpha\mathtt{P}}(\mathsf{Const}(\alpha\mathtt{P}))$$

The alphabet of `P`, α`P`, is the set of the local variables used in the process `P` and also all the global variables assigned to by this process. These variables may be obtained using a syntactic pass over `P`. It must be made sure that the alphabets of different processes are pairwise disjoint.

Infinite Loops

`always P` specifies that `P` will be repeatedly executed forever. This is equivalent to repeating `P` indefinitely at the beginning:

$$\texttt{always P} \quad \equiv \quad \texttt{initial (forever P)}$$

Continuous Assignments

Another type of module is the `assign` module, which performs a continuous assignment. In this section, only the semantics of immediate `assign` modules are considered. The case with a delay between the input and output is discussed in a later section.

The module `assign v=e` makes sure that whenever expression `e` changes value, `v` immediately gets the new value. This behaviour is practically identical to that of a combinational circuit, where there is practically no delay between the input and output of the component. The semantics of such a statement are straightforward to define:

$$[\![\texttt{assign}\ \mathrm{v{=}e}]\!] \stackrel{def}{=} \surd(v = e)$$

One must make sure that no combinational circuits arise from such statements since such situations may lead to infinite changes in zero time. Two examples showing this type of infinite behaviour are:

$$\texttt{assign v = }\neg\texttt{v}$$
$$\texttt{assign v = }\neg\texttt{w} \parallel \texttt{assign w = }\neg\texttt{v}$$

A simple syntactic check suffices to ensure that no such behaviour can arise. It is discussed in detail in section 5.5.1.

Parallel Components

Given two top-level instructions in parallel, we will simply define their combined semantics as:

$$[\![\texttt{P} \parallel \texttt{Q}]\!] \quad \stackrel{def}{=} \quad [\![\texttt{P}]\!] \wedge [\![\texttt{Q}]\!]$$

Note that the following law justifies the use of *internal parallel composition* to describe the `fork ...join` construct:

$$\left(\begin{array}{ll} & \texttt{initial P} \\ \parallel & \texttt{initial Q} \end{array}\right) \equiv \texttt{initial fork P} \parallel \texttt{Q join}$$

This also justifies the use of `P` $\parallel$ `Q` as shorthand for `fork P` $\parallel$ `Q join`.

5.4 Other Extensions to the Language

5.4.1 Local Variables

It is sometimes desirable to declare local variables to make code more readable. The semantics of such a construct can be formalised as follows:

$$[\![\texttt{var v; P; end v;}]\!]_W(D) \quad \stackrel{def}{=} \quad \exists v \cdot [\![\texttt{P}]\!]_{W\cup\{v\}}(D)$$

For simplicity we assume that v is not in W. To relax this restriction, we would first rename the variable to a fresh instance and then use the semantics given above.

5.4.2 Variable Initialisation

It is usually very useful to initialise global registers to particular values. This is not possible because of the restriction that all assignments to such registers must be time guarded. However this restriction may be slightly weakened by introducing a new instruction:

$$\begin{array}{lcl} \langle\texttt{initialisation}\rangle & \texttt{::=} & \texttt{init}\ \langle\texttt{var-list}\rangle = \langle\texttt{bool-list}\rangle \\ \langle\texttt{module}\rangle & \texttt{::=} & \texttt{initial}\ \langle\texttt{initialisation}\rangle;\ \langle\texttt{statement}\rangle \\ & | & \texttt{forever}\ \langle\texttt{initialisation}\rangle;\ \langle\texttt{statement}\rangle \\ & & \vdots \end{array}$$

As expected, the variables in the initialisation list must be output variables of the current module.

The semantics extend naturally for this new construct:

$$\begin{array}{rcl} [\![\texttt{initial init V=E; P}]\!] & \stackrel{def}{=} & [\![\texttt{initial V=E;P}]\!] \\ [\![\texttt{always init V=E;P}]\!] & \stackrel{def}{=} & [\![\texttt{initial V=E; forever P}]\!] \end{array}$$

In most cases, the `init` keyword will be left out since its presence can be deduced immediately.

5.4.3 Delayed Continuous Assignments

Delayed continuous assignments behave like inertial delays between the input and the output: whenever a change in the input wires is noticed, an assignment of the input expression to the output is set to be executed after the delay. Any other assignment to the same variable due to execute before the new assignment is removed. In other words, the output changes value once the inputs have all remained constant for the specified delay.

$$
\begin{array}{lrl}
[\![\texttt{assign v = \#n e}]\!] \stackrel{def}{=} & & \left(\begin{array}{rl} & l < n \wedge \lfloor \neg v \rfloor) \\ \vee & (l = n \wedge \lceil \neg v \rceil); \mathbf{true} \end{array}\right) \wedge \\
& & (\exists \overline{b} \cdot \lceil \mathsf{var}(e) = \overline{b} \rceil) \xrightarrow{\mathtt{n}} \lceil v = \mathtt{n} \gg e \rceil \wedge \\
& \neg & (\exists \overline{b} \cdot \lceil \mathsf{var}(e) = \overline{b} \rceil) \xrightarrow{\mathtt{n}} \lceil v = 1 \gg v \rceil
\end{array}
$$

where $\overline{b}$ is a sequence of boolean values of the same length as the sequence of variables $\mathsf{var}(e)$. Thus, if the variables in expression `e` remain unchanged for `n` time units, `v` takes that value, while if any of the variables have changed value, `v` keeps its old value.

Notice that this behaviour does not fully encompass the behaviour of the `assign` statement in Verilog. Whenever the variables in the expression remain constant for exactly `n` time units, the simulator semantics allows one of two sequences of execution:

1. The change on one of the variables of `e` takes place before the assignment of `v` and thus pre-empts the upcoming assignment.
2. The assignment to `v` takes place before the change of value of the variable in expression `e`.

This situation thus leads to a non-deterministic choice, with the behaviour depending on how the simulator is implemented. This situation is undesirable since the result does not fully describe the (intuitive) corresponding hardware. More complex examples may exhibit even more unexpected behaviour. Our view is that a more hardware related approach is to take the behaviour as already described. The simpler, more direct semantics are easier to handle and to prove results in.

Hence, to make sure that these formal semantics correspond directly to the semantics of actual Verilog, one would have to make sure that changes to the variables in expression `e` would never occur exactly `n` time units apart. If on the other hand one is viewing these semantics from a hardware point of view, where the delayed continuous assignment corresponds to a high capacitance delay, these semantics should be adequate.

Note that `assign v=#n e` guarantees that `v` does not change more often than every `n` time units: $\mathcal{S}(v, n)$ and $\mathcal{S}(\neg v, n)$.

5.4.4 Transport Delay

Another type of delayed assignment is transport delayed assignment. As opposed to inertial delay, an event on the variables on the right hand side of the assignment take place no matter whether any other change occurs before the actual assignment takes place. Later, it will be shown how transport delay can be implemented as several inertial delays in sequence. Also, since we would like our approach to be extendible and applicable to other hardware description languages, we will also be using transport delay in the coming chapters. For this reason, we introduce a continuous transport delay assignment: $\texttt{assign}_T$ `v=#n e`.

$$[\![\texttt{assign}_T\ \texttt{v=\#n e}]\!] \stackrel{def}{=} \surd(v = n \gg e)$$

5.4.5 Non-Blocking Assignments

Non-blocking assignments behave just like blocking assignments except that the next statement in the module executes immediately without waiting for the assignment to terminate. Note that this may introduce concurrent threads controlling the same variables and thus the parallel composition operator already defined fails to correctly describe the semantics. The solution we opt for is based on the concept of *parallel by merge* as described and expounded upon in [HJ98]. The basic underlying idea is that each process controls a copy of the shared variables, which are then merged into the actual variable when parallel composition is performed. [HJ98] use this basic idea to generalise different forms of parallel composition ranging from shared clock parallel programs to concurrent logic programs. This approach is mentioned again in section 5.5.3, where resolution functions are discussed.

The semantics of non-blocking assignment are defined by:

$$[\![\texttt{v<=c e}]\!]_W(D) \stackrel{def}{=} [\![\texttt{v=c e}]\!]_{\{v\}}(\mathsf{Const}(v)) \underset{\{v\}}{\parallel} D$$

Since both processes running in parallel can control the variable `v`, they are composed together using a merging parallel composition operator. This operator allows for common variables to be changed by more than one process. Whenever a variable is assigned to by both processes it non-deterministically takes one of the values it is assigned to. It should now be clear that it is necessary to maintain an extra boolean state for every Verilog variable `v`: assign_v, which holds in those time slots when variable `v` has just been assigned a value.

The semantics of the merging parallel composition is given by renaming the common variable and then merging the two together:

$$\begin{array}{rcl} [\![P \underset{\{v\}}{\parallel} Q]\!]_W(D) & \stackrel{def}{=} & \exists v_P, v_Q, \mathsf{assign}_{v_P}, \mathsf{assign}_{v_Q} \cdot \\ & & \quad P[v_P, \mathsf{assign}_{v_P}/v, \mathsf{assign}_v] \wedge \\ & & \quad Q[v_Q, \mathsf{assign}_{v_Q}/v, \mathsf{assign}_v] \wedge \\ & & \quad \mathsf{Merge}(v_P, v_Q, v) \end{array}$$

$\mathsf{Merge}(v_1, v_2, v)$ manages variable v as a merging of the two variables v_1 and v_2. Obviously, assign_v is true exactly when either of assign_{v_1} and assign_{v_2} is true. Furthermore, for any time unit, the value of v is:

- the same as v_1 whenever v_1 has just been assigned a value, but not v_2,
- the same as v_2 whenever v_2 has just been assigned a value, but not v_1,
- the same as the previous value of v whenever neither v_1 nor v_2 have just been assigned,

- a non-deterministic choice between v_1 and v_2 when both have just been assigned a value.

Formally, this can be written as:

$$\begin{array}{rcl}
\mathsf{Merge}(v_1, v_2, v) & \stackrel{def}{=} & \lfloor \mathsf{assign}_v = \mathsf{assign}_{v_1} \vee \mathsf{assign}_{v_2} \rfloor \wedge \\
& & \Box(\lceil\lceil \mathsf{assign}_{v_1} \wedge \neg \mathsf{assign}_{v_2} \rceil\rceil \Rightarrow \lceil\lceil v = v_1 \rceil\rceil) \wedge \\
& & \Box(\lceil\lceil \mathsf{assign}_{v_2} \wedge \neg \mathsf{assign}_{v_1} \rceil\rceil \Rightarrow \lceil\lceil v = v_2 \rceil\rceil) \wedge \\
& & \Box(\lceil\lceil \mathsf{assign}_{v_1} \wedge \mathsf{assign}_{v_2} \rceil\rceil \Rightarrow \lceil\lceil v = v_1 \vee v = v_2 \rceil\rceil) \wedge \\
& & \Box(\lceil\lceil \neg \mathsf{assign}_{v_1} \wedge \neg \mathsf{assign}_{v_2} \rceil\rceil \Rightarrow \lceil\lceil v = 1 \gg v \rceil\rceil)
\end{array}$$

Obviously, the state variables assign_v need to be maintained by the model. This is done by adding the information that assign_v is true immediately after an assignment:

$$[\![\texttt{v=e}]\!]_W(D) \stackrel{def}{=} (\lceil\rceil \wedge \overrightarrow{\mathsf{assign}_v} = \text{true} \wedge \overrightarrow{v} = \mathbf{start}(e) \wedge \mathsf{Const}(W - \{v\}))\ ⨾\ D$$

Also, whenever a variable is kept constant with $\mathsf{Const}(v)$ we make sure that assign_v does not become true:

$$\mathsf{Const'}(v) \stackrel{def}{=} \mathsf{Const}(v) \wedge (\lceil\rceil \vee (l = 1 \wedge \mathsf{Const}(\mathsf{assign}_v))\ ⨾\ (\lfloor \neg \mathsf{assign}_v \rfloor \wedge \overrightarrow{\mathsf{assign}_v} = \mathbf{false})$$

In other words, assign_v keeps the value it has just been assigned prior to the current interval but then reverts to zero after one time unit.

Since a shared variable is not considered local, any input from a shared variable v is changed to read from v_{in}. At the topmost level, it would then be necessary to add the invariant: $\surd v_{in} = v$.

Finally note that the relational chop type is no longer the same as the alphabet of the process (W) but:

$$W \cup \{\mathsf{assign}_v \mid v \in W\}$$

As can be seen from the above discussion, using non-blocking assignments can considerably complicate the model. However, the extra information does not invalidate proofs about sub-systems which do not use non-blocking assignments. In fact, it is possible to split the semantics of a process into the conjunction of two separate parts: one controlling the variable values and the other recording whether the variables have been assigned to. If the program has no non-blocking assignments, the first conjunct will be identical to the semantics of the program as given in a previous section. In brief, proofs on programs not using non-blocking assignments can be done using the previous, simpler model. The extra information about assign_v would only be necessary in modules which use such assignments.

A difference from standard Verilog is that, in our case, blocking and non-blocking assignments have the same priority. In the case of standard Verilog, this is not true, and changes to variable values induced by non-blocking assignments are executed only when no other changes are possible. The model presented here may be extended to deal with priority (possibly by introducing multi-level signal strengths) but we choose to stay at a weaker, less deterministic level. For example consider the program `v<=#1 1; v=#1 0`. In Verilog, `v` would always end up with value `1`, since the normal (blocking) assignment `v=#1 0` would be executed before the non-blocking assignment of `v` to `1`. The semantics presented here would not guarantee this, and it would be possible to show only that after 1 time unit `v` has value `1` or `0`.

A more hardware oriented view is not to allow multiple drivers to assign a value to the common output at the same time. This may effectively lead to a short circuit, thus making the resultant circuit dangerous and its behaviour chaotic. The semantics of parallel merging would look like:

$$\llbracket P \underset{\{v\}}{\|} Q \rrbracket_W(D) \stackrel{def}{=} \exists v_P, v_Q, \mathsf{assign}_{v_P}, \mathsf{assign}_{v_Q} \cdot \Box(\lfloor \neg\mathsf{assign}_{v_P} \vee \neg\mathsf{assign}_{v_Q} \rfloor) \Rightarrow \ldots$$

Whenever a multiple assignment takes place, the result would thus be chaos (**true**) which implements only the trivial specification.

5.5 Discussion

5.5.1 Avoiding Combinational Loops

To avoid the possibility of combinational loops, it is sufficient to perform a syntactic check. This check can be formalised by defining a dependency relation $\mathsf{con}(\mathtt{P})$, where, if `(v,w)` is in the relation, then `v` is connected to `w`.

$$\begin{aligned}
\mathsf{con}(\mathtt{assign\ v=e}) &\stackrel{def}{=} \{\mathtt{(v,w)} \mid \mathtt{w} \text{ is a variable in } \mathtt{e}\} \\
\mathsf{con}(\mathtt{initial\ P}) &\stackrel{def}{=} \emptyset \\
\mathsf{con}(\mathtt{always\ P}) &\stackrel{def}{=} \emptyset \\
\mathsf{con}(\mathtt{P}\|\mathtt{Q}) &\stackrel{def}{=} (\mathsf{con}(\mathtt{P}) \cup \mathsf{con}(\mathtt{Q}))^+
\end{aligned}$$

where R^+ is the transitive closure of relation R.

To check that a program `P` has no combinational feedback, it is enough to make sure that no wire is related to itself, or formally, that:

$$\mathsf{con}(\mathtt{P}) \cap Id = \emptyset$$

where Id is the identity relation over the set of wires.

5.5.2 Time Consuming Programs

From the restrictions placed on Verilog, it is clear that we would like to assess whether an instruction always take time to execute or not. To do this we will define $\mathsf{dur}(\mathtt{P})$, meaning that `P` must always take time to execute, as follows:

$$\begin{aligned}
\mathsf{dur}(\mathtt{wait\ v}) &\stackrel{def}{=} \text{false} \\
\mathsf{dur}(\mathtt{@v}) &\stackrel{def}{=} \text{true} \\
\mathsf{dur}(\mathtt{@posedge\ v}) &\stackrel{def}{=} \text{true} \\
\mathsf{dur}(\mathtt{@negedge\ v}) &\stackrel{def}{=} \text{true} \\
\mathsf{dur}(\mathtt{\#n}) &\stackrel{def}{=} \text{true} \qquad \text{iff } n > 0
\end{aligned}$$

$$
\begin{array}{rcl}
\mathsf{dur}(\texttt{g1 or g2}) & \stackrel{def}{=} & \mathsf{dur}(\texttt{g1}) \wedge \mathsf{dur}(\texttt{g2}) \\
\mathsf{dur}(\texttt{v = e}) & \stackrel{def}{=} & \text{false} \\
\mathsf{dur}(\texttt{v = g e}) & \stackrel{def}{=} & \mathsf{dur}(g) \\
\mathsf{dur}(\texttt{g v = e}) & \stackrel{def}{=} & \mathsf{dur}(g) \\
\mathsf{dur}(\texttt{skip}) & \stackrel{def}{=} & \text{false} \\
\mathsf{dur}(\texttt{if b then P else Q}) & \stackrel{def}{=} & \mathsf{dur}(P) \wedge \mathsf{dur}(Q) \\
\mathsf{dur}(\texttt{while b do P}) & \stackrel{def}{=} & \text{false} \\
\mathsf{dur}(\texttt{P ; Q}) & \stackrel{def}{=} & \mathsf{dur}(P) \vee \mathsf{dur}(Q) \\
\mathsf{dur}(\texttt{fork P} \parallel \texttt{Q join}) & \stackrel{def}{=} & \mathsf{dur}(P) \vee \mathsf{dur}(Q)
\end{array}
$$

For any sequential program P, we can now show whether dur(P) is true or not. Note that it is possible to show that if dur(P) is true, the execution of P takes time to terminate:

$$(\llbracket \texttt{t=0;P;t=1} \rrbracket_W (\mathsf{Const}(W)) \wedge \overrightarrow{t} = 1) \quad \Rightarrow \quad l > 0$$

It is now possible to formalise some of the restrictions:

- Body of loops must take time: To use `while b do P`, dur(P) must be true.
- Guarded assignments to global registers: If `v` is a global register, and the current process can write to it, then `v=g e` and `g v=e` are acceptable provided that dur(g) is true.

The second restriction can be relaxed by making sure that any two assignments to a global variable are separated by a program of non-zero duration.

5.5.3 Extending Boolean Types

Verilog users may prefer to use the four valued logic with $\{\mathtt{x}, 0, 1, \mathtt{z}\}$ rather than the two valued one that is presented here. Such a system can be implemented by having two boolean states describing the value of the variables. For a variable `w`, `w.m` will be true if `w` has a meaningful value (0 or 1) and `w.v` will specify whether the value is high or low (the value is either 1 or `z` if `w.v` is true). This extended model is more complex and in many cases using 1 and 0 values is sufficient. It is for these reasons that the semantic values were limited to the two boolean ones.

An important use of the `z` and `x` values is when designing tri-state devices [Pal96, SSMT93]. Obviously, in this case, simply adding the extra values will not suffice since, in general, one would desire to have multiple drivers on the same variables and possibly also use resolution functions to arbitrate resultant values. Parallel merging is, once again, the way one could model such a situation. As before, each process writes to a local variable, where all these local values are then cascaded together using the merge operator. For example, in the case of a bidirectional bus where one device may want to send a high impedance value `z` to allow the other device to write a value, the part of the merge operator which manages this situation would look like the following:

$$\surd(v_1 = \mathtt{z} \Rightarrow v = v_2)$$

Once again notice the extra information which needs to be carried around to handle this kind of situation. Also, due to the nature of the parallel merge operator, which acts as a closure operator, our model loses some of its compositional aspects.

5.5.4 Concurrent Read and Write

The major constraint on the language is the fact that we do not allow concurrent read and write between processes. This constraint can be removed by making the reading of non-local variables in assignment statements take either the previous or next value non-deterministically. This should guarantee sound semantics even if we allow processes to read and write concurrently. However, the semantics will still not be complete since, for example, we would still not be able to prove that at the end of `@posedge v; w=v` the variable `w` has value 1.

5.6 Related Work

The main aim of [Goo93b] was to analyse how structure and behaviour interact in HDL specifications. However, it is one of the best overviews of the different approaches taken to formalise HDL behaviour as of 1993. Unfortunately, it exclusively discusses continuous assignment style specifications with no imperative programming features.

Since then considerable work which formalises HDLs has appeared. However, as we note in the introduction, not much work has been done on Verilog itself. A notable exception is the VFE (Verilog Formal Equivalence) project [GG95] set up by the M.J.C. Gordon and D.J. Greaves. [Gor95] set the scene for research being done by the VFE project and also served as direct inspiration for this work. [Gor98] shows how programs written in a subset of Verilog can be translated into Mealy machines hence enabling formal analysis of these programs. In [GG98] the interpretation is done down to a RTL (Register Transfer Level) specification language which is given a mathematical semantics. Two levels of semantics are discussed: simulation (or event) semantics discuss the behaviour at the simulation cycle level while trace semantics abstract over unstable and intermediate states. Two other semantic levels are identified but not yet analysed.

Most of the work done on industry standard HDLs, however, tends to deal with VHDL. Anthologies of formal semantics for VHDL have appeared in [KB95] and [Bor95].

A popular approach is to give an operational semantics to emulate the HDL simulation cycle. [Goo95] gives an operational semantics to VHDL, while [Tas90, TH91, Tas92] use HOL to formally specify the behaviour of the simulation cycle. The popularity of this sort of semantics for HDLs is natural, since most of the intricate behaviour is closely related to the way the languages are simulated. In our case, we prefer to restrict the language to give a more elegant and abstract semantics.

Other approaches have also been reported in the literature. Of particular interest is [WMS91] which proposes a semantics of VHDL based on an interval temporal logic. The authors formalise a relatively large subset of VHDL but leaving out delta delays. To deal with the relational-temporal nature of the language they use an interesting mechanism, whereby the behaviour is first specified in terms of steps in resource (as opposed to simulation) time, each of which corresponds to a simulator instruction execution. This is then abstracted away by introducing a global clock (counting simulation time) which synchronises the different behaviour traces. The resultant semantics convey the expected behaviour, but time related specifications in the resultant model lack high level abstraction.

Other denotational approaches have appeared in [BS95, BFK95, Dav93]. In particular, [BFK95] gives an elegant semantics using the idea of a world-line and current time point to describe the state. This semantics is then used to define an equivalent axiomatic semantics for VHDL. Various other approaches have been tried, most of which are far too detached from our presentation to merit discussion in detail. [KB95] is a good starting point for such research.

5.7 Conclusions

The semantics of the basic language are quite easy to handle. As we include other features, the semantics grow more and more complicated. This indicates the need for algebraic laws to reason about the language in a more abstract fashion. These laws will be given in chapter 6. An attractive part of the language semantics design is its modularity. One can reason about a program which uses only blocking assignments using the simple semantics. If a non-blocking assignment is to be added to the program, the behaviour of the extra boolean states can be derived separately. Also, provided that there is no interference between the non-blocking assignment and the already existent program, all previous results shown are still valid. This compositional approach to language semantics promotes and gives the opportunity to follow good and sound design methodologies.

Chapter 6

Algebraic Laws

6.1 Introduction

Having defined the semantics of the language in question, one can prove properties about programs. However, proving properties in terms of relational duration calculus can, at times, be very tedious. Sometimes we would simply like to transform a given implementation into an equivalent one, or refine a program into a more efficient version. Following the approach as used in [H$^+$85, RH86, HMC96] we give a number of algebraic equalities (and inequalities) which are satisfied by any Verilog program. These laws can be verified with respect to the semantics given in the previous chapter, thus allowing us to prove results using these laws rather than having to fall back onto the underlying semantics. The laws are given in terms of variables which can be replaced by any arbitrary program (unless there is a side condition which permits its use only on programs which respect certain properties).

6.2 Notation

Most of these laws give an equality between two programs. Since we are giving a continuation semantics of the language, it is important to explain what we mean when we say that $P = Q$. The first condition is that the alphabet of P is the same as that of Q. The other condition is that for any possible continuation, the two programs behave identically. Formally, this may be written as:

$$P = Q \quad \stackrel{def}{=} \quad [\![P]\!]_V(D) = [\![Q]\!]_V(D)$$

where D can range over all valid relational duration formulae with V being the alphabet of both P and Q.

However, in certain cases, full blown equality is far too strong a condition to satisfy. In such cases, we consider refinements: we say that a program Q refines a program P (written as $P \sqsubseteq Q$) if, under all possible continuations, the behaviour of Q implies that of P.

$$P \sqsubseteq Q \quad \stackrel{def}{=} \quad [\![Q]\!]_V(D) \Rightarrow [\![P]\!]_V(D)$$

Again, the implication must be satisfied for any relational duration formula D with V being the alphabet of P and Q.

Note that equality can be rephrased in terms of refinement as follows:

$$P = Q \equiv P \sqsubseteq Q \;\wedge\; Q \sqsubseteq P$$

Note that the left hand side and right hand side of the equalities and inequalities must have the same output alphabet.

6.3 Monotonicity

The first class of laws we will be giving state that the programming constructs in Verilog are monotonic. In other words, if we selectively refine portions of the program, we are guaranteed a refinement of the whole program. The proofs of these laws are given fully due to their importance. For the rest of the chapter we will only state the nature of the proof.

The following lemma will be needed to prove some laws in this chapter.

Lemma: For any program P with alphabet W, $[\![P]\!]_W$ is a monotonic function:

$$D \Rightarrow E \vdash [\![P]\!]_W(D) \Rightarrow [\![P]\!]_W(E)$$

Proof: The proof proceeds by structural induction on the program.

Base cases:

Skip:

$$\begin{array}{ll} & [\![\mathtt{skip}]\!]_W(D) \\ = & \{ \text{ by definition of semantics } \} \\ & D \\ \Rightarrow & \{ \text{ by premise } \} \\ & E \\ = & \{ \text{ by definition of semantics } \} \\ & [\![\mathtt{skip}]\!]_W(E) \end{array}$$

Assignment:

$$\begin{array}{ll} & [\![v = e]\!]_W(D) \\ = & \{ \text{ by definition of semantics } \} \\ & (\overrightarrow{v} = \mathbf{start}(e) \wedge \mathsf{Const}(W - \{v\}) \wedge \lceil\rceil) \mathbin{;} D \\ \Rightarrow & \{ \text{ by monotonicity of relational chop and premise } \} \\ & (\overrightarrow{v} = \mathbf{start}(e) \wedge \mathsf{Const}(W - \{v\}) \wedge \lceil\rceil) \mathbin{;} E \\ = & \{ \text{ by definition of semantics } \} \\ & [\![v = e]\!]_W(E) \end{array}$$

Guards:

$$\begin{array}{ll} & [\![g]\!]_W(D) \\ = & \{ \text{ by definition of semantics } \} \\ & (g_{nt} \wedge \mathsf{Const}(W)) \vee ((g_{ter} \wedge \mathsf{Const}(W)) \mathbin{;} D) \\ \Rightarrow & \{ \text{ by monotonicity of relational chop and disjunction and premise } \} \\ & (g_{nt} \wedge \mathsf{Const}(W)) \vee ((g_{ter} \wedge \mathsf{Const}(W)) \mathbin{;} E) \\ = & \{ \text{ by definition of semantics } \} \\ & [\![g]\!]_W(E) \end{array}$$

Inductive cases:

Sequential composition:

$$
\begin{array}{ll}
 & [\![Q;R]\!]_W(D) \\
= & \{ \text{ by definition of semantics } \} \\
 & [\![Q]\!]_W([\![R]\!]_W(D)) \\
\Rightarrow & \{ \text{ inductive hypothesis on } [\![\mathrm{Q}]\!]_W \text{ and } [\![\mathrm{R}]\!]_W \ \} \\
 & [\![Q]\!]_W([\![R]\!]_W(E)) \\
= & \{ \text{ by definition of semantics } \} \\
 & [\![Q;R]\!]_W(D)
\end{array}
$$

Conditionals:

$$
\begin{array}{ll}
 & [\![\texttt{if } b \texttt{ then } P \texttt{ else } Q]\!]_W(D) \\
= & \{ \text{ by definition of conditional } \} \\
 & [\![P]\!]_W(D) \triangleleft \overleftarrow{b} \triangleright [\![Q]\!]_W(D) \\
\Rightarrow & \{ \text{ monotonicity of DC conditional, inductive hypothesis and premise } \} \\
 & [\![P]\!]_W(E) \triangleleft \overleftarrow{b} \triangleright [\![Q]\!]_W(E) \\
= & \{ \text{ by definition of conditional } \} \\
 & [\![\texttt{if } b \texttt{ then } P \texttt{ else } Q]\!]_W(E)
\end{array}
$$

Loops:

$$
\begin{array}{ll}
 & [\![\texttt{while } b \texttt{ do } P]\!]_W(D) \\
= & \{ \text{ by definition of semantics } \} \\
 & \mu X \cdot [\![P]\!]_W(X) \triangleleft \overleftarrow{b} \triangleright D \\
\Rightarrow & \{ \text{ monotonicity, inductive hypothesis and premise} \} \\
 & \mu X \cdot [\![P]\!]_W(X) \triangleleft \overleftarrow{b} \triangleright E \\
= & \{ \text{ by definition of semantics } \} \\
 & [\![\texttt{while } b \texttt{ do } P]\!]_W(E)
\end{array}
$$

Internal parallel composition:

$$
\begin{array}{ll}
 & [\![\texttt{fork } P\texttt{; } Q \texttt{ join}]\!]_W(D) \\
= & \{ \text{ by definition of semantics } \} \\
 & \quad ([\![P]\!]_{W_P}(\mathsf{Const}(W_P) \mathbin{;} D) \wedge [\![Q]\!]_{W_Q}(D)) \\
 & \vee \quad ([\![P]\!]_{W_P}(D) \wedge [\![Q]\!]_{W_Q}(\mathsf{Const}(W_P) \mathbin{;} D)) \\
\Rightarrow & \{ \text{ monotonicity and inductive hypothesis } \} \\
 & \quad ([\![P]\!]_{W_P}(\mathsf{Const}(W_P) \mathbin{;} E) \wedge [\![Q]\!]_{W_Q}(E)) \\
 & \vee \quad ([\![P]\!]_{W_P}(E) \wedge [\![Q]\!]_{W_Q}(\mathsf{Const}(W_P) \mathbin{;} E)) \\
= & \{ \text{ by definition of semantics } \} \\
 & [\![\texttt{fork } P\texttt{; } Q \texttt{ join}]\!]_W(E)
\end{array}
$$

Note that the lemma automatically holds for commands defined in terms of these instructions and constructs (such as `g v=e` which is defined as `g; v=e`).

□

Law: Sequential composition is monotonic:

$$\text{If } P \sqsubseteq Q \text{ then } X;P \sqsubseteq X;Q \text{ and } P;X \sqsubseteq Q;X$$

Note that this law holds only for programs not using non-blocking assignments.

Proof: The lemma is needed to prove monotonicity on the first operand:

$$
\begin{array}{ll}
 & [\![X;P]\!]_W(D) \\
= & \{ \text{ definition of semantics } \} \\
 & [\![X]\!]_W([\![P]\!]_W(D)) \\
\Leftarrow & \{ \text{ by lemma and premise } \} \\
 & [\![X]\!]_W([\![Q]\!]_W(D)) \\
= & \{ \text{ definition of semantics } \} \\
 & [\![X;Q]\!]_W(D)
\end{array}
$$

Monotonicicity on the second operand is easy to establish:

$$
\begin{array}{ll}
 & [\![P;X]\!]_W(D) \\
= & \{ \text{ definition of semantics } \} \\
 & [\![P]\!]_W([\![X]\!]_W(D)) \\
\Leftarrow & \{ \text{ premise } \} \\
 & [\![Q]\!]_W([\![X]\!]_W(D)) \\
= & \{ \text{ definition of semantics } \} \\
 & [\![Q;X]\!]_W(D)
\end{array}
$$

□

Law: Parallel composition is monotonic:

$$\text{If } P \sqsubseteq Q \text{ then } X \| P \sqsubseteq X \| Q \text{ and } P \| X \sqsubseteq Q \| X$$

Proof: Monotonicity on the left operand:

$$
\begin{array}{ll}
 & [\![P \parallel X]\!]_W(D) \\
= & \{ \text{ definition of semantics } \} \\
 & \quad ([\![P]\!]_{W_P}(D) \wedge [\![X]\!]_{W_X}(\mathsf{Const}(W_X) \,;\, D)) \\
 & \vee \quad ([\![P]\!]_{W_P}(\mathsf{Const}(W_P) \,;\, D) \wedge [\![X]\!]_{W_X}(D)) \\
\Leftarrow & \{ \text{ premise and monotonicity } \} \\
 & \quad ([\![Q]\!]_{W_Q}(D) \wedge [\![X]\!]_{W_X}(\mathsf{Const}(W_X) \,;\, D)) \\
 & \vee \quad ([\![Q]\!]_{W_Q}(\mathsf{Const}(W_Q) \,;\, D) \wedge [\![X]\!]_{W_X}(D)) \\
= & \{ \text{ definition of semantics } \} \\
 & [\![Q \parallel X]\!]_W(D)
\end{array}
$$

The proof for the second branch follows identically.

□

Law: Conditionals are monotonic:

$$\text{If } P \sqsubseteq Q \text{ then } X \triangleleft b \triangleright P \sqsubseteq X \triangleleft b \triangleright Q \text{ and } P \triangleleft b \triangleright X \sqsubseteq Q \triangleleft b \triangleright X$$

Proof: Monotonicity of the left branch program:

$$
\begin{array}{ll}
 & [\![P \triangleleft b \triangleright X]\!]_W(D) \\
= & \{ \text{ definition of semantics } \} \\
 & [\![P]\!]_W(D) \triangleleft \overleftarrow{b} \triangleright [\![X]\!]_W(D) \\
\Leftarrow & \{ \text{ premise and monotonicity } \} \\
 & [\![Q]\!]_W(D) \triangleleft \overleftarrow{b} \triangleright [\![X]\!]_W(D) \\
= & \{ \text{ definition of semantics } \} \\
 & [\![Q \triangleleft b \triangleright X]\!]_W(D)
\end{array}
$$

The proof for the second branch follows identically.

□

Law: Loops are monotonic:

$$\text{If } P \sqsubseteq Q \text{ then } b * P \sqsubseteq b * Q \text{ and } P * b \sqsubseteq Q * b$$

Proof: Monotonicity of while loops on their only operand:

$$
\begin{array}{ll}
 & [\![b * P]\!]_W(D) \\
= & \{ \text{ definition of semantics } \} \\
 & \mu X \cdot [\![P]\!]_W(X) \triangleleft \overleftarrow{b} \triangleright D \\
\Leftarrow & \{ \text{ premise and monotonicity } \} \\
 & \mu X \cdot [\![Q]\!]_W(X) \triangleleft \overleftarrow{b} \triangleright D \\
= & \{ \text{ definition of semantics } \} \\
 & [\![b * Q]\!]_W(D)
\end{array}
$$

Monotonicity of repeat loops then follows immediately:

$$
\begin{array}{ll}
& P * b \\
= & \{ \text{ by definition } \} \\
& P; b * P \\
\sqsubseteq & \{ \text{ by monotonicity of while loops and sequential composition } \} \\
& Q; b * Q \\
= & \{ \text{ by definition } \} \\
& Q * b
\end{array}
$$

$\square$

As with monotonicity of sequential composition, this law holds only in the semantics of the language without non-blocking assignments.

These laws justify the use of algebraic representation. In other words, if we have a program context $C(P)$, then we can guarantee that

1. if $P \sqsubseteq Q$ then $C(P) \sqsubseteq C(Q)$. This follows directly from the laws just given and structural induction on the context C.
2. if $P = Q$ then $C(P) = C(Q)$. This follows from the equivalence between equality and refinement in both directions and point 1.

6.4 Assignment

Normally, in Verilog, immediate assignments do not follow the normal laws of assignment statements, as for instance used in [H+85]. For example, the program `v=~v; v=~v` is not equivalent to `v=v`, since a parallel thread may choose to read the input at that very moment. Another possibility is that the spike will trigger an edge guard.

In our case, the semantics we give to Verilog are for a specific subset of Verilog programs for which this equality holds. The problem has been bypassed by not allowing programs to read and write simultaneously. Also, spikes are not possible since assignments to global variables cannot be instantaneous.

The laws in this section follow immediately from the semantics of assignment and the definition of relational chop. In fact, as already mentioned in chapter 3, relational chop and pre- and post-values act almost like assignment, provided that unchanged variables are explicitly stated to be so. The following laws are very similar (and in some cases identical) to those in [H+85].

Law: Composition of immediate assignments:

$$
\begin{array}{lcl}
\texttt{v=e;w=f} & = & \texttt{v,w=e,f[e/v]} \\
\texttt{v=e;v=f} & = & \texttt{v=f[e/v]}
\end{array}
$$

These rules for combining single assignments can be generalised to parallel assignments:

Law: Parallel assignments can be commuted:

$$\texttt{V=E} \;=\; \pi\texttt{V=}\pi\texttt{(F)}$$

where π is a permutation.

This commutativity property is used to simplify the presentation of the parallel assignment composition law:

Law: Composition of parallel immediate assignments:

$$\texttt{U,V=E,F;U,W=G,H} \;=\; \texttt{U,V,W=G[E,F/U,V],F,H[E,F/U,V]}$$

where variable sequences V and W have no elements in common.

Since concurrent read and write is not allowed, we have the following law:

Law: Assignments distribute in and out of parallel composition:

$$\texttt{(V=E; } P\texttt{)} \parallel Q \;=\; \texttt{V=E; (}P \parallel Q\texttt{)}$$

To prove this law it is sufficient to note that the behaviour of Q is not affected by an initial change in the value of variable list V which can be proved by structural induction on the program.

If we are using the semantic model which does not allow for non-blocking assignments, an assignment of a variable to itself has no noticeable effect:

Law: Skip and self-assignment (using the semantics which do not allow for non-blocking assignments):

$$\texttt{v=v} \;=\; \texttt{skip}$$

The law follows immediately from the fact that $(\lceil\rceil \wedge \mathsf{Const}(W))$ acts like the left one of $\overset{W}{;}$. Note that if non-blocking assignments are used, the behaviour of the assignment can cause a situation where multiple values are assigned to the same variable at the same time.

Law: Programs and assignment — provided that P does not assign to v or free variables in e, all variables in e are in the output alphabet of P and v is not free in e:

$$\texttt{v=e; } P \;=\; \texttt{v=e; } P\texttt{; v=e}$$

Again, this law holds provided that there are no non-blocking assignments to v. The proof follows by structural induction on P to show that the post-value of v remains constant, as does that of e. The proof fails if we take the semantics which handle non-blocking assignment since the right hand side program guarantees that the post-value of assign_v is true whereas the left hand side program does not.

6.5 Parallel and Sequential Composition

Sequential composition is associative with skip as its left and right unit:

$$\begin{aligned} P;(Q;R) &= (P;Q);R \\ P;\texttt{skip} &= P \\ \texttt{skip};P &= P \end{aligned}$$

These laws follow immediately from the definition of skip and associativity of functional composition.

The following laws about parallel composition hold for internal parallel composition (using the fork ...join construct). The following law, however, allows most of these laws to be applied equally well to top-level parallel composition:

$$\texttt{initial}\ P \parallel \texttt{initial}\ Q \quad = \quad \texttt{initial fork}\ P \parallel Q\ \texttt{join}$$

This law follows from the fact that $\mathsf{Const}(W) \overset{W}{\fatsemi} \mathsf{Const}(W) = \mathsf{Const}(W)$.

Parallel composition is commutative and associative with `skip` as its unit:

$$\begin{array}{rcl} P \parallel Q & = & Q \parallel P \\ P \parallel (Q \parallel R) & = & (P \parallel Q) \parallel R \\ P \parallel \texttt{skip} & = & P \end{array}$$

The laws follow from associativity and commutativity of conjunction.

For programs which take time to execute ($\mathsf{dur}(P)$ holds), $\#\mathtt{1}$ is also a unit of the composition. This can be proved by structural induction on programs satisfying $\mathsf{dur}(P)$.

Provided that $\mathsf{dur}(P)$:

$$P \parallel \#\mathtt{1} \quad = \quad P$$

In fact $\#\mathtt{1}$ distributes in and out of parallel composition:

$$\#\mathtt{1}; P \parallel \#\mathtt{1}; Q \quad = \quad \#\mathtt{1}; (P \parallel Q)$$

Law: If P and Q have different output alphabets, we can show that composing P with Q refines P:

$$P \sqsubseteq (P \parallel Q)^1$$

The main application of this law, is when we choose to refine individual parallel threads independently. Thus, for example, if we prove that $P_i \sqsubseteq P_i'$ (i ranging from 1 to n), we can use induction on n and this law to deduce that:

$$P_1 \parallel \ \ldots \ \parallel P_n \sqsubseteq P_1' \mid \ \ldots \ \parallel P_n$$

[1]Note that this law cannot be applied if Q has a non-empty output alphabet (since the left and right hand side would have different output alphabets). To solve this problem, we can introduce a new program $\mathtt{chaos}_V$ which allows the variables V to change non-deterministically.

$$[\![\mathtt{chaos}_V]\!]_V(D) = \mathbf{true}$$

Clearly, any program outputting to V is a refinement of this program:

$$\mathtt{chaos}_V \sqsubseteq P$$

This allows us to deduce the intended meaning of the parallel composition law, since $P \sqsubseteq P$, the above law and monotonicity of parallel composition enable us to deduce:

$$(P \parallel \mathtt{chaos}_V) \sqsubseteq (P \parallel Q)$$

where V is the output alphabet of Q.

6.6 Non-determinism and Assumptions

Restricting ourselves to the Verilog constructs can be problematic when proving certain properties. Sometimes it is useful to introduce certain new constructs which may feature in the middle part of the proof but would eventually be removed to reduce the program back to a Verilog program. This approach is used extensively in the correctness proofs of the hardware compiler in chapter 10.

One useful construct is non-deterministic composition. Informally, the non-deterministic composition of two programs can behave as either of the two. More formally, we define the semantics of non-determinism ($\sqcap$) as:

$$[\![P \sqcap Q]\!]_W(D) \stackrel{def}{=} [\![P]\!]_W(D) \vee [\![Q]\!]_W(D)$$

From this definition it immediately follows that non-determinism is commutative, associative, idempotent and monotonic:

$$\begin{array}{rcl} P \sqcap Q & = & Q \sqcap P \\ P \sqcap (Q \sqcap R) & = & (P \sqcap Q) \sqcap R \\ P \sqcap P & = & P \\ \text{If } P \sqsubseteq Q \text{ then } P \sqcap R & \sqsubseteq & Q \sqcap R \\ \text{and } R \sqcap P & \sqsubseteq & R \sqcap Q \end{array}$$

Non-determinism also distributes over sequential composition since disjunction distributes over relational chop:

$$\begin{array}{rcl} P; (Q \sqcap R) & = & (P;Q) \sqcap (P;R) \\ (P \sqcap Q); R & = & (P;R) \sqcap (Q;R) \end{array}$$

Another useful class of statements we will use are assumptions. The statement *assume* b, expressed as $b^\top$, claims that expression b has to be true at that point in the program:

$$[\![b^\top]\!]_W(D) \stackrel{def}{=} (\lceil\rceil \vee \lceil b \rceil; \mathbf{true}) \wedge D$$

$false^\top$ and $true^\top$ are, respectively, the left zero and one of sequential composition:

$$\begin{array}{rcl} false^\top; P & = & false^\top \\ true^\top; P & = & P \end{array}$$

Conjunction of two conditions results in sequential composition of the conditions:

$$(b \wedge c)^\top = b^\top; c^\top$$

This law follows immediately from simple Duration Calculus reasoning. Two corollaries of this law are the commutativity and idempotency of assumptions with respect to sequential composition:

$$
\begin{aligned}
b^\top &= b^\top; b^\top \\
b^\top; c^\top &= c^\top; b^\top
\end{aligned}
$$

Disjunction of two conditions acts like non-determinism:

$$(b \vee c)^\top; P = (b^\top; P) \sqcap (c^\top; P)$$

Assumptions simply make a program more deterministic:

$$P \sqsubseteq b^\top; P$$

This holds since prepending an assumption to a program P simply adds another conjunct to the semantics of P.

6.7 Conditional

The conditional statement is idempotent and associative:

$$
\begin{aligned}
P \triangleleft b \triangleright P &= P \\
P \triangleleft b \triangleright (Q \triangleleft b \triangleright R) &= (P \triangleleft b \triangleright Q) \triangleleft b \triangleright R
\end{aligned}
$$

These properties follow immediately from the definition of the semantics of conditional and idempotency and associativity of Duration Calculus conditionals.

Sequential, parallel and non-deterministic composition distributes out of conditionals:

$$
\begin{aligned}
(P \triangleleft b \triangleright Q); R &= (P; R) \triangleleft b \triangleright (Q; R) \\
P \sqcap (Q \triangleleft b \triangleright R) &= (P \sqcap Q) \triangleleft b \triangleright (P \sqcap R) \\
P \parallel (Q \triangleleft b \triangleright R) &= (P \parallel Q) \triangleleft b \triangleright (P \parallel R)
\end{aligned}
$$

Finally, a number of laws can be given which convey better the meaning of the conditional statement:

$$
\begin{aligned}
P \triangleleft \mathtt{true} \triangleright Q &= P \\
P \triangleleft b \triangleright Q &= Q \triangleleft \neg b \triangleright P \\
P \triangleleft b \triangleright (Q \triangleleft b \triangleright R) &= P \triangleleft b \triangleright R \\
(P \triangleleft b \triangleright Q) \triangleleft b \triangleright R &= P \triangleleft b \triangleright R
\end{aligned}
$$

Again, these laws follow from the definition of the semantics and the laws of conditional in Duration Calculus.

Provided that the values of the variables in expression b are not changed immediately by programs P and Q, a conditional can be expressed in terms of non-determinism and assumptions:

$$P \triangleleft b \triangleright Q = (b^\top; P) \sqcap (\neg b^\top; Q)$$

In the language considered for hardware compilation (see chapter 10), the precondition is always satisfied. In the general language, one would have to check that P (Q) has a prefix P_1 (Q_1) such that $\mathsf{dur}(P_1)$ ($\mathsf{dur}(Q_1)$) and the variables of b are not assigned in P_1 (Q_1).

6.8 Loops

The recursive nature of loops can be expressed quite succinctly in terms of algebraic laws:

Law: Unique least fixed point: $Q = (P; Q) \triangleleft b \triangleright \texttt{skip}$ if and only if $Q = b * P$.

As immediate corollaries of this law, we can show that:

$$\begin{aligned}
&(b * P) \triangleleft b \triangleright \texttt{skip} = b * P \\
&Q = P; Q \text{ if and only if } Q = \texttt{forever}\ P \\
&Q = P; (Q \triangleleft b \triangleright \texttt{skip}) \text{ if and only if } Q = P * b
\end{aligned}$$

Law: The first law can be strengthened to: $Q = (P; Q) \triangleleft b \triangleright R$ if and only if $Q = (b * P); R$.

The following law allows us to move part of a loop body outside:

Law: If $Q; P; Q = Q; Q$ then $(P; Q) * b = P; (Q * b)$.

The unique fixed point law is one of the more interesting laws presented in this chapter, and we will thus give a complete proof of the result.

Lemma 1: If $\mathsf{dur}(P)$ holds, then there exist relational duration formulae D_1 and D_2 such that, for any relational duration formula D:

$$[\![P]\!]_W(D) = D_1 \vee (D_2 \wedge l \geq 1) \overset{W}{\fatsemi} D$$

where D_1 and D_2 are independent of D.

Proof: The proof proceeds by induction on the structure P with the following inductive hypothesis:

$$\begin{aligned}
[\![P]\!]_W(D) &= D_1 \vee (D_2 \wedge l \geq 1) \overset{W}{\fatsemi} D \qquad \text{if } \mathsf{dur}(P) \\
[\![P]\!]_W(D) &= D_1 \vee D_2 \overset{W}{\fatsemi} D
\end{aligned}$$

The proof follows in a routine manner and is thus left out.

$\square$

Lemma 2: If $\mathsf{dur}(P)$ holds and $l \leq n \Rightarrow Q = R$, then:

$$l \leq n + 1 \Rightarrow P; Q = P; R$$

Proof:

$$\begin{array}{cl}
& [\![P;Q]\!]_W(D) \\
= & \{ \text{ by definition of semantics } \} \\
& [\![P]\!]_W([\![Q]\!]_W(D)) \\
= & \{ \text{ by lemma 1 } \} \\
& D_1 \ \vee \ (D_2 \wedge l \geq 1) \mathbin{;} ([\![Q]\!]_W(D)) \\
= & \{ \ l \leq n+1 \text{ and chop definition } \} \\
& D_1 \ \vee \ (D_2 \wedge l \geq 1) \mathbin{;} (l \leq n \wedge [\![Q]\!]_W(D)) \\
= & \{ \text{ premise } \} \\
& D_1 \ \vee \ (D_2 \wedge l \geq 1) \mathbin{;} (l \leq n \wedge [\![R]\!]_W(D)) \\
= & \{ \ l \leq n+1, \text{ chop definition and monotonicity of DC operators } \} \\
& D_1 \ \vee \ (D_2 \wedge l \geq 1) \mathbin{;} ([\![R]\!]_W(D)) \\
= & \{ \text{ by lemma 1 } \} \\
& [\![P]\!]_W([\![R]\!]_W(D)) \\
= & \{ \text{ by definition of semantics } \} \\
& [\![P;R]\!]_W(D)
\end{array}$$

□

Lemma 3: $\forall n : \mathbb{N} \cdot (l \leq n \Rightarrow D = E)$ if and only if $D = E$.

Proof: Follows immediately from the underlying axiom $\exists n : \mathbb{N} \cdot l = n$.

□

To avoid long mathematical expressions, we will now define:

$$F(\alpha) \quad \stackrel{def}{=} \quad \texttt{if } b \texttt{ then } (P;\alpha) \texttt{ else skip}$$

We will also be referring to multiple applications of F to an arbitrary program α:

$$\begin{array}{rcl}
F^1(\alpha) & \stackrel{def}{=} & F(\alpha) \\
F^{n+1}(\alpha) & \stackrel{def}{=} & F(F^n(\alpha))
\end{array}$$

Lemma 4: If α is a solution of the equation $X = F(X)$, then α is also a solution to $X = F^n(X)$, for any positive integer n.

Proof: The proof follows immediately from the definition of $F^i(X)$:

Base case: $X = F(X) = F^1(X)$.

Inductive case:

$$\begin{array}{cl}
& X \\
= & \{ \text{ premise } \} \\
& F(X) \\
= & \{ \text{ monotonicity of } F \text{ and inductive hypothesis } \} \\
& F(F^n(X)) \\
= & \{ \text{ by definition of } F^{n+1} \ \} \\
& F^{n+1}(X)
\end{array}$$

□

Theorem: If α and β are solutions of the equation $X = F(X)$ and $\mathsf{dur}(P)$, then $\alpha = \beta$.

Proof: By lemma 3, it is sufficient to prove that:

$$\forall n : \mathbb{N} \cdot l \leq n \Rightarrow \alpha = \beta$$

We will now show by induction on n, that this is true for any value of n.

Base case: $n = 0$

We start by proving the following equality:

$$
\begin{array}{cl}
 & l \leq 0 \wedge \\
 & [\![P;\alpha]\!]_W(D) \\
= & \{ \text{ by definition of semantics } \} \\
 & l \leq 0 \wedge \\
 & [\![P]\!]_W([\![\alpha]\!]_W(D)) \\
= & \{ \text{ by lemma 1 } \} \\
 & l \leq 0 \wedge \\
 & (D_1 \ \vee \ (D_2 \wedge l \geq 1) \,;\, ([\![\alpha]\!]_W(D))) \\
= & \{ \text{ DC reasoning about interval length } \} \\
 & l \leq 0 \wedge D_1 \\
= & \{ \text{ DC reasoning about interval length } \} \\
 & l \leq 0 \wedge \\
 & (D_1 \ \vee \ (D_2 \wedge l \geq 1) \,;\, ([\![\beta]\!]_W(D))) \\
= & \{ \text{ by lemma 1 } \} \\
 & l \leq 0 \wedge \\
 & [\![P]\!]_W([\![\beta]\!]_W(D)) \\
= & \{ \text{ by definition of semantics } \} \\
 & l \leq 0 \wedge \\
 & [\![P;\beta]\!]_W(D)
\end{array}
$$

We will now use this equality to prove the base case of the induction:

$$
\begin{array}{cl}
 & [\![\alpha]\!]_W(D) \\
= & \{ \text{ by lemma 4 } \} \\
 & [\![F(\alpha)]\!]_W(D) \\
= & \{ \text{ by definition of } F \ \} \\
 & [\![\texttt{if } b \texttt{ then } (P;\alpha) \texttt{ else skip}]\!]_W(D) \\
= & \{ \text{ by definition of semantics } \} \\
 & [\![P;\alpha]\!]_W(D) \triangleleft \overleftarrow{b} \triangleright D \\
= & \{ \text{ monotonicity of conditional, } l \leq 0 \text{ and previous result } \} \\
 & [\![P;\beta]\!]_W(D) \triangleleft \overleftarrow{b} \triangleright D \\
= & \{ \text{ by definition of semantics } \} \\
 & [\![\texttt{if } b \texttt{ then } (P;\beta) \texttt{ else skip}]\!]_W(D) \\
= & \{ \text{ by definition of } F \ \} \\
 & [\![F(\beta)]\!]_W(D) \\
= & \{ \text{ by lemma 4 } \} \\
 & [\![\beta]\!]_W(D)
\end{array}
$$

This concludes the base case of the theorem.

Inductive case: $n = k + 1$

$$
\begin{array}{rl}
& [\![\alpha]\!]_W(D) \\
= & \{ \text{ by lemma 4 } \} \\
& [\![F^{k+1}(\alpha)]\!]_W(D) \\
= & \{ \text{ by definition of } F^{k+1} \ \} \\
& [\![F(F^k(\alpha))]\!]_W(D) \\
= & \{ \text{ by definition of } F \ \} \\
& [\![\texttt{if } b \texttt{ then } (P; F^k(\alpha)) \texttt{ else skip}]\!]_W(D) \\
= & \{ \text{ by inductive hypothesis, lemma 2 and monotonicity of conditional } \} \\
& [\![\texttt{if } b \texttt{ then } (P; F^k(\beta)) \texttt{ else skip}]\!]_W(D) \\
= & \{ \text{ by definition of } F \ \} \\
& [\![F(F^k(\beta))]\!]_W(D) \\
= & \{ \text{ by definition of } F^{k+1} \ \} \\
& [\![F^{k+1}(\beta)]\!]_W(D) \\
= & \{ \text{ by lemma 4 } \} \\
& [\![\beta]\!]_W(D)
\end{array}
$$

□

From this theorem we can conclude that:

Corollary: $Q = (P; Q) \triangleleft b \triangleright \texttt{skip}$ if and only if $Q = b * P$.

Proof: ($\Rightarrow$) By definition, $b * P$ satisfies the equation $X = F(X)$. By the first equation, so does Q. Hence, by the theorem $Q = b * P$.

($\Leftarrow$) We simply unravel the loop once.

□

6.9 Continuous Assignments

Continuous assignments with no delay can be applied syntactically. This is a direct consequence of Duration Calculus invariants' laws:

$$
\left(\begin{array}{ll} & \texttt{assign v=e(x)} \\ \| & \texttt{assign x=f} \end{array} \right) = \left(\begin{array}{ll} & \texttt{assign v=e(f)} \\ \| & \texttt{assign x=f} \end{array} \right)
$$

If variable w is stable for at least 2 time units, we can transform a transport delay into a number of unit inertial delays. Provided that $\mathcal{S}(w, 2)$ and $\mathcal{S}(\neg w, 2)$:

$$
\texttt{assign}_T \texttt{ v=\#n w} \quad = \quad \begin{array}{l} \texttt{var v}_0, \ldots \texttt{v}_n \\ \left(\begin{array}{ll} & \texttt{assign v}_1\texttt{=w} \\ \|_{i=2}^n & \texttt{assign v}_i\texttt{=\#1 v}_{i-1} \\ \| & \texttt{assign v=v}_n \end{array} \right) \\ \texttt{end v}_0, \ldots \texttt{v}_n \end{array}
$$

Note that this law cannot be used if `w` is not a single variable (since inertial delay is triggered not when the value of the whole expression changes but when the value of any of the variables in the expression changes).

For convenient presentation of certain properties, we will also allow programs of the form P`; assign v=e`. This will act like:

$$[\![P]\!]_W([\![\texttt{assign v=e}]\!] \wedge \mathsf{Const}(W - \{v\}))$$

Similarly, P`; (assign v=e` $\|$ Q`)` acts like:

$$[\![P]\!]_W([\![\texttt{assign v=e} \parallel \texttt{initial } Q]\!])$$

6.10 Communication

The use of synchronisation signals can greatly increase the readability of code. This type of communication can be handled using the subset of Verilog we are using.

These 'channels' can be implemented as global variables which have a normal value of zero, but go briefly up to one when a signal is sent over them. 'Wait for signal', $s?$, is thus easily implemented by:

$$s? \stackrel{def}{=} \texttt{wait } s$$

The method used in the Fibonacci number calculator in section 2.4.2 to send a signal on s was by using the following code portion: `s=#1 1;s<=#1 0`. Note that this program blocks the execution of future events along this thread by one time unit, which we would rather avoid.

To send a signal, without blocking the rest of the program for any measurable simulation time but leaving the signal on for a non-zero time measure, the non-blocking assignment statement can be quite handy:

$$s! \stackrel{def}{=} s = 1 \ ; \ s <= \#\delta\ 0$$

So what value of δ is to be used? Obviously, δ has to be larger than zero. However, taking a value of 1 or larger leads to a conflict in the program:

$$s! \ ; \ \#\delta \ ; \ s!$$

The solution is to use a value of 0.5 for δ. This may seem to invalidate all the previous reasoning based on discrete time. However, this is not the case. Effectively, what we have done is to reduce the size of the smallest time step to a level which normal Verilog programs will not have direct access to. Alternatively, one could have used a value of one for δ and multiplied all delays in the program by a factor of two (effectively allowing Verilog programs to use only even sized delays).

If non-blocking assignments are only used for signals, and signals are accessed only using $s!$ and $f(s)?$, we can guarantee the monotonicity of sequential composition and loops despite the presence of non-blocking assignments.

6.11 Algebraic Laws for Communication

6.11.1 Signal Output

Processes which write to a signal s are assumed to do so only using the command $s!$ (except for initialisation). We will use $s! \in P$ as shorthand for the (syntactic) check whether the command $s!$ occurs anywhere in program P. Similarly, $s! \notin P$ refers to the converse. In both cases, it is assumed that s is in the output alphabet of P.

Signals start off as false, provided that they are not initially written to:

$$\texttt{initial } P; Q = \texttt{initial } \neg s^{\top}; P; Q$$

provided that $\mathsf{dur}(P)$ and $s! \notin P$. The law can be proved by structural induction on program P.

$s!$ sets signal s to true:

$$s! = s!; s^\top$$

Between two programs which do not signal on s, both of which take time to execute, s is false:

$$P; Q = P; \neg s^\top; Q$$

provided that $s! \notin P; Q$ and $\mathsf{dur}(P)$ and $\mathsf{dur}(Q)$. Again, structural induction on P and Q is used to prove this property.

Output on a signal can be moved out of parallel composition:

$$(s!; P) \parallel Q = s!; (P \parallel Q)$$

The proof is almost identical to the similar law which moves assignments out of parallel composition.

Signalling and assumptions commute:

$$s!; b^\top = b^\top; s!$$

6.11.2 Wait on Signal

Waiting stops once the condition is satisfied:

$$(s^\top; P) \parallel (s?; Q) = (s^\top; P) \parallel Q$$

Follows immediately from the definition of the semantics of $s?$.

Execution continues in parallel components until the condition is satisfied:

$$(\neg s^\top; P; Q) \parallel (s?; R) = \neg s^\top; P; (Q \parallel s?; R)$$

provided that $s! \notin P$. As before, follows from the semantics of $s?$.

If we also add the constraint that $\mathsf{dur}(P)$, we can even afford an extra time unit:

$$(\neg s^\top; P; Q) \parallel (\#1; s?; R) = \neg s^\top; P; (Q \parallel s?; R)$$

6.11.3 Continuous Assignment Signals

Sometimes, it may be necessary to have signals written to by continuous assignments. The laws given to handle signal reading and writing no longer apply directly to these signals and hence need modification.

The following laws thus allow algebraic reasoning about a signal s written to by a continuous assignment of the form:

$$\texttt{assign } s = f(s_1, \ldots, s_n)$$

where all variables s_1 to s_n are signals. For s to behave like a normal signal (normally zero except for half unit long phases with value one), $f(0, 0, \ldots 0)$ has to take value 0.

If $b \Rightarrow f(\overline{s})$ then:

$$\texttt{assign } s = f(\overline{s}) \parallel b^\top; P \quad = \quad b^\top; s!; (\texttt{assign } s = f(\overline{s}) \parallel P)$$

If $b \Rightarrow \neg f(\overline{s})$ and $s_i \notin P$ then:

$$\texttt{assign } s = f(\overline{s}) \parallel b^\top; P; Q \quad = \quad b^\top; P; (\texttt{assign } s = f(\overline{s}) \parallel Q)$$

To sequentialise outputs on signals s_i:

$$\texttt{assign } s = f(\overline{s}) \parallel s_i!; P \quad = \quad s_i!; (\texttt{assign } s = f(\overline{s}) \parallel s_i^\top; P)$$

The above laws are consequences of the following law:

$$\texttt{assign } s = f(\overline{s}) \parallel P(s?) \quad = \quad \texttt{assign } s = f(\overline{s}) \parallel P(f(\overline{s})?)$$

This law is a consequence of Duration Calculus invariants' laws and the semantics of $s?$ and $s!$.

Furthermore, if P is the process controlling a signal s, and s starts off with a value 0, the value of $f(s)$ will remain that of $f(0)$ until P takes over.

$$(s = 0)^\top; f(s)?; P \quad = \quad (s = 0)^\top; f(0)?; P$$

(the underlying assumption is that s is a signal and in the output alphabet of P).

6.12 Signals and Merge

Signals now allow us to transform a process to use a different set of variables. The trick used is to define a merging process which merges the assignments on a variable to another.

The first thing to do is to make sure that we know whenever a variable has been assigned to. The technique we use is simply to send a signal on assign_v to make it known that v has just been assigned a value. Note that we will only allow this procedure to be used on global variables, which guarantees that no more than one assignment on a particular variable can take place at the same time.

We thus replace all assignments: `v=g e` by `v=g e;` assign_v`!` and similarly `g v=e` by `g v=e;` assign_v`!`

The program Merge will collapse the variables v_1 and v_2 into a single variable v, provided that they are never assigned to at the same time:

$$\begin{array}{lcl} \mathsf{Merge} & \stackrel{def}{=} & \mathtt{var}\ v_1, v_2, \mathsf{assign}_{v_1}, \mathsf{assign}_{v_2} \\ & & \left(\begin{array}{ll} & \mathtt{assign}\ \mathsf{assign}_v = \mathsf{assign}_{v_1} \vee \mathsf{assign}_{v_2} \\ \| & \mathtt{assign}\ v = (v_1 \triangleleft \mathsf{assign}_{v_1} \triangleright v_2) \triangleleft \mathsf{assign}_v \triangleright v^- \\ \| & \mathtt{assign}\ v^- = \#0.5\ \ v \end{array}\right) \\ & & \mathtt{end}\ v_1, v_2, \mathsf{assign}_{v_1}, \mathsf{assign}_{v_2} \end{array}$$

This merge program can be used, for instance to allow parallel programs effectively to use the same variables. It also obeys a number of laws which relate Merge to parallel composition and conditional:

$$\begin{array}{rcl} (P[v_1/v]; Q[v_2/v]) \parallel \mathsf{Merge} & = & P; Q \\ (P[v_1/v] \triangleleft b \triangleright Q[v_2/v]) \parallel \mathsf{Merge} & = & P \triangleleft b \triangleright Q \end{array}$$

Note that for the first law to hold, both sides of the equality must be valid programs.

The laws given in this section follow by structural induction on the program and the laws of invariants (which appear in Merge).

6.13 Conclusions

The aim of this chapter was to identify a number of algebraic laws which Verilog programs obey. As discussed in [H+85], the formal semantics allow us to look at programs as mathematical expressions. In mathematics, algebraic laws allow the manipulation and simplification of programs with relative ease, without having to refer back to the more complex underlying axioms. This is the main motivation behind the work presented in this chapter.

Part III

This part explores the use of the formal semantics. Chapter 7 presents the transformation of of a real-time specification into a Verilog implementation, while chapter 8 shows how a simple specification language for simple hardware components can be defined. A set of laws which decompose and eventually implement components in Verilog by following a simple decision procedure are also given and are used in a number of simple examples in chapter 9. Other examples of the use of these semantics for verification can also be found in [PH98].

Chapter 7

A Real Time Example

7.1 Introduction

One of the advantages of Verilog over most software languages is the control over the passage of time. This makes it possible to implement time sensitive specifications realistically. This chapter considers a standard real-time specification, and analyses its implementation in Verilog. The implementation is constructed, rather than divined, thus reducing the number of proofs necessary. This example should also indicate how general principles can be defined to be used to convert standard sections of a specification into correct code.

The example considered is that of a railway level crossing, using the real-time specification language based on duration calculus presented in chapter 4. The specification is a combination of evolution and stability constraints. These ensure that the system evolves and changes when required, but will not do so unnecessarily.

The contribution of this chapter is to show how one can derive a (correct) Verilog implementation of an abstract specification. The specification is taken from [HO94] where a clocked circuit implementation is also proved correct. The implementations strategies differ considerably, since we have more abstract constructs at our disposal. The approach used in this chapter is much more constructive than that used in [HO94]. The resulting implementations are quite different although they have a similar flavour, which may indicate that the specification language is already rather low level.

7.2 System Specification

7.2.1 The Boolean States Describing the System

We consider a railway crossing with a gate, traffic lights and railtrack.

1. The environment will provide information as to whether the gate is *open* or *closed*. The boolean states to represent these conditions are:

   - O for *open*
   - C for *closed*

2. The environment will also provide information whether the track is *empty* or occupied by an *approaching* train:

- E for *empty*
- A for *approaching*

3. The implementation must provide the environment with a signal to start *opening* or *closing* the gate:

   - O^{ing} for *opening*
   - C^{ing} for *closing*

4. The implementation will also be responsible for setting the traffic lights *red*, *yellow* and *green*:

   - R for *red*
   - Y for *yellow*
   - G for *green*

Note that we still have no restrictions implying that only one of the states can be true at any one time. Up to this point all boolean states are independent of one another. Thus, for instance, E and A can still both be satisfied simultaneously — the track being empty and occupied at the same time!

7.2.2 System Requirements

We will now list the requirements of the railway crossing. The requirements will be stated in terms of a specification constant ϵ. By designing an implementation in terms of ϵ, we will effectively have designed a family of implementations, for a family of specifications.

- Closing Gate

  Once a train starts approaching, the gate must start closing within $15 + \epsilon$ time units:

  $$Req_1 \stackrel{def}{=} A \stackrel{15+\epsilon}{\longrightarrow} C^{ing}$$

  Within ϵ time units, the light must not be green, but yellow or red:

  $$Req_2 \stackrel{def}{=} A \stackrel{\epsilon}{\longrightarrow} (Y \vee R)$$

  If $3 + \epsilon$ time units elapse with the gate closing but it has not yet closed, the light must be red:

  $$Req_3 \stackrel{def}{=} (C^{ing} \wedge \neg C) \stackrel{3+\epsilon}{\longrightarrow} R$$

- Opening Gate

  After just ϵ time units with the track empty, the gate must start opening:

  $$Req_4 \stackrel{def}{=} E \stackrel{\epsilon}{\longrightarrow} O^{ing}$$

If the gate starts opening but does not open within $3 + \epsilon$ time units, something must be wrong, and we turn the red light on:

$$Req_5 \stackrel{def}{=} (O^{ing} \wedge \neg O) \xrightarrow{3+\epsilon} R$$

Otherwise, the light is green:

$$Req_6 \stackrel{def}{=} E \xrightarrow{\epsilon} (G \vee R)$$

- Light Control

  The light must show at least one of red, yellow or green:

$$Req_7 \stackrel{def}{=} \surd(R \vee Y \vee G)$$

  No two colours may be on at the same time:

$$Req_8 \stackrel{def}{=} \surd(\neg(G \wedge Y)) \wedge \surd(\neg(R \wedge Y)) \wedge \surd(\neg(R \wedge G))$$

- Stability of the System

  It takes at least 4 time units to open or close the gate:

$$Req_9 \stackrel{def}{=} \mathcal{S}(O^{ing}, 4)$$

$$Req_{10} \stackrel{def}{=} \mathcal{S}(C^{ing}, 4)$$

7.2.3 System Assumptions

In order to prove our implementation correct, we make certain assumptions about the system. Otherwise, certain parts will not be implementable or, just as bad, trivially implementable (such as, for instance, by letting the red light stay on indefinitely).

The train takes at least 20 time units to pass through the crossing:

$$Ass_1 \stackrel{def}{=} \mathcal{S}(A, 20)$$

Trains are separated one from another by at least 10 time units:

$$Ass_2 \stackrel{def}{=} \mathcal{S}(E, 10)$$

As already discussed, boolean states A and E must be mutually exclusive:

$$Ass_3 \stackrel{def}{=} A \Leftrightarrow \neg E$$

7.3 Implementing the Specification

Ideally, the work to prove the correctness of an implementation with respect to a specification is reduced as much as possible. Proofs of specific, possibly optimised code against a specification usually requires a lot of hard tedious work. The approach we take is to design and transform the specification segments into an implementation in a progressive, constructive fashion. This considerably reduces the work necessary to provide a safe implementation.

If necessary, optimisation of the code can then be performed later, using algebraic laws which transform the program into a smaller or faster one, depending on the particular needs and available resources.

7.4 Specification and Implementation Decomposition

Note that in both the specification and implementation both composition operators $\wedge$ and $\|$ are semantically equivalent, provided we keep to the signal restrictions placed on Verilog. This allows us to construct an implementation piecewise from the requirements. The following laws show how this is done. In both cases, we assume that the parallel composition is legal:

- If $[\![P]\!] \Rightarrow R$ then $[\![P \| P']\!] \Rightarrow R$
- If for all i (i ranging from 1 to n), $[\![P]\!] \Rightarrow R_i$ then $[\![P]\!] \Rightarrow \bigwedge_{i=1}^{n} R_i$

Both proofs are rather straightforward. In the first case:

$$\begin{array}{ll} & [\![P \| P']\!] \\ \Rightarrow & \{ \text{ by definition of parallel composition } \} \\ & [\![P]\!] \wedge [\![P']\!] \\ \Rightarrow & \{ \wedge \text{ elimination } \} \\ & [\![P]\!] \\ \Rightarrow & \{ \text{ premise } \} \\ & R \end{array}$$

□

The second result is also easily proved by induction on n. When $n = 1$:

$$\begin{array}{ll} & [\![P]\!] \\ \Rightarrow & \{ \text{ premise } \} \\ & R_1 \\ \Rightarrow & \{ \text{ by definition of } \bigwedge_{i=1}^{1} R_i \ \} \\ & \bigwedge_{i=1}^{n} R_i \end{array}$$

As for the inductive case, we assume that $[\![P]\!] \Rightarrow \bigwedge_{i=1}^{n} R_i$ and prove that $[\![P]\!] \Rightarrow \bigwedge_{i=1}^{n+1} R_i$:

$$\begin{array}{ll} & [\![P]\!] \\ \Rightarrow & \{ \text{ premise and inductive hypothesis } \} \\ & R_{n+1} \wedge \bigwedge_{i=0}^{n} R_i \\ \Rightarrow & \{ \text{ by definition of } \bigwedge_{i=0}^{n+1} R_i \ \} \\ & \bigwedge_{i=0}^{n+1} R_i \end{array}$$

□

We can now thus shift our focus to the implementation of the individual requirements.

7.5 Calculation of the Implementation

7.5.1 Closing and Opening the Gate

Note that the behaviour of the closing state C^{ing} is completely specified by Req_1 and Req_{10}. The progress Req_1 may be weakened to $A \xrightarrow{\delta} C^{ing}$ where $\delta \leq 15+\epsilon$. Furthermore, the requirement suggests an implementation using a delayed continuous assignment. The following program satisfies this requirement:

`assign Cing = #`δ_1` A` $\qquad\qquad \delta_1 \leq 15+\epsilon$

Does this implementation also satisfy Req_{10}? Firstly, we note that from the assignment `assign v=#n e` it immediately follows that `v` is stable for n time units. Hence, by choosing δ_1 to be 4 or more, we also satisfy Req_{11}. In fact, however, from the assign statement and assumption 2 it can be shown that Req_{10} is also satisfied when $\delta_1 < 4$. We thus end up with the following program which satisfies Req_1 and Req_{10}:

`assign Cing = #`δ_1` A` $\qquad\qquad \delta_1 \leq 15+\epsilon$

Similarly, Req_4 and Req_9, specifying the behaviour of O^{ing} are satisfied by:

`assign Oing = #`δ_2` E` $\qquad\qquad \delta_2 \leq 15+\epsilon$

7.5.2 The Traffic Lights

Red

The behaviour of the red light is influenced by Req_3, Req_5, Req_7 and Req_8. Since we have already seen how to implement progress requirements, we try to construct such an implementation and then strive to satisfy the other requirements later. However, apparently there is a problem since the specification provides two, not one, progress requirements for red. This problem is solved by combining the two Req_3 and Req_5 into one:

$$((C^{ing} \wedge \neg C) \vee (O^{ing} \wedge \neg O)) \xrightarrow{3+\epsilon} R$$

Using the monotonicity of the *leads to* operator, it is obvious that if this requirement is satisfied, then so are Req_3 and Req_5.

Using the approach advocated in the previous section, this is implemented by:

```
   assign Malfunction = (Cing and ~C) or (Oing and ~O)
|| assign R = #δ3 Malfunction                       δ3 ≤ 3 + ϵ
```

Note that the decomposition of the implementation into two parts is done to make the program clearer and to avoid complications arising from situations where the individual variables, but not the value of the whole expression, change.

Yellow

The invariant conditions given in Req_7 and Req_8 indicate that at least one of the traffic lights could be implemented to be on whenever the other two are off. This immediately satisfies Req_7 and part of Req_8. For this we can choose either yellow or green (since red has already been implemented). We choose yellow, although an implementation could just as easily have been constructed if green had been chosen.

```
assign Y = ~G and ~R
```

Green

Replacing Y by $\neg G \wedge \neg R$ reduces Req_7 to true. Requirement 8 is simply reduced to:

$$\surd(\neg(G \wedge R))$$

Also, using this replacement and replacing E by $\neg A$ (from assumption 3), Req_2 and Req_6 reduce to:

$$\begin{array}{rcl} \neg A & \xrightarrow{\epsilon} & G \vee R \\ A & \xrightarrow{\epsilon} & \neg G \vee R \end{array}$$

If we use an intermediate state A^-, we can use monotonicity of the leads to operator to decompose these requirements into:

$$\begin{array}{rcl} \neg A & \xrightarrow{\epsilon} & \neg A^- \\ A & \xrightarrow{\epsilon} & A^- \\ \surd(A^- & \Rightarrow & (\neg G \vee R)) \\ \surd(\neg A^- & \Rightarrow & (G \vee R)) \end{array}$$

The first two lines are implemented by a continuous assignment:

assign A^- = #δ_4 A	$\delta_4 \leq \epsilon$

Using propositional calculus transformations, it can easily be shown that if G is replaced by $\neg A^- \wedge \neg R$, the remaining two invariant conditions and the transformed version of Req_8 are all satisfied.

```
assign G = ~A⁻ and ~R
```

This result may be reached by propositional calculus reasoning or by simply drawing up a truth table of the invariants with the boolean states G, A^-, R and expressing G as a function of the other boolean states.

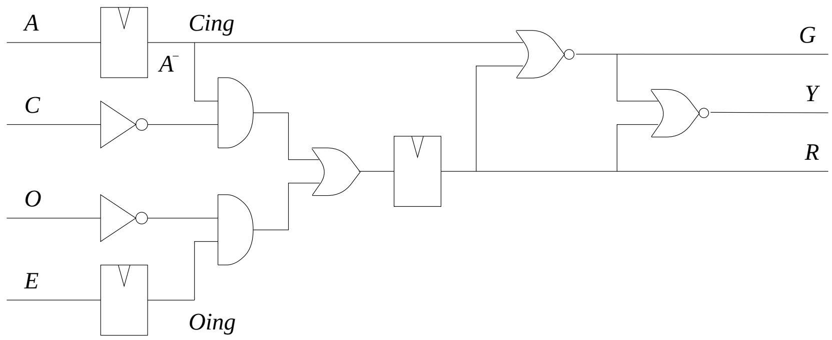


Figure 7.1: A circuit satisfying the real-time constraints

7.6 The Implementation

A whole family of implementations has thus been calculated from the requirements. The whole implementation is the following:

```
     assign Cing = #δ1 A
  || assign Oing = #δ2 E
  || assign Malfunction = (Cing and ~C) or (Oing and ~O)
  || assign R = #δ3  Malfunction
  || assign Y = ~G and ~R
  || assign A⁻ = #δ4 A
  || assign G = ~A⁻ and ~R
```

where the following restrictions are placed on the delays:

$$\begin{array}{rcl} \delta_1 & \leq & 15+\epsilon \\ \delta_2 & \leq & 15+\epsilon \\ \delta_3 & \leq & 3+\epsilon \\ \delta_4 & \leq & \epsilon \end{array}$$

Thus, for example, if ϵ is at least 1, we can satisfy all constraints by taking $\delta_1 = \delta_2 = \delta_3 = \delta_4 = 1$. This implementation is shown in figure 7.1.

7.7 Other Requirements

The original specification had one other requirement: The gate must remain open for at least 15 time units:

$$Req_{11} \stackrel{def}{=} \mathcal{S}(O, 15)$$

Two other assumptions about the system environment were also given to provide valid implementations of the specification:

- The gate can only open when it receives an opening signal.

$$Ass_4 \stackrel{def}{=} \Box(\lceil \neg O \rceil \,;\, \lceil O \rceil \Rightarrow true \,;\, \lceil O^{ing} \rceil \,;\, \lceil O \rceil)$$

- It remains open unless a closing signal is received:

$$Ass_5 \stackrel{def}{=} O \ unless \ C^{ing}$$

Note that the new requirement places a constraint on one of the system inputs, not outputs, and is thus not possible to prove directly as part of the construction.

Furthermore, the implementation given here does not satisfy the new requirement and would thus require a more intelligent approach.

One possible implementation of the system would be to implement C^{ing} more intelligently as follows:

$$\begin{array}{ll} & \texttt{assign Cing = \#}\delta_1\texttt{ K} \\ \| & \texttt{assign K = A}\wedge\neg\texttt{Oing} \end{array}$$

Note that Req_1 and Req_{10} are still satisfied using this implementation provided that $\delta_1 + \delta_2 \leq 15 + \epsilon$.

Provided that $\epsilon \geq 15$, we can satisfy the new requirement by taking $\delta_1 = \delta_2 = 15$. Note that a more general implementation can also be derived using less constructive techniques together with more intuition.

Chapter 8

Hardware Components

It is desirable to have a number of circuit properties from which one is able to construct more complex circuit specifications. This chapter gives a number of simple components which one may use to describe hardware. A normal form for this specification language is identified and laws are given to transform arbitrary specifications into this normal form. Finally, a set of laws converting normal form specifications into Verilog code are also given.

8.1 Combinational Circuits

One common property is that of having a boolean state always carrying a function of a number of other states with no delay in between. This is basically a combinational circuit. Such a property is specified as follows:

$$\mathcal{C}(v \longleftarrow e) \stackrel{def}{=} \surd(v = e)$$

8.2 Transport Delay

Quite frequently, it is necessary to have a signal carrying the same information as another wire but delayed by a number of time units. This type of delay is called *transport delay* and can be specified by:

$$\mathcal{T}(v \stackrel{\delta}{\longleftarrow} e) \stackrel{def}{=} \surd(v = \delta \gg e)$$

8.3 Inertial Delay

A rather more complex type of delay is inertial delay. An inertially delayed signal changes its value only after the inputs have remained constant for a given number of time units. We can describe this delay by:

$$\mathcal{I}(v \stackrel{\delta}{\longleftarrow} e) \quad \stackrel{def}{=} \quad \begin{array}{rrcl} & \mathsf{Stable}(\mathsf{var}(e)) & \stackrel{\delta}{\longrightarrow} & \lceil v = \delta \gg e \rceil \\ \wedge & \neg\mathsf{Stable}(\mathsf{var}(e)) & \stackrel{\delta}{\longrightarrow} & \lceil v = 1 \gg v \rceil \end{array}$$

where $\mathsf{Stable}(v) \stackrel{def}{=} \lfloor v \rfloor \vee \lfloor \neg v \rfloor$, and $\mathsf{Stable}(V) \stackrel{def}{=} \bigwedge_{v \in V} \mathsf{Stable}(v)$.

Thus, if the variables in expression e remain unchanged for δ time units v takes that value, while if any of the variables have changed value, v keeps its old value.

8.4 Weak Inertial Delay

The inertial delay behaviour, as described in the previous section can, in some circumstances, be too strong. Usually, it is enough to ensure the positive behaviour of the inertial delay: that the output is specified whenever the input has remained constant for at least δ time units. The extra requirement, that whenever the inputs have just changed, the output remains constant may be rather difficult to implement in certain circumstances, and may not be necessary for the whole design. For this reason, we define a weak inertial delay:

$$\mathcal{I}^-(v \xleftarrow{\delta} e) \quad \stackrel{def}{=} \quad \mathsf{Stable}(\mathsf{var}(e)) \xrightarrow{\delta} \lceil v = \delta \gg e \rceil$$

8.5 Edge Triggered Devices

An edge triggered register will output its current state, which is changed to the input whenever a rising (or falling) edge is encountered on the trigger signal. This device will be written as $\uparrow T \models v \longleftarrow e$ for input e, output v and trigger signal T. It is sensitive to a rising edge on T (a downwards arrow instead of the upwards one is used for falling edge triggered devices). To avoid feedback we assume that T is not free in e.

A delayed edge triggered device acts just like a normal edge triggered device but the state is changed δ units after the trigger event to the value of the input (at the trigger event). We assume that the trigger will not have more than one rising edge within δ time units of another. As before, T must not be free in e.

The formal definition of a delayed rising edge triggered device follows:

$$\uparrow T \models v \xleftarrow{\delta} e \quad \stackrel{def}{=} \quad \begin{array}{rrcl} & \overleftarrow{\mathrm{Rise}}(T) & \xrightarrow{\delta} & \lceil v = \delta \gg e \rceil \\ \wedge & \neg\overleftarrow{\mathrm{Rise}}(T) & \xrightarrow{\delta} & \lceil v = 1 \gg v \rceil \end{array}$$

where $\overleftarrow{\mathrm{Rise}}(T)$ states that T has risen at the start of the current interval: $\overleftarrow{\mathrm{Rise}}(T) \stackrel{def}{=} (\lceil\rceil \wedge \overleftarrow{T} = \mathrm{false} \wedge \overrightarrow{T} = \mathrm{true}); \mathbf{true}$.

Immediate edge triggered components can now be defined in terms of the delayed version:

$$\uparrow T \models v \longleftarrow e \quad \stackrel{def}{=} \quad \uparrow T \models v \xleftarrow{0} e$$

8.6 Weak Edge Triggered Devices

As we did with the inertial delay it is sometimes sufficient to describe the behaviour of an edge triggered device weaker than the ones introduced in the previous section. It is usually sufficient to specify that δ time units after a trigger, the value of the output will change to the value of the input as it was at the triggering event. The output will not change until another trigger is detected.

Note that we are weakening the specification by not determining the behaviour of the output from when the trigger is detected until δ time units later. It is usually far easier to satisfy this behaviour, and it is usually sufficient for most behavioural purposes. Formally, this behaviour is specified as:

$$\uparrow T \models^{-} v \xleftarrow{\delta} e \quad \stackrel{def}{=} \quad \begin{array}{rrcl} & \overleftarrow{\mathrm{Rise}}(T) & \xrightarrow{\delta} & \lceil v = \delta \gg e \rceil \\ \wedge & \neg\overleftarrow{\mathrm{Rise}}(T) \wedge \neg\mathrm{Rise}(T) & \xrightarrow{\delta} & \lceil v = 1 \gg v \rceil \end{array}$$

where $\mathrm{Rise}(T) \stackrel{def}{=} \Diamond(\lceil \neg T \rceil \,;\, \lceil T \rceil)$.

$$\uparrow T \models^{-} v \longleftarrow e \quad \stackrel{def}{=} \quad \uparrow T \models^{-} v \xleftarrow{0} e$$

The behaviour of a weak falling edge triggered device can be similarly defined.

8.7 Hiding

Specifications are usually built hierarchically. Modules are specified in terms of sub-modules, and the process is repeated until the resultant modules are simple enough to specify directly. At every stage of abstraction, it is convenient to be able to hide away information which will no longer be useful at the higher levels. This is done via the hiding operator:

$$\begin{array}{ll} \texttt{var} & v \\ & P \\ \texttt{end} & v \end{array} \quad \stackrel{def}{=} \quad \exists v \cdot P$$

Since the hiding operator is commutative, we will extend these semantics to:

$$\begin{array}{ll} \texttt{var} & u, \ldots w \\ & P \\ \texttt{end} & u, \ldots w \end{array} \quad \stackrel{def}{=} \quad \exists u, \ldots w \cdot P$$

8.8 Composition of Properties

Every component has a defined set of outputs, and a relation defining how the signals are dependent on each other (in a combinational fashion). These are defined in figures 8.1 and 8.2, where $\emptyset$ signifies the empty set and $\times$ is the Cartesian product of two sets.

Given that two hardware properties P_1 and P_2 satisfy the following two conditions, we will be able to define their composition $P_1 \| P_2$.

Condition 1: $\mathsf{out}(P_1) \cap \mathsf{out}(P_2) = \emptyset$

Condition 2: $(\mathsf{dep}(P_1) \cup \mathsf{dep}(P_2))^{+} \cap \mathrm{Id} = \emptyset$

$$
\begin{array}{lcl}
\mathsf{out}(\mathcal{C}(v \longleftarrow e)) & \stackrel{def}{=} & \{v\} \\
\mathsf{out}(\mathcal{T}(v \stackrel{\delta}{\longleftarrow} e)) & \stackrel{def}{=} & \{v\} \\
\mathsf{out}(\mathcal{I}(v \stackrel{\delta}{\longleftarrow} e)) & \stackrel{def}{=} & \{v\} \\
\mathsf{out}(\mathcal{I}^{-}(v \stackrel{\delta}{\longleftarrow} e)) & \stackrel{def}{=} & \{v\} \\
\mathsf{out}(\uparrow T \models v \stackrel{\delta}{\longleftarrow} e) & \stackrel{def}{=} & \{v\} \\
\mathsf{out}(\downarrow T \models v \stackrel{\delta}{\longleftarrow} e) & \stackrel{def}{=} & \{v\} \\
\mathsf{out}(\uparrow T \models^{-} v \stackrel{\delta}{\longleftarrow} e) & \stackrel{def}{=} & \{v\} \\
\mathsf{out}(\downarrow T \models^{-} v \stackrel{\delta}{\longleftarrow} e) & \stackrel{def}{=} & \{v\} \\
\mathsf{out}(\uparrow T \models v \longleftarrow e) & \stackrel{def}{=} & \{v\} \\
\mathsf{out}(\downarrow T \models v \longleftarrow e) & \stackrel{def}{=} & \{v\} \\
\mathsf{out}(\mathtt{var}\ v\ P\ \mathtt{end}\ v) & \stackrel{def}{=} & \mathsf{out}(P) - \{v\}
\end{array}
$$

Figure 8.1: The Output function

where R^+ is the transitive closure of a relation, and Id is the identity function over wires.

The first condition makes sure that no two properties control the same outputs. The second makes sure that there are no combinational loops.

We can now define the composition of two properties as follows:

$$
\begin{array}{rcl}
P_1 \parallel P_2 & \stackrel{def}{=} & P_1 \wedge P_2 \\
\mathsf{out}(P_1 \parallel P_2) & \stackrel{def}{=} & \mathsf{out}(P_1) \cup \mathsf{out}(P_2) \\
\mathsf{dep}(P_1 \parallel P_2) & \stackrel{def}{=} & (\mathsf{dep}(P_1) \cup \mathsf{dep}(P_2))^+
\end{array}
$$

8.9 Algebraic Properties of Hardware Components

A number of algebraic properties pertaining to the specification components will help us to prove theorems without having to fall back onto the original definitions.

8.9.1 Combinational Properties

The laws of combinational properties listed here will frequently be used to split specifications into simpler ones.

Law AP-1 *Equivalence of combinational properties*
Combinational circuits which compute equivalent propositions are themselves equivalent.

$$
\begin{array}{l}
\textit{If } e = f \textit{ then} \\
\qquad \mathcal{C}(v \longleftarrow e) = \mathcal{C}(v \longleftarrow f) \qquad\qquad [\textbf{Law} - \textbf{equiv}\mathcal{C}]
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{dep}(\mathcal{C}(v \longleftarrow e)) & \stackrel{def}{=} & \{v\} \times \mathsf{var}(e) \\
\mathsf{dep}(\mathcal{T}(v \stackrel{\delta}{\longleftarrow} e)) & \stackrel{def}{=} & \emptyset \\
\mathsf{dep}(\mathcal{I}(v \stackrel{\delta}{\longleftarrow} e)) & \stackrel{def}{=} & \emptyset \\
\mathsf{dep}(\mathcal{I}^-(v \stackrel{\delta}{\longleftarrow} e)) & \stackrel{def}{=} & \emptyset \\
\mathsf{dep}(\uparrow T \models v \stackrel{\delta}{\longleftarrow} e) & \stackrel{def}{=} & \emptyset \\
\mathsf{dep}(\downarrow T \models v \stackrel{\delta}{\longleftarrow} e) & \stackrel{def}{=} & \emptyset \\
\mathsf{dep}(\uparrow T \models^- v \stackrel{\delta}{\longleftarrow} e) & \stackrel{def}{=} & \emptyset \\
\mathsf{dep}(\downarrow T \models^- v \stackrel{\delta}{\longleftarrow} e) & \stackrel{def}{=} & \emptyset \\
\mathsf{dep}(\uparrow T \models v \longleftarrow e) & \stackrel{def}{=} & \{v\} \times \mathsf{var}(e) \\
\mathsf{dep}(\downarrow T \models v \longleftarrow e) & \stackrel{def}{=} & \{v\} \times \mathsf{var}(e) \\
\mathsf{dep}(\mathtt{var}\ v\ P\ \mathtt{end}\ v) & \stackrel{def}{=} & \{(u,w) \in \mathsf{dep}(P) \mid u \neq v \wedge w \neq v\}
\end{array}
$$

Figure 8.2: The dependency function

Proof:

$$
\begin{array}{cl}
 & \mathcal{C}(v \longleftarrow e) \\
= & \{ \text{ by definition } \} \\
 & \surd v = e \\
= & \{ \text{ monotonicity of invariants and premise } \} \\
 & \surd(v = e \ \wedge \ e = f) \\
= & \{ \text{ monotonicity of invariants and premise } \} \\
 & \surd v = f \\
= & \{ \text{ by definition } \} \\
 & \mathcal{C}(v \longleftarrow f)
\end{array}
$$

□

Law AP-2 *Combinational properties and composition*
Combinational equivalences can be propagated over parallel composition.

$$\mathcal{C}(v \longleftarrow e) \parallel Q \Rightarrow Q[e/v] \qquad [\textbf{Law} - \mathcal{C} - \parallel]$$

The proof of this law follows by induction on the structure of Q and the law of duration calculus that: $P(v) \wedge \surd(v = e) \Rightarrow P(e)$.

8.9.2 Delay Properties

Law AP-3 *Monotonicity of weak inertial delays*
Weak inertial delays are refined as the delay decreases.

If $\Delta \geq \delta$ *then*

$$\mathcal{I}^-(v \stackrel{\delta}{\longleftarrow} e) \Rightarrow \mathcal{I}^-(v \stackrel{\Delta}{\longleftarrow} e) \qquad [\textbf{Law} - \textbf{mono}\ \mathcal{I}^-]$$

Proof: An outline of the proof follows:

$$
\begin{array}{ll}
 & \mathcal{I}^-(v \xleftarrow{\delta} e) \\
= & \{ \text{ by definition } \} \\
 & \mathsf{Stable}(\mathsf{var}(e)) \xrightarrow{\delta} \lceil v = \delta \gg e \rceil \\
\Rightarrow & \{ \text{ monotonicity of leads to operator and } \Delta \geq \delta \ \} \\
 & \mathsf{Stable}(\mathsf{var}(e)) \xrightarrow{\Delta} \lceil v = \delta \gg e \rceil \\
\Rightarrow & \{ \text{ definition of leads to operator } \} \\
 & \Box((\mathsf{Stable}(\mathsf{var}(e)) \wedge l = \Delta); l > 0 \Rightarrow (l = \Delta; \lceil v = \delta \gg e \rceil) \\
\Rightarrow & \{ \text{ right hand side implies } e \text{ remains constant } \} \\
 & \Box((\mathsf{Stable}(\mathsf{var}(e)) \wedge l = \Delta); l > 0 \Rightarrow (l = \Delta; \lceil v = \Delta \gg e \rceil) \\
\Rightarrow & \{ \text{ definition of leads to operator } \} \\
 & \mathsf{Stable}(\mathsf{var}(e)) \xrightarrow{\Delta} \lceil v = \Delta \gg e \rceil \\
= & \{ \text{ by definition } \} \\
 & \mathcal{I}^-(v \xleftarrow{\Delta} e)
\end{array}
$$

□

Law AP-4 *Equivalence of delays*
Delay circuits of the same type and which compute equivalent propositions are themselves equivalent.

$$
\begin{array}{l}
\textit{If } e = f \textit{ and } \mathsf{var}(e) = \mathsf{var}(f) \textit{ and } \mathcal{D} \in \{\mathcal{T}, \mathcal{I}, \mathcal{I}^-\} \textit{ then} \\
\quad \mathcal{D}(v \xleftarrow{\delta} e) = \mathcal{D}(v \xleftarrow{\delta} f)
\end{array}
\qquad [\mathbf{Law} - \mathbf{equiv}\mathcal{D}]
$$

The proof of this law follows in a similar fashion to that of Law AP-2.

Law AP-5 *Weak inertial delays in sequence*
Weak inertial delays can be refined by decomposing them into smaller delays in sequence.

$$
\begin{array}{l}
\mathcal{I}^-(z \xleftarrow{\delta} f(in'_1, \ldots in'_n)) \parallel \mathcal{I}^-(in'_i \xleftarrow{\delta'} in_i) \\
\quad \Rightarrow \mathcal{I}^-(z \xleftarrow{\delta+\delta'} f(in_1, \ldots in_n))
\end{array}
\qquad [\mathbf{Law} - \mathcal{I}^- + \mathcal{I}^- \to \mathcal{I}^-]
$$

The proof follows almost identically the one given for Law AP-3, but uses additivity of the leads to operator: $(P \xrightarrow{n_1} Q \wedge Q \xrightarrow{n_2} R) \Rightarrow P \xrightarrow{n_1+n_2} Q$.

8.9.3 Edge Triggers

Law AP-6 *Weak edge trigger refined by strong edge trigger and weak inertial delay*
A weak edge triggered device may be decomposed into two parts: an immediate trigger acting on the input and a weak inertial device to perform the computation.

$$
\begin{array}{l}
\uparrow T \models In' \longleftarrow In \parallel \mathcal{I}^-(v \xleftarrow{\delta} f(in'_1 \ldots in'_n)) \\
\quad \Rightarrow \uparrow T \models^- v \xleftarrow{\delta} f(in_1 \ldots in_n)
\end{array}
\qquad [\mathbf{Law} - \mathbf{weak\ decomp}]
$$

This law follows by reasoning similar to that of Law AP-3.

8.9.4 Composition

Law AP-7 *Commutativity of composition*

$$
P \parallel Q = Q \parallel P \qquad [\mathbf{Law} - \mathbf{comm} \parallel]
$$

Law AP-8 *Associativity of composition*

$$(P \parallel Q) \parallel R = P \parallel (Q \parallel R) \qquad \textbf{[Law} - \textbf{assoc} \parallel\textbf{]}$$

Law AP-9 *Piecewise refinement (monotonicity of composition)*

If $P \Rightarrow Q$ and both $P \parallel R$ and $Q \parallel R$ are defined, then

$$P \parallel R \Rightarrow Q \parallel R \qquad \textbf{[Law} - \textbf{mono} \parallel\textbf{]}$$

All three laws follow immediately from the fact that $\parallel$ is defined as conjunction (which is commutative, associative and monotonic).

8.9.5 Hiding

The laws given in this section simplify statements which use hiding. All the laws follow from the laws of state variable quantification.

Law AP-10 *Renaming hidden variables*
Provided that no name clashes occur, hidden variables may be renamed freely with no consequence on the computation performed.

Provided that v' is not free in P:

$$\texttt{var}\ v\ \ P\ \ \texttt{end}\ v = \texttt{var}\ v'\ \ P[v'/v];\ \texttt{end}\ v'$$

$$\textbf{[Law} - \textbf{rename]}$$

The proof of this law is very similar to that of Law AP-2.

Law AP-11 *Moving hiding outside of properties*
The scope of hidden variables may be increased as long as there is no name clash.

Provided that v is not free in P:

$$P \parallel \texttt{var}\ v\ \ Q\ \ \texttt{end}\ v = \texttt{var}\ v\ \ P \parallel Q\ \ \texttt{end}\ v$$

$$\textbf{[Law} - \textbf{out var/end]}$$

Proof:

$$\begin{array}{ll} & P \parallel \texttt{var}\ v\ \ Q\ \ \texttt{end}\ v \\ = & \{\ \text{by definition}\ \} \\ & P \wedge\ \exists v \cdot Q \\ = & \{\ v \text{ is not free in } P\ \} \\ & \exists v \cdot P \wedge Q \\ = & \{\ \text{by definition}\ \} \\ & \texttt{var}\ v\ \ P \parallel Q\ \ \texttt{end}\ v \end{array}$$

$\square$

Law AP-12 *Monotonicity of hiding*
Refinment may be performed within the hiding operator.

If $P \Rightarrow Q$ then

$$\texttt{var}\ v\ \ P\ \ \texttt{end}\ v \Rightarrow \texttt{var}\ v\ \ Q\ \ \texttt{end}\ v$$

$$\textbf{[Law} - \textbf{mono var/end]}$$

The law follows immediately from monotonicity of quantification of state variables.

Law AP-13 *One point rule*
A hidden variable which is uniquely (and combinationally) determined can be replaced throughout by its value.

$$\texttt{var}\ v\ \ P \parallel \mathcal{C}(v \longleftarrow e)\ \ \texttt{end}\ v \Rightarrow P[e/v] \qquad \textbf{[Law} - \textbf{OPR]}$$

Follows from the application of laws AP-2 and AP-12 and the fact that v is not free in $P[e/v]$.

8.10 Reducing Properties to a Normal Form

If the system complies with certain requirements, we can simplify and reduce certain properties into simpler constituent ones. The following laws can be used to reduce an arbitrary collection of properties into a normal form. The normal form properties may then be implemented as a Verilog program.

8.10.1 The Normal Form

A property is said to be in normal form if it is a valid property and is of the form $\texttt{var}\ v, \ldots z \quad P \quad \texttt{end}\ v, \ldots z$, where P is a composition of the following types of properties:

- Combinational circuits
- Inertial delays
- Immediate positive edge triggered registers

8.10.2 Transport to Inertial Delay

These laws will be used to convert transport delays to inertial delays which are more readily implementable as Verilog code.

Law NF-1 *Equivalence between transport and inertial delay when* $\delta = 1$
Unit transport and inertial delays are interchangable.

$$\mathcal{I}(v \xleftarrow{1} e) = \mathcal{T}(v \xleftarrow{1} e) \qquad [\mathbf{NF} - \mathcal{T}_1 \to \mathcal{I}_1]$$

Proof:

$$\begin{array}{ll}
 & \mathcal{I}(v \xleftarrow{1} e) \\
= & \{ \text{ by definition } \} \\
 & \quad \begin{array}{lll} & \mathsf{Stable}(\mathsf{var}(e)) & \xrightarrow{1} \lceil v = 1 \gg e \rceil \\ \wedge & \neg\mathsf{Stable}(\mathsf{var}(e)) & \xrightarrow{1} \lceil v = 1 \gg v \rceil \end{array} \\
= & \{ \text{ discrete duration calculus } \} \\
 & \quad \begin{array}{lll} & \mathbf{true} & \xrightarrow{1} \lceil v = 1 \gg e \rceil \\ \wedge & \mathbf{false} & \xrightarrow{1} \lceil v = 1 \gg v \rceil \end{array} \\
= & \{ \text{ DC reasoning about leads to operator } \} \\
 & \mathbf{true} \xrightarrow{1} \lceil v = 1 \gg e \rceil \\
= & \{ \text{ DC reasoning } \} \\
 & \surd(v = 1 \gg e) \\
= & \{ \text{ by definition } \} \\
 & \mathcal{T}(v \xleftarrow{1} e)
\end{array}$$

□

Law NF-2 *Decomposition of transport delay*
Transport delay can be split into two delays in sequence such that the sum of the decomposed delays is equal to the sum of the original delay.

$$\begin{array}{ll}
\mathcal{T}(v \xleftarrow{\delta_1+\delta_2} e) = & \texttt{var}\ v' \\
 & \quad \mathcal{T}(v \xleftarrow{\delta_2} v') \parallel \mathcal{T}(v' \xleftarrow{\delta_1} e); \\
 & \texttt{end}\ v'
\end{array}$$

$$[\mathbf{NF} - \mathbf{decomp}\mathcal{T}]$$

This is a consequence of the law of shift: $(n_1 + n_2) \gg P = n_1 \gg (n_2 \gg P)$.

Law NF-3 *Transport delay maintains stability*
The output of a transport delayed stable variable is itself stable.

$$\begin{array}{ll} \textit{If } \mathcal{S}(\mathsf{var}(e), n) \textit{ and } \mathcal{T}(v \xleftarrow{\delta} e) \textit{ then} & \\ \quad \mathcal{S}(v, n) & \qquad [\mathbf{NF} - \mathbf{stab}\mathcal{T}] \end{array}$$

This law follows immediately from the fact that shift maintains stability of state expressions.

These three laws can be reduced to just one:

Law NF-4 *Transport to Inertial delay*
Transport delays may be decomposed into a number of sequential unit inertial delays.

$$\begin{array}{rl} \mathcal{T}(v \xleftarrow{\delta} e) = & \texttt{var } t_1 \ldots t_\delta \\ & \quad \displaystyle\Big\Vert_{i=0}^{\delta} \mathcal{I}(t_i \xleftarrow{1} t_{i+1}) \\ & \texttt{end } t_1 \ldots t_\delta \\ \textit{where } t_0 = v \textit{ and } t_\delta = e & \qquad [\mathbf{NF} - \mathcal{T} \to \mathcal{I}] \end{array}$$

8.10.3 Weak Inertial Delays

The behaviour of positive inertial delays is simulated by a transport delay.

Law NF-5 *Weak Inertial to Transport delay*
Transport delays are a refinement of weak inertial delays.

$$\mathcal{T}(v \xleftarrow{\delta} e) \Rightarrow \mathcal{I}^-(v \xleftarrow{\delta} e) \qquad [\mathbf{NF} - \mathcal{I}^- \to \mathcal{T}]$$

Thus follows from the law: $D \Rightarrow (E \Rightarrow D)$.

8.10.4 Edge Triggered Registers

The laws in this section are used to reduce delayed edge triggered registers into transport delays and immediate acting edge triggered registers. All falling edge triggers will be converted into rising edge sensitive ones.

Law NF-6 *Falling to rising edge triggers (immediate)*
Strong falling edge triggers can be emulated by strong rising edge triggers which act on the negation of the original trigger.

$$\begin{array}{rl} \downarrow T \models v \longleftarrow e = & \texttt{var } T' \\ & \quad \mathcal{C}(T' \longleftarrow \neg T) \parallel \uparrow T' \models v \longleftarrow e \\ & \texttt{end } T' \end{array}$$

$$[\mathbf{NF} - \downarrow \to \uparrow]$$

This is a consequence of law AP-2 and the definition of rising triggers. The next two laws can also be similarly proved.

Law NF-7 *Falling to rising edge triggers (weak)*
Weak falling edge triggers can be emulated by weak rising edge triggers which act on the negation of the original trigger.

$$\begin{array}{rl} \downarrow T \models^- v \xleftarrow{\delta} e = & \texttt{var}\ T' \\ & \quad \mathcal{C}(T' \longleftarrow \neg T) \parallel \uparrow T' \models^- v \xleftarrow{\delta} e \\ & \texttt{end}\ T' \end{array}$$

$$[\textbf{NF} - \textbf{weak}\ \downarrow \rightarrow \uparrow]$$

Law NF-8 *Falling to rising edge triggers (delayed)*
Delayed falling edge triggers can be emulated by delayed rising edge triggers which act on the negation of the original trigger.

$$\begin{array}{rl} \downarrow T \models v \xleftarrow{\delta} e = & \texttt{var}\ T' \\ & \quad \mathcal{C}(T' \longleftarrow \neg T) \parallel \uparrow T' \models v \xleftarrow{\delta} e \\ & \texttt{end}\ T' \end{array}$$

$$[\textbf{NF} - \downarrow \xrightarrow{\delta} \uparrow]$$

Law NF-9 *Weak to strong rising edge triggered devices*
Strong edge triggered devices are a refinement of weak edge triggered devices.

$$\uparrow T \models v \xleftarrow{\delta} e \Rightarrow \uparrow T \models^- v \xleftarrow{\delta} e$$

$$[\textbf{NF} - \textbf{weak-strong}\ \uparrow]$$

This is a direct consequence of conjunction elimination: $(D \wedge E) \Rightarrow D$.

Law NF-10 *Remove delayed triggers*
Delayed triggers can be replaced by immediate triggers acting upon transport delayed inputs and trigger signal.

$$\begin{array}{rl} \uparrow T \models v \xleftarrow{\delta} e = & \texttt{var}\ e', T' \\ & \quad \mathcal{T}(T' \xleftarrow{\delta} T) \\ & \parallel\ \mathcal{T}(e' \xleftarrow{\delta} e) \\ & \parallel\ \uparrow T' \models v \longleftarrow e' \\ & \texttt{end}\ e', T' \end{array}$$

$$[\textbf{NF} - \delta \rightarrow 0]$$

This law follows from monotonicity of invariants and the definition of rising edge triggers.

8.11 Reduction to Normal Form

Given any property satisfying the constraints given in section 8.8, we can now reduce it to normal form, together with a list of system requirements under which the normal form implies the original property. The correctness of the algebraic laws used in the process guarantees this implication.

Using the normal form laws just given, this transformation is rather straightforward. Throughout the following procedure, the laws about composition are used repeatedly.

1. Using the hiding operator laws, all hidden variables can be renamed, if necessary, and moved to the outermost level.
2. Using laws NF-6, NF-7 and NF-8, all falling edge triggers are converted to rising edge ones.
3. Weak edge triggered devices are replaced by strong ones using law NF-9.
4. Using law NF-10 all delayed edge triggered registers are converted into immediate ones (together with transport delays). Any new hiding operators can be move immediately to the topmost level (as done in step 1).
5. Any weak inertial delay properties are converted into their transport delay counterpart by using law NF-5.
6. Law NF-4 is now used to remove all transport delays.

This generates a normal form specification with a list of system requirements.

8.12 Implementation of Properties in Verilog

Law IMPL-1 *Implementation of a combinational circuit*

$$\mathcal{C}(v \longleftarrow e) \sqsubseteq \texttt{assign } v = e \qquad [\mathbf{Impl} - \mathcal{C}]$$

Proof:

$$\begin{array}{ll} & [\![\texttt{assign } v = e]\!] \\ = & \{ \text{ by definition of semantics } \} \\ & \surd v = e \\ \Rightarrow & \{ \text{ by definition of combinational circuit } \} \\ & \mathcal{C}(v \longleftarrow e) \end{array}$$

□

Law IMPL-2 *Implementation of an inertial delay*

$$\mathcal{I}(v \xleftarrow{\delta} e) \sqsubseteq \texttt{assign } v = \#\delta \;\; e \qquad [\mathbf{Impl} - \mathcal{I}]$$

Proof:

$$\begin{array}{ll} & [\![\texttt{assign } v = \#\delta \;\; e]\!] \\ = & \{ \text{ by definition of semantics } \} \\ & \left(\begin{array}{ll} & l < \delta \wedge \lfloor \neg v \rfloor) \\ \vee & (l = \delta \wedge \lceil \neg v \rceil); \mathbf{true} \end{array} \right) \wedge \\ & (\exists \overline{b} \cdot \lceil \mathsf{var}(e) = \overline{b} \rceil) \xrightarrow{\delta} \lceil v = \delta \gg e \rceil \wedge \\ & \neg(\exists \overline{b} \cdot \lceil \mathsf{var}(e) = \overline{b} \rceil) \xrightarrow{\delta} \lceil v = 1 \gg v \rceil \\ \Rightarrow & \{ \wedge \text{ elimination } \} \\ & (\exists \overline{b} \cdot \lceil \mathsf{var}(e) = \overline{b} \rceil) \xrightarrow{\delta} \lceil v = \delta \gg e \rceil \wedge \\ & \neg(\exists \overline{b} \cdot \lceil \mathsf{var}(e) = \overline{b} \rceil) \xrightarrow{\delta} \lceil v = 1 \gg v \rceil \\ \Rightarrow & \{ \text{ definition of } \mathsf{Stable}(W) \ \} \\ & \mathsf{Stable}(\mathsf{var}(e)) \xrightarrow{\delta} \lceil v = \delta \gg e \rceil \wedge \\ & \neg \mathsf{Stable}(\mathsf{var}(e)) \xrightarrow{\delta} \lceil v = 1 \gg v \rceil \\ = & \{ \text{ by definition of inertial delay } \} \\ & \mathcal{I}(v \xleftarrow{\delta} e) \end{array}$$

□

Law IMPL-3 *Implementation of a rising edge triggered register*

$$\uparrow T \models v \longleftarrow e \sqsubseteq \texttt{always @posedge } T \;\; v = e \qquad [\textbf{Impl} - \uparrow T]$$

Proof: Define $LOOP$ as follows:

$$LOOP \stackrel{def}{=} \mu X \cdot \begin{pmatrix} & (\lfloor T \rfloor \;\wedge\; \mathsf{Const}(v)) \\ \vee & (\lfloor T \rfloor ; \lfloor \neg T \rfloor \;\wedge\; \neg \overrightarrow{T} \;\wedge\; \mathsf{Const}(v)) \\ \vee & (\lfloor T \rfloor ; \lceil \neg T \rceil \;\wedge\; \overrightarrow{T} \;\wedge\; \mathsf{Const}(v)) \fatsemi (\lceil\rceil \wedge \overrightarrow{v} = \overrightarrow{e}) \fatsemi X \end{pmatrix}$$

The invariant we will be working towards is:

$$Inv \stackrel{def}{=} \begin{pmatrix} \Box((\neg \overleftarrow{\text{Rise}}(T) \wedge \lceil\rceil) \Rightarrow \overleftarrow{v} = \overrightarrow{v}) \wedge \\ \Box((\overleftarrow{\text{Rise}}(T) \wedge \lceil\rceil) \Rightarrow \overrightarrow{v} = \overrightarrow{e}) \end{pmatrix}$$

The following DC reasoning will be found useful later:

$$\begin{array}{ll} & (\overrightarrow{v} = \overrightarrow{e} \wedge \overleftarrow{\text{Rise}}(T) \wedge \lceil\rceil) \fatsemi (\lfloor T \rfloor \wedge \mathsf{Const}(v)) \\ \Rightarrow & \{ \text{ relational chop and } \lfloor \cdot \rfloor \text{ definition } \} \\ & (\overrightarrow{v} = \overrightarrow{e} \wedge \overleftarrow{\text{Rise}}(T) \wedge \lceil\rceil) \vee \\ & (\overrightarrow{v} = \overrightarrow{e} \wedge \overleftarrow{\text{Rise}}(T) \wedge \lceil\rceil) \fatsemi (\lceil T \rceil \wedge \mathsf{Const}(v)) \\ \Rightarrow & \{ \text{ relational chop definition and DC reasoning } \} \\ & (\overrightarrow{v} = \overrightarrow{e} \wedge \overleftarrow{\text{Rise}}(T) \wedge \lceil\rceil) \vee \\ & (\lceil\lceil v = e \rceil\rceil \wedge \overleftarrow{\text{Rise}}(T)) ; (\mathsf{Const}(v) \wedge \Box(\neg \overleftarrow{\text{Rise}}(T))) \\ \Rightarrow & \{ \text{ always DC clauses } \} \\ & Inv \end{array}$$

Now consider the following proof outline:

$$\begin{array}{ll} & (\overrightarrow{v} = \overrightarrow{e} \wedge \overleftarrow{\text{Rise}}(T) \wedge \lceil\rceil) \fatsemi LOOP \\ \Rightarrow & \{ \text{ recursion unfolding and above DC reasoning } \} \\ & Inv \vee (l > 0 \wedge Inv) ; (\overrightarrow{v} = \overrightarrow{e} \wedge \overleftarrow{\text{Rise}}(T) \wedge \lceil\rceil) \fatsemi LOOP \\ \Rightarrow & \{ \text{ fixed point } \} \\ & \mu X \cdot Inv \vee (l > 0 \wedge Inv) ; X \\ \Rightarrow & \{ \text{ always clauses and recursion } \} \\ & Inv \end{array}$$

Hence, we can deduce that:

$$\begin{array}{ll} & [\![\texttt{always @posedge } T \;\; v = e]\!] \\ = & \{ \text{ by definition of semantics } \} \\ & \mu X \cdot \begin{pmatrix} & (\lfloor T \rfloor \;\wedge\; \mathsf{Const}(v)) \\ \vee & (\lfloor T \rfloor ; \lfloor \neg T \rfloor \;\wedge\; \overrightarrow{T} = \text{false} \;\wedge\; \mathsf{Const}(v)) \\ \vee & (\lfloor T \rfloor ; \lceil \neg T \rceil \;\wedge\; \overrightarrow{T} = \text{true} \;\wedge\; \mathsf{Const}(v)) \fatsemi (\lceil\rceil \wedge \overrightarrow{v} = \overleftarrow{e}) \fatsemi X \end{pmatrix} \\ = & \{ \text{ no concurrent reading and writing and } v \notin \mathsf{var}(e) \ \} \\ & \mu X \cdot \begin{pmatrix} & (\lfloor T \rfloor \;\wedge\; \mathsf{Const}(v)) \\ \vee & (\lfloor T \rfloor ; \lfloor \neg T \rfloor \;\wedge\; \neg \overrightarrow{T} \;\wedge\; \mathsf{Const}(v)) \\ \vee & (\lfloor T \rfloor ; \lceil \neg T \rceil \;\wedge\; \overrightarrow{T} \;\wedge\; \mathsf{Const}(v)) \fatsemi (\lceil\rceil \wedge \overrightarrow{v} = \overrightarrow{e}) \fatsemi X \end{pmatrix} \end{array}$$

$$
\begin{array}{ll}
\Rightarrow & \{ \text{ recursion unfolding } \} \\
& \left(\begin{array}{ll} & (\lfloor T \rfloor \ \wedge \ \mathsf{Const}(v)) \\ \vee & (\lfloor T \rfloor ; \lfloor \neg T \rfloor \ \wedge \ \neg \overrightarrow{T} \ \wedge \ \mathsf{Const}(v)) \\ \vee & (\lfloor T \rfloor ; \lceil \neg T \rceil \ \wedge \ \overrightarrow{T} \ \wedge \ \mathsf{Const}(v)) \fatsemi (\lceil\rceil \wedge \overrightarrow{v} = \overrightarrow{e}) \fatsemi LOOP \end{array} \right) \\
\Rightarrow & \{ \text{ DC reasoning } \} \\
& Inv \vee (Inv ; (\overrightarrow{v} = \overrightarrow{e} \wedge \overleftarrow{\mathrm{Rise}}(T) \wedge \lceil\rceil) \fatsemi LOOP) \\
\Rightarrow & \{ \text{ previous reasoning and always clauses in DC } \} \\
& Inv \\
\Rightarrow & \{ \text{ discrete DC and monotonicity of } \Box \text{ and } ; \} \\
& \Box((\lceil\rceil \wedge \overleftarrow{\mathrm{Rise}}(T)) ; l > 0 \Rightarrow \lceil v = e \rceil ; \mathbf{true}) \ \wedge \\
& \Box((\lceil\rceil \wedge \neg \overleftarrow{\mathrm{Rise}}(T)) ; l > 0 \Rightarrow \lceil v = 1 \gg v \rceil ; \mathbf{true}) \\
= & \{ \text{ by definition of } \longrightarrow \} \\
& \overleftarrow{\mathrm{Rise}}(T) \longrightarrow \lceil v = 0 \gg e \rceil \wedge \\
& \neg \overleftarrow{\mathrm{Rise}}(T) \longrightarrow \lceil v = 1 \gg v \rceil \\
= & \{ \text{ by definition of triggered register with delay 0 } \} \\
& \uparrow T \models v \xleftarrow{0} e \\
= & \{ \text{ by definition of immediate triggered register } \} \\
& \uparrow T \models v \longleftarrow e
\end{array}
$$

□

Law IMPL-4 *Implementation of composition*

If $P_1 \sqsubseteq Q_1$ *and* $P_2 \sqsubseteq Q_2$ *then*

$$P_1 \| P_2 \sqsubseteq Q_1 \| Q_2 \qquad [\mathbf{Impl} - \ \|]$$

Proof:

$$
\begin{array}{ll}
& [\![Q_1 \parallel Q_2]\!] \\
= & \{ \text{ by definition of semantics } \} \\
& [\![Q_1]\!] \wedge [\![Q_2]\!] \\
\Rightarrow & \{ \text{ by premise and monotonicity of } \wedge \} \\
& P_1 \wedge P_2 \\
= & \{ \text{ by definition of } \| \} \\
& P_1 \| P_2
\end{array}
$$

□

8.13 Summary

This chapter shows how one can design a specification language which can be mechanically transformed into Verilog using verifiable laws. The specification language given is not particularly high level, but it is sufficient to demonstrate a number of interesting uses as will be shown in the next chapter.

Chapter 9

Decomposition of Hardware Component Specifications: Examples

This chapter will specify variations on an n-bit adder using the specification language given in the previous chapter. The specifications will be modified and then implemented as Verilog programs using the laws of the specification language.

Three specifications of the adder are given: combinational, delayed and triggered:

- At the simplest level, the whole circuit may be considered as a combinational one. The transformation becomes rather straightforward and requires only the standard propositional calculus proof usually used to prove the correctness of an n-bit adder.
- At a higher, slightly more complex, level the individual components are given an inherent delay.
- Finally, a triggering signal is added to the circuit to set off the adder. A rising edge on the trigger will, after a necessary delay, produce the desired result on the outputs.

The main scope of this example is not simply the implementation of the given specification as a Verilog program. The normal form conversion laws given in chapter 8 allow us to do this automatically. Before the implementation process is carried out, we split up the specification into a number of blocks in predefined formats. In the combinational case, for example, the specification is split up into AND, XOR and OR gate specifications. The eventual Verilog implementation would thus be closer to the chosen hardware components.

9.1 A Combinational n-bit Adder

The first case is a combinational circuit specification. The complete specification, for an n-bit adder can be thus described by:

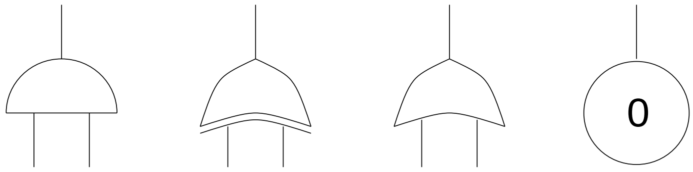


Figure 9.1: The gates used: `AND` $(\wedge)$, `XOR` $(\oplus)$, `OR` $(\vee)$, `GND` respectively

$$\mathsf{C\text{-}NBA\text{-}SPEC}(sout, cout, a, b, n) \stackrel{def}{=} \Big\|_{i=1}^{n} \mathcal{C}(sout_i \longleftarrow (\sum_{j=0}^{n-1} 2^j(a_j + b_j))!i)$$
$$\| \quad \mathcal{C}(cout \longleftarrow (\sum_{j=0}^{n-1} 2^j(a_j + b_j))!n)$$

where $b!i$ is the ith bit of bit-string b.

Note that this may be immediately synthesised into Verilog code by using law [**Impl** $- \mathcal{C}$]. We would, however, prefer to decompose the adder into smaller pieces and perform the transformation into Verilog on the component. We use the standard method to build an n-bit adder from n full adders, each of which is made up of two half adders and combinational gates. The half adders are then built directly from combinational gates. We start by showing that this decomposition of the circuit is in fact a correct one, and then, finally, transform the eventual result into a Verilog program.

9.1.1 Combinational Gates

Figure 9.1 shows the different gates which will be used to build an adder. Their semantics are defined as follows:

$$\begin{aligned}
\texttt{C-AND}(z,a,b) &\stackrel{def}{=} \mathcal{C}(z \longleftarrow a \wedge b) \\
\texttt{C-XOR}(z,a,b) &\stackrel{def}{=} \mathcal{C}(z \longleftarrow a \oplus b) \\
\texttt{C-OR}(z,a,b) &\stackrel{def}{=} \mathcal{C}(z \longleftarrow a \vee b) \\
\texttt{C-GND}(z) &\stackrel{def}{=} \mathcal{C}(z \longleftarrow 0)
\end{aligned}$$

9.1.2 Half Adder

A half adder can now be defined in terms of the combinational gates. Output c is true if and only if the inputs a and b are both true (the one bit sum of a and b leaves a carry). Output s carries the one-bit sum of a and b. Figure 9.2 shows the composed circuit, which is defined as:

$$\texttt{C-HA}(s,c,a,b) \stackrel{def}{=} \texttt{C-XOR}(s,a,b) \,\|\, \texttt{C-AND}(c,a,b)$$

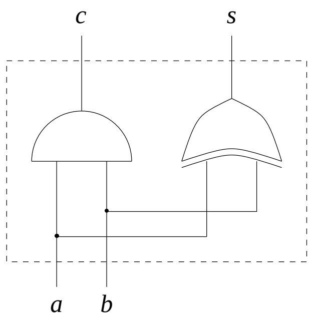


Figure 9.2: The composition of a half adder

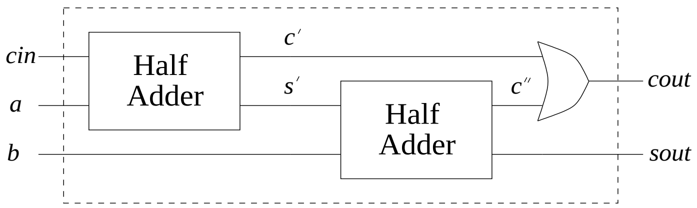


Figure 9.3: Decomposition of s full adder into half adders

9.1.3 Full Adder

A full adder can now be constructed using two half adders and a disjunction. Figure 9.3 shows how the construction is done.

$$\begin{array}{rcl} \texttt{C-FA}(sout, cout, a, b, cin) \stackrel{def}{=} & \texttt{var} & s', c', c'' \\ & & \texttt{C-HA}(s', c', cin, a) \\ & \parallel\!| & \texttt{C-HA}(sout, c'', s', b) \\ & \parallel\!| & \texttt{C-OR}(cout, c', c'') \\ & \texttt{end} & s', c', c'' \end{array}$$

9.1.4 Correctness of the Full Adder

At this stage, we can show that the above definition of a full adder is, in fact, a correct refinement of the following specification:

$$\begin{array}{rcl} \textsf{C-FA-SPEC}(sout, cout, a, b, cin) & \stackrel{def}{=} & \mathcal{C}(cout \longleftarrow (a + b + cin) \ \texttt{div} \ 2) \\ & & \mathcal{C}(sout \longleftarrow (a + b + cin) \ \texttt{mod} \ 2) \end{array}$$

Note that a, b and cin are single bits and hence this definition is not more abstract than the specification of the n-bit adder as given in section 9.1.

The proof is rather straightforward: The definition of $\texttt{C-FA}(sout, cout, a, b, cin)$ is opened up and law [**Law** $- \mathcal{C} - \parallel\!|$] is applied to s', c' and c''. Using just propositional calculus, and [**Law** $-$ **equiv**$\mathcal{C}$] it can then be easily established that:

$$\texttt{C-FA}(sout, cout, a, b, cin) \Rightarrow \textsf{C-FA-SPEC}(sout, cout, a, b, cin)$$

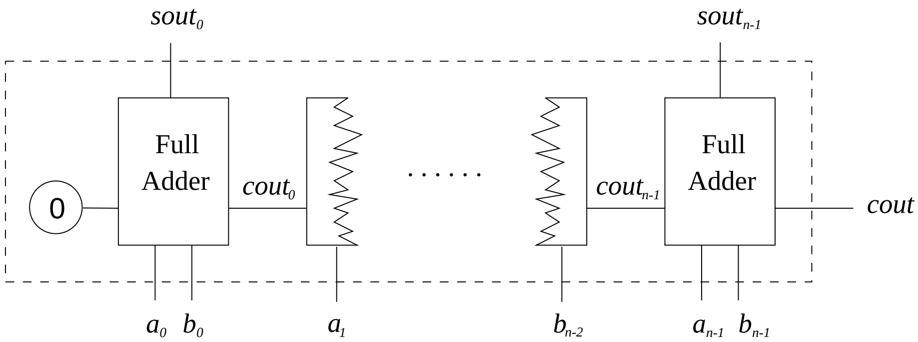


Figure 9.4: Decomposition of an n-bit adder into full adders

An outline of the proof follows:

$$
\begin{array}{ll}
& \texttt{C-FA}(sout, cout, a, b, cin) \\
\Rightarrow & \{ \text{ definition of } \texttt{C-FA} \; \} \\
& \begin{array}{ll}
\texttt{var} & s', c', c'' \\
& \mathcal{C}(s' \longleftarrow cin \oplus a) \\
\| & \mathcal{C}(c' \longleftarrow cin \wedge a) \\
\| & \mathcal{C}(sout \longleftarrow s' \oplus b) \\
\| & \mathcal{C}(c'' \longleftarrow s' \wedge b) \\
\| & \mathcal{C}(cout \longleftarrow c' \vee c'') \\
\texttt{end} & s', c', c''
\end{array} \\
\Rightarrow & \{ \text{ applying } [\mathbf{Law} - \mathbf{OPR}] \; \} \\
& \mathcal{C}(sout \longleftarrow (cin \oplus a) \oplus b) \\
& \mathcal{C}(cout \longleftarrow (cin \wedge a) \vee ((cin \oplus a) \wedge b)) \\
\Rightarrow & \{ \text{ using } [\mathbf{Law} - \mathbf{equiv}\mathcal{C}] \; \} \\
& \mathcal{C}(cout \longleftarrow (a + b + cin) \; \texttt{div} \; 2) \\
& \mathcal{C}(sout \longleftarrow (a + b + cin) \; \texttt{mod} \; 2) \\
\Rightarrow & \{ \text{ definition of } \textsf{C-FA-SPEC} \; \} \\
& \textsf{C-FA-SPEC}(sout, cout, a, b, cin)
\end{array}
$$

9.1.5 n-bit Adder

Finally, we prove, that if we connect n full adders together (as shown in figure 9.4), the resultant composition implies the original specification C-NBA-SPEC.

$$
\texttt{C-NBA}(sout, cout, a, b, n) \stackrel{def}{=} \begin{array}{ll}
\texttt{var} & cin, cout_0, \ldots cout_{n-1} \\
& \texttt{C-NBA}'(sout, cout, a, b, n) \\
\texttt{end} & cin, cout_0, \ldots coutn - 1
\end{array}
$$

where $\mathtt{C\text{-}NBA'}$ is defined recursively as follows:

$$\begin{array}{rcl}
\mathtt{C\text{-}NBA'}(sout, cout, a, b, cin, 1) & \stackrel{def}{=} & \mathtt{GND}(cin) \\
& \| & \mathtt{C\text{-}FA}(sout_0, cout_0, a_0, b_0, cin) \\
\mathtt{C\text{-}NBA'}(sout, cout, a, b, cin, n+1) & \stackrel{def}{=} & \mathtt{C\text{-}NBA}(sout, cout, a, b, cin, n) \\
& \| & \mathtt{C\text{-}FA}(sout_n, cout_n, a_n, b_n, cout_{n-1})
\end{array}$$

Again, the proof is rather straightforward. Simply by using predicate calculus and monotonicity of combinational properties, it can be shown that the decomposition of the n-bit adder is, in fact, correct.

Theorem: $\mathtt{C\text{-}NBA'}(sout, cout, a, b, cin, n) \Rightarrow \textsf{C-NBA}(sout, cout, a, b, cin, n)$

Proof: The proof proceeds by induction on n. A detailed outline of the proof is presented below:

Base case: $n = 1$

$$\begin{array}{cl}
& \mathtt{C\text{-}NBA'}(sout, cout, a, b, cin, 1) \\
= & \{ \text{ by definition of } \mathtt{C\text{-}NBA'} \ \} \\
& \mathtt{GND}(cin) \ \| \ \mathtt{C\text{-}FA}(sout_0, cout_0, a_0, b_0, cin) \\
\Rightarrow & \{ \text{ result from section 9.1.4 } \} \\
& \mathtt{GND}(cin) \ \| \ \textsf{C-FA-SPEC}(sout_0, cout_0, a_0, b_0, cin) \\
= & \{ \text{ by definition } \} \\
& \quad \mathcal{C}(cin \longleftarrow 0) \\
& \| \quad \mathcal{C}(cout_0 \longleftarrow (a_0 + b_0 + cin) \ \mathtt{div} \ 2) \\
& \| \quad \mathcal{C}(sout_0 \longleftarrow (a_0 + b_0 + cin) \ \mathtt{mod} \ 2) \\
\Rightarrow & \{ \text{ using } [\mathbf{Law - equiv}\mathcal{C}] \ \} \\
& \quad \mathcal{C}(cout_0 \longleftarrow (a_0 + b_0 + 0) \ \mathtt{div} \ 2) \\
& \| \quad \mathcal{C}(sout_0 \longleftarrow (a_0 + b_0 + 0) \ \mathtt{mod} \ 2) \\
\Rightarrow & \{ \text{ by definition of } \textsf{C-NBA-SPEC} \ \} \\
& \textsf{C-NBA-SPEC}(sout, cout, a, b, cin, 1)
\end{array}$$

Inductive case: $n = k + 1$

$$\begin{array}{cl}
& \mathtt{C\text{-}NBA'}(sout, cout, a, b, cin, k+1) \\
= & \{ \text{ by definition of } \mathtt{C\text{-}NBA'} \ \} \\
& \quad \mathtt{C\text{-}NBA}(sout, cout, a, b, cin, k) \\
& \| \quad \mathtt{C\text{-}FA}(sout_k, cout_k, a_k, b_k, cout_{k-1}) \\
\Rightarrow & \{ \text{ inductive hypothesis } \} \\
& \quad \textsf{C-NBA-SPEC}(sout, cout, a, b, cin, k) \\
& \| \quad \mathtt{C\text{-}FA}(sout_k, cout_k, a_k, b_k, cout_{k-1}) \\
\Rightarrow & \{ \text{ result from section 9.1.4 } \} \\
& \quad \textsf{C-NBA-SPEC}(sout, cout, a, b, cin, k) \\
& \| \quad \textsf{C-FA-SPEC}(sout_k, cout_k, a_k, b_k, cout_{k-1}) \\
= & \{ \text{ by definition } \} \\
& \displaystyle\mathop{\Big\|}_{i=1}^{k} \quad \mathcal{C}(sout_i \longleftarrow (\textstyle\sum_{j=0}^{k-1} 2^j(a_j + b_j))!i) \\
& \| \quad \mathcal{C}(cout_{k-1} \longleftarrow (\textstyle\sum_{j=0}^{k-1} 2^j(a_j + b_j))!k) \\
& \| \quad \mathcal{C}(cout_k \longleftarrow (a_k + b_k + cout_{k-1}) \ \mathtt{div} \ 2) \\
& \| \quad \mathcal{C}(sout_k \longleftarrow (a_k + b_k + cout_{k-1}) \ \mathtt{mod} \ 2)
\end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ replacing equals for equals } \} \\
& \quad \mathop{\Big\|}_{i=1}^{k} \quad \mathcal{C}(sout_i \longleftarrow (\sum_{j=0}^{k-1} 2^j(a_j + b_j))!i) \\
& \quad \| \quad \mathcal{C}(sout_{k+1} \longleftarrow (\sum_{j=0}^{k} 2^j(a_j + b_j))!k) \\
& \quad \| \quad \mathcal{C}(cout_{k-1} \longleftarrow (\sum_{j=0}^{k-1} 2^j(a_j + b_j))!k) \\
& \quad \| \quad \mathcal{C}(cout_k \longleftarrow (a_k + b_k + cout_{k-1}) \texttt{ div } 2) \\
= & \{ \text{ higher order bits do not affect lower order ones } \} \\
& \quad \mathop{\Big\|}_{i=1}^{k} \quad \mathcal{C}(sout_i \longleftarrow (\sum_{j=0}^{k} 2^j(a_j + b_j))!i) \\
& \quad \| \quad \mathcal{C}(sout_{k+1} \longleftarrow (\sum_{j=0}^{k} 2^j(a_j + b_j))!k) \\
& \quad \| \quad \mathcal{C}(cout_{k-1} \longleftarrow (\sum_{j=0}^{k-1} 2^j(a_j + b_j))!k) \\
& \quad \| \quad \mathcal{C}(cout_k \longleftarrow (a_k + b_k + cout_{k-1}) \texttt{ div } 2) \\
& \{ \text{ generalised parallel composition } \} \\
& \quad \mathop{\Big\|}_{i=1}^{k+1} \quad \mathcal{C}(sout_i \longleftarrow (\sum_{j=0}^{k-1} 2^j(a_j + b_j))!i) \\
& \quad \| \quad \mathcal{C}(cout_{k-1} \longleftarrow (\sum_{j=0}^{k-1} 2^j(a_j + b_j))!k) \\
& \quad \| \quad \mathcal{C}(cout_k \longleftarrow (a_k + b_k + cout_{k-1}) \texttt{ div } 2) \\
\Rightarrow & \{ \text{ replacing equals for equals } \} \\
& \quad \mathop{\Big\|}_{i=1}^{k+1} \quad \mathcal{C}(sout_i \longleftarrow (\sum_{j=0}^{k} 2^j(a_j + b_j))!i) \\
& \quad \| \quad \mathcal{C}(cout_k \longleftarrow (\sum_{j=0}^{k} 2^j(a_j + b_j))!(k+1)) \\
= & \{ \text{ by definition } \} \\
& \textsf{C-NBA-SPEC}(sout, cout, a, b, cin, k+1)
\end{array}
$$

□

9.1.6 Implementation in Verilog

This alternative specification can be automatically implemented as a Verilog program using the normal form method. Again, we stress that the only reason because of which this transformation was not performed on the original specification was the desire to transform the specification into a composition of parts chosen from a limited set of components. This approach leads to enhanced design modularity.

9.2 Delayed Adder

The next step is to introduce delays within the components. Rather than just using a combinational circuit to represent the the components, we consider gates with inherent delays. The specification of the n-bit adder is now:

$$\textsf{D-NBA-SPEC}(sout, cout, a, b, n, \delta) \stackrel{def}{=} \quad \mathcal{I}^-(cout \xleftarrow{\delta} (\sum_{j=0}^{n-1} 2^j (a_j + b_j))!n)$$
$$\big\Vert_{i=1}^{n} \quad \mathcal{I}^-(sout_i \xleftarrow{\delta} (\sum_{j=0}^{n-1} 2^j (a_j + b_j))!i)$$

Note that from the previous section this can be refined to:

$$\begin{array}{lll}
\textsf{D-NBA-SPEC}(sout, cout, a, b, n, \delta) \sqsubseteq & \texttt{var} & sout', cout' \\
 & & \textsf{C-NBA-SPEC}(sout', cout', a, b, n) \\
 & \Vert & \mathcal{I}^-(cout \xleftarrow{\delta} cout') \\
 & \Vert_{i=1}^{n} & \mathcal{I}^-(sout!i \xleftarrow{\delta} sout'!i) \\
 & \texttt{end} & sout', cout
\end{array}$$

However, we prefer to build the adder from delayed gates for a deeper study of delays.

9.2.1 The Logic Gates

A number of logic gates with an inherent delay are defined.

$$\begin{array}{rcl}
\texttt{D-AND}(z, a, b, \delta) & \stackrel{def}{=} & \mathcal{I}^-(z \xleftarrow{\delta} a \wedge b) \\
\texttt{D-XOR}(z, a, b, \delta) & \stackrel{def}{=} & \mathcal{I}^-(z \xleftarrow{\delta} a \oplus b) \\
\texttt{D-OR}(z, a, b, \delta) & \stackrel{def}{=} & \mathcal{I}^-(z \xleftarrow{\delta} a \vee b)
\end{array}$$

9.2.2 Half Adder

A half adder can now be constructed using the gates just defined. To simplify slightly the presentation, we take both gates in the half adder to have the same delay δ.

$$\begin{array}{rll}
\texttt{D-HA}(s, c, a, b, \delta) \stackrel{def}{=} & & \texttt{D-XOR}(s, a, b, \delta) \\
 & \Vert & \texttt{D-AND}(c, a, b, \delta)
\end{array}$$

9.2.3 Full Adder

We can now construct a full adder by using two half adders and an `OR` gate. For simplicity, we take both half adders and the disjunction gate to have the same delay δ.

$$
\begin{array}{rll}
\mathtt{D\text{-}FA}(s, cout, a, b, cin, \delta) \stackrel{def}{=} & \mathtt{var} & s', c', c'' \\
& & \mathtt{D\text{-}HA}(s', c', a, b, \delta) \\
& \parallel & \mathtt{D\text{-}HA}(s, c'', s', cin, \delta) \\
& \parallel & \mathtt{D\text{-}OR}(cout, c', c'', \delta) \\
& \mathtt{end} & s', c', c''
\end{array}
$$

9.2.4 Proof of Correctness

The specification of the full adder is modified for the new situation. Due to the reaction time of the gates, we cannot be as optimistic about the circuit behaviour as we were in the previous case. However, it will be enough for a hardware engineer to be guaranteed that if the inputs remain stable for long enough, the right result will finally appear on the output wires.

$$
\begin{array}{rll}
\textsf{D-FA-SPEC}(sout, cout, a, b, cin, \delta) & \stackrel{def}{=} & \mathcal{I}^-(cout \stackrel{\delta}{\longleftarrow} (a+b+cin) \ \mathtt{div}\ 2) \\
& \parallel & \mathcal{I}^-(sout \stackrel{\delta}{\longleftarrow} (a+b+cin) \ \mathtt{mod}\ 2)
\end{array}
$$

We now claim that:

$$
\textsf{D-FA-SPEC}(sout, cout, a, b, cin, \Delta) \sqsubseteq \mathtt{D\text{-}FA}(sout, cout, a, b, cin, \delta)
$$

provided that $\Delta \geq 3\delta$. This stability requirement is usually easy to fulfill, since we would normally only allow changes on the global signals at a clock signal, whose period would be much larger than the propagation delay within a logic gate.

The proof of this claim is expounded below:

$$
\begin{array}{ll}
& \mathtt{D\text{-}FA}(sout, cout, a, b, cin, \delta) \\
\Rightarrow & \{ \text{ definition of } \mathtt{D\text{-}FA} \ \} \\
& \begin{array}{ll}
\mathtt{var} & s', c', c'' \\
 & \mathcal{I}^-(s' \stackrel{\delta}{\longleftarrow} cin \oplus a) \\
\parallel & \mathcal{I}^-(c' \stackrel{\delta}{\longleftarrow} cin \wedge a) \\
\parallel & \mathcal{I}^-(sout \stackrel{\delta}{\longleftarrow} s' \oplus b) \\
\parallel & \mathcal{I}^-(c'' \stackrel{\delta}{\longleftarrow} s' \wedge b) \\
\parallel & \mathcal{I}^-(cout \stackrel{\delta}{\longleftarrow} c' \vee c'') \\
\mathtt{end} & s', c', c''
\end{array} \\
\Rightarrow & \{ \text{ applying } [\mathbf{Law} - \mathcal{I}^- + \mathcal{I}^- \rightarrow \mathcal{I}^-] \ \} \\
& \begin{array}{ll}
\mathtt{var} & s', c', c'' \\
 & \mathcal{I}^-(sout \stackrel{2\delta}{\longleftarrow} (cin \oplus a) \oplus b) \\
\parallel & \mathcal{I}^-(cout \stackrel{3\delta}{\longleftarrow} (cin \wedge a) \vee ((cin \oplus a) \wedge b)) \\
\mathtt{end} & s', c', c''
\end{array} \\
\Rightarrow & \{ \text{ applying } [\mathbf{Law} - \mathbf{mono}\ \mathcal{I}^-] \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathcal{I}^{-}(sout \xleftarrow{\Delta} (cin \oplus a) \oplus b) \\
 \parallel & \mathcal{I}^{-}(cout \xleftarrow{\Delta} (cin \wedge a) \vee ((cin \oplus a) \wedge b))
\end{array}
$$

$\Rightarrow$ { applying [**Law − equiv** $\mathcal{D}$] }

$$
\begin{array}{ll}
 & \mathcal{I}^{-}(sout \xleftarrow{\Delta} (a+b+c) \ \texttt{mod}\ 2) \\
 \parallel & \mathcal{I}^{-}(cout \xleftarrow{\Delta} (a+b+c) \ \texttt{div}\ 2)
\end{array}
$$

$\Rightarrow$ { by definition of D-FA-SPEC }

D-FA-SPEC$(sout, cout, a, b, cin, \Delta)$

9.2.5 n-bit Delayed Adder

An n-bit adder can now be constructed from these components. The implementation is identical to the one in section 9.1.5 except that delayed full adders are used. If full adders constructed from δ delay half adders are used, the implementation will guarantee the following specification:

$$\textsf{D-NBA-SPEC}(sout, cout, a, b, n, \Delta) \sqsubseteq \texttt{D-NBA}(sout, cout, a, b, n, \delta)$$

provided that the $\Delta \geq 3n\delta$.

The proof is similar to the proof of the correctness of the full adder decomposition.

9.2.6 Implementation in Verilog

By transforming D-NBA into normal form and implementing in Verilog, we automatically have an implementation of D-NBA-SPEC$(sout, cout, a, b, n, \Delta)$. Needless to say, we must ensure that the proof obligations arising from the transformation are satisfied for the non-input wires. This is easily shown to be satisfied if the inputs remain constant for longer than the δ inherent delay of implementation of the n-bit adder. The resultant Verilog program is shown below:

$$
\begin{array}{l}
\texttt{D-NBA-VERILOG}(sout, cout, a, b, \delta, n) \stackrel{def}{=} \\
\qquad \begin{array}{ll}
 & \texttt{assign}\ cin = 0 \\
 \parallel & \texttt{D-FA-VERILOG}(sout_p, cout_0, a_0, b_0, cin) \\
 \parallel & \vdots \\
 \parallel & \texttt{D-FA-VERILOG}(sout_{n-1}, cout, a_{n-1}, b_{n-1}, cout_{n-1})
\end{array}
\end{array}
$$

where the full adder implementation is:

$$
\begin{array}{lll}
\texttt{D-FA-VERILOG}(sout, cout, a, b, cin, \delta) & \stackrel{def}{=} & \\
 & \texttt{var} & cin \\
 & & \texttt{assign}\ s' = \#\delta\ \ a \oplus b \\
 & \parallel & \texttt{assign}\ c' = \#\delta\ \ a \wedge b \\
 & \parallel & \texttt{assign}\ sout = \#\delta\ \ s' \oplus cin \\
 & \parallel & \texttt{assign}\ c'' = \#\delta\ \ s' \wedge cin \\
 & \parallel & \texttt{assign}\ cout = \#\delta\ \ c' \vee c'' \\
 & \texttt{end} & cin
\end{array}
$$

9.3 Triggered Adder

Finally, we add a triggering signal T to the adder. δ time units after a rising edge on the trigger the output represents the sum of the inputs at the time of the trigger.

The specification of this device is rather straightforward:

$$\begin{array}{ll}
\mathsf{T\text{-}NBA\text{-}SPEC}(sout, cout, a, b, \delta, n) \stackrel{def}{=} & \\
\quad \Vert_{i=1}^{n} & \uparrow T \models^{-} sout_i \stackrel{\delta}{\longleftarrow} (\sum_{j=0}^{n-1} 2^j (a_j + b_j))!i \\
\quad \Vert & \uparrow T \models^{-} cout \stackrel{\delta}{\longleftarrow} (\sum_{j=0}^{n-1} 2^j (a_j + b_j))!n
\end{array}$$

9.3.1 Components

We now need another new basic component, a triggered register. Basically, such a register changes its state to the current value of the input whenever a triggering event occurs.

$$\mathtt{T\text{-}REG}(T, z, a) \quad \stackrel{def}{=} \quad \uparrow T \models z \longleftarrow a$$

9.3.2 Decomposition of the Specification

Using the implementation of a delayed n-bit adder, we can now easily obtain a reasonable decomposition of the specification.

$$\begin{array}{lll}
 & \Vert_{i=1}^{n} & \mathtt{T\text{-}REG}(T, a'!i, a!i) \\
 & \Vert_{i=1}^{n} & \mathtt{T\text{-}REG}(T, b'!i, b!i) \\
 & \Vert & \mathtt{D\text{-}NBA}(sout, cout, a', b', n, \delta) \\
\Rightarrow & \multicolumn{2}{l}{\{ \text{ definition of } \mathtt{T\text{-}REG} \text{ and correctness of } \mathtt{D\text{-}NBA} \text{ } \}} \\
 & \Vert_{i=1}^{n} & \uparrow T \models a'_i \longleftarrow a_i \\
 & \Vert_{i=1}^{n} & \uparrow T \models a'_i \longleftarrow a_i \\
 & \Vert_{i=1}^{n} & \mathcal{I}^{-}(sout_i \stackrel{\delta}{\longleftarrow} (\sum_{j=0}^{n-1} 2^j (a'_j + b'_j))!i) \\
 & \Vert & \mathcal{I}^{-}(cout \stackrel{\delta}{\longleftarrow} (\sum_{j=0}^{n-1} 2^j (a'_j + b'_j))!n) \\
\Rightarrow & \multicolumn{2}{l}{\{ \text{ using } [\textbf{Law} - \textbf{weak decomp}] \text{ } \}}
\end{array}$$

$$\begin{array}{ll} \Big\|_{i=1}^{n} & \uparrow T \models^{-} sout_i \xleftarrow{\delta} (\sum_{j=0}^{n-1} 2^j(a_j + b_j))!i \\ \| & \uparrow T \models^{-} cout \xleftarrow{\delta} (\sum_{j=0}^{n-1} 2^j(a_j + b_j))!n \end{array}$$

$\Rightarrow$ { definition of T-NBA-SPEC }

$\mathsf{T\text{-}NBA\text{-}SPEC}(sout, cout, a, b, \delta, n)$

9.3.3 Implementation

The specification, which we have just transformed into logical blocks can now be automatically translated into Verilog:

$$\mathsf{T\text{-}NBA\text{-}SPEC}(sout, cout, a, b, \delta, n) \sqsubseteq \mathtt{T\text{-}NBA\text{-}VERILOG}(sout, cout, a, b, \delta, n)$$

where $\mathtt{T\text{-}NBA\text{-}VERILOG}(sout, cout, a, b, \delta, n)$ is the following Verilog program:

$$\begin{array}{lll} \mathtt{T\text{-}NBA\text{-}VERILOG}(sout, cout, a, b, \delta, n) & \stackrel{def}{=} & \mathtt{always\ @posedge}\ T\, a' = a \\ & \| & \mathtt{always\ @posedge}\ T\, b' = b \\ & \| & \mathtt{D\text{-}NBA\text{-}VERILOG}(sout, cout, a, b, \delta, n) \end{array}$$

9.4 Conclusions

This chapter shows how the automatic normal form transformation from circuit specifications into Verilog does not need to be done immediately on the specification. System modularity is achieved by using only a limited number of components. The algebraic laws which are used, are usually sufficient to combine a number of different delayed circuit properties into a single one. Hence, the n-bit adder can be just as easily included in another design as the gates were used in the full adder or the full adder in the n-bit adder. Since the laws are always applied in the same fashion, this process may be (at least, partially) mechanically done, which reduces tedious work and human errors. A similar approach to triggered components may be used to decompose the n-bit triggered adder into triggered full adders and the same approach would then be applicable to design a circuit which uses n-bit adders.

The main restriction of the specification language which may seem to be, at first sight, far too restrictive is the fact that combinational circuits with feedback are not allowed. This has been done so as to avoid certain problems such as short circuits (for example $\mathcal{C}(v \longleftarrow \neg v)$). On the other hand, one may argue that feedback is essential to build memory devices and is thus desirable. Various approaches have been proposed in the literature to deal with this problem [Hoa90, Win86, Her88], most of which use sophisticated models to predict such behaviour. With these models it is usually then possible to verify the behaviour of a small number of devices which, when combined together under certain restrictions (not unlike the ones placed in our language), their behaviour can be described using a simpler description. The approach taken here was to introduce memory devices as primitives, thus bypassing the problem altogether.

However note that we cannot decompose the memory devices any further within our model.

Thanks to the restricted specification language, the transformation of circuit properties into separate components can be handled rather easily at the specification level. This allows the generation of a Verilog program which uses only certain components. An alternative approach would be to have a number of laws to transform a Verilog program into a better structured one. Due to the generality of Verilog programs, this would have been much more difficult than the approach taken here.

Part IV

The use of the formal semantics of language is not restricted to proving programs correct with respect to a specification. In this final part we consider two other aspects — we prove the correctness of a compilation procedure from Verilog to hardware and show that a simple simulation semantics is equivalent to the denotational semantics already defined.

Chapter 10

A Hardware Compiler for Verilog

10.1 Introduction

Verilog instructions can be rather neatly classified into two types: imperative programming style instructions such as sequential composition and loops, and more hardware oriented instructions such as continuous assignments. Design of a product usually starts by writing a behavioural module which solves the problem. This is, however, not directly implementable as hardware and has to be refined to produce modules which can be synthesised to hardware. Refinement is usually a rather *ad hoc* procedure where parts of the solution are proposed and their output compared to that of the behavioural description via simulation. The behavioural program is thus taken to be the specification of the product.

Various problems can be identified in this refinement process.

- The base case: the behavioural description of the problem is not guaranteed to be correct. The 'correctness' of the 'specification' (with respect to the desired product) is usually checked via a number of simulation runs on different inputs. This process is dangerous enough in a sequential programming environment, let alone in a language like Verilog, where new dangers introduced by parallel interleaving and the nature of most hardware products to continuously sample the inputs (as opposed to being given a number of inputs at the beginning of the run of a program). One may argue that a mathematical description of the requirements still suffers from the same problem. However, mathematics is a far more abstract and powerful means of describing expectations and is thus considerably less susceptible to this kind of error.

- The refinement steps: As already noted elsewhere, simulation can only identify the presence of errors and not their absence. Nothing short of exhaustive simulation can guarantee correctness.

The first problem can be solved by formally verifying the correctness of an imperative behavioural description with respect to a more abstract mathematical specification. The second problem may also be solved in a similar fashion. Indeed, one may choose to verify directly the correctness of the final product. However, the closer we get to the lowest level implementation, the more difficult and tedious the proofs become. This problem can be solved in a different fashion: by formally proving the correctness of a compilation process from imperative programs down to hardware-like programs.

In the approach presented here, the compilation procedure performs no preprocessing on the input program. This contrasts with commercial synthesis tools [Pal96, SSMT93], which perform certain transformations to help synthesize code which would otherwise be impossible to convert to hardware in a meaningful manner. For example, consider the following program:

```
always @clk v=¬v; w=v; v=¬v;
```

Clearly, unless `v` keeps the temporary value (the negation of its original value) for an amount of time, `w` could never manage to read that value. However, on the other hand, if the spike on `v` take any time at all, it may violate timing constraints satisfied by the above code portion. The solution is to collapse the assignments into a single parallel assignment:

```
always @clk v,w=v,¬v;
```

Note that in our case, this kind of transformations would have to be performed manually, before the compilation procedure is performed.

The approach used here is very similar to [HJ94, Pag93, May90]. We compare our work with their approach in more detail in section 10.5.4.

10.2 Triggered Imperative Programs

We will not be dealing with all programs, but only ones which are triggered by a start signal and upon termination issue a finish signal. The environment is assumed not to interfere with the program by issuing a further start signal before the program has terminated.

Triggered programs can be constructed from general imperative programs:

$$\psi_s^f(P) \quad \stackrel{def}{=} \quad \texttt{forever}\,(s?;\ P;\ f!)$$

At the topmost level, these programs also ensure that the termination signal is initialised to zero:

$$i\psi_s^f(P) \quad \stackrel{def}{=} \quad \texttt{initial}\ f=0;\ \psi_s^f(P)$$

The environment constraint for two signals s and f may now be easily expressed:

$$\epsilon_s^f \quad \sqsupseteq \quad \texttt{forever}\ \ (\ s!;\ \#1;\ f?;\Delta_0)$$

where Δ_0 is a statement which, once triggered, waits for an arbitrary length of time (possibly zero or infinite) before allowing execution to resume.

$$[\![\Delta_0]\!]_W(D) \quad \stackrel{def}{=} \quad \mathsf{Const}(W) \vee \mathsf{Const}(W) \mathbin{;} D$$

The unit delay ensures that if a start signal is sent immediately upon receiving a finish signal, we do not interpret the old finish signal as another finish signal.

Now, we ensure that if we start off with a non-zero delay, the start signal is initially off:

$$i\epsilon_s^f \quad \sqsupseteq \quad ((\neg s^\top; \Delta) \sqcap \texttt{skip})\ ;\ \epsilon_s^f$$

Δ is a statement almost identical to Δ_0 but which, once triggered, waits for a non-zero arbitrary length of time (possibly infinite) before allowing execution to resume.

$P \sqsupseteq Q$, read as 'P is a refinement of Q,' is defined as $P \Rightarrow Q$.

$$[\![\Delta]\!]_W(D) \quad \stackrel{def}{=} \quad \mathsf{Const}(W) \vee (l > 0 \wedge \mathsf{Const}(W)) \mathbin{;} D$$

Δ obeys a number of laws which we will find useful:

$\texttt{skip}$ and delay: $\Delta_0 = \texttt{skip} \sqcap \Delta$

$\mathsf{dur}(P)$ and delay: If $\mathsf{dur}(P)$, then $\Delta \sqsubseteq P$[1].

Signals and delay: If a variable is a *signal* (recall definition of signals from chapter 6), then its behaviour is a refinement of:

$$(\texttt{skip} \sqcap \neg s^\top; \Delta); \texttt{forever}\ s!; \Delta$$

The results which follow usually state a refinement which holds if a particular environment condition holds. Hence, these are of the form:

$$\text{environment condition} \Rightarrow (P \Rightarrow Q)$$

To avoid confusion with nested implications, we define the conditional refinement $i\epsilon_s^f \vdash P \sqsubseteq Q$ as follows:

$$i\epsilon_s^f \vdash P \sqsubseteq Q \quad \stackrel{def}{=} \quad \vdash i\epsilon_s^f \Rightarrow (Q \Rightarrow P)$$

We can now use this to specify equality of processes under a particular environment condition:

$$i\epsilon_s^f \vdash P = Q \quad \stackrel{def}{=} \quad (i\epsilon_s^f \vdash P \sqsubseteq Q) \wedge (i\epsilon_s^f \vdash Q \sqsubseteq P)$$

The following proofs assume that all programs (and sub-programs) satisfy $\mathsf{dur}(P)$ (hence programs take time to execute). Note that if P and Q satisfy this constraint, so do $P; Q$, $P \lhd b \rhd Q$ and $P * b$.

Since we will be replacing sequential composition by parallel composition, writing to a variable and then reading immediately after can cause problems. We thus add the constraint that programs do not read or write as soon as they a executed ($P = Q; R$ such that Q does not read or write data. Again, note that if all primitive programs satisfy this condition, so do programs constructed using sequential composition, conditionals and loops.

[1] This has a similar type problem as we had with the law $P \sqsubseteq P \parallel Q$. A similar approach, using **chaos** can, however, resolve the problem.

10.3 The Main Results

10.3.1 Sequential Composition

Theorem 1.1: Sequential composition can be thus decomposed:

$$i\epsilon_s^f \quad \vdash \quad i\psi_s^f(P;Q) \sqsubseteq i\psi_s^m(P') \parallel i\psi_m^f(Q') \parallel \mathsf{Merge}$$

where for any program P, we will use P' to represent $P[v_P/v]$. Merge has been defined in section 6.12.

Proof: First note the following result:

$$\begin{array}{ll}
 & i\epsilon_s^f \parallel \psi_s^f(P;Q) \\
= & \{ \text{ communication laws } \} \\
 & (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P; Q; f!; \Delta_0; (\epsilon_s^f \parallel \psi_s^f(P;Q)) \\
= & \{ \text{ by law of } \Delta_0 \text{ } \} \\
 & (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P; Q; f!; (\neg s^\top; \Delta \sqcap \texttt{skip}); (\epsilon_s^f \parallel \psi_s^f(P;Q)) \\
= & \{ \text{ by definition of } i\epsilon \text{ } \} \\
 & (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P; Q; f!; (i\epsilon_s^f \parallel \psi_s^f(P;Q)) \\
= & \{ \text{ definition of } \texttt{forever} \text{ } \} \\
 & \texttt{forever } (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P; Q; f! \\
\sqsubseteq & \{ \text{ new signal introduction and laws of signals } \} \\
 & \texttt{forever } \neg m^\top; (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P; m!; Q; f!
\end{array}$$

Now consider the other side of the refinement:

$$\begin{array}{ll}
 & i\epsilon_s^f \parallel i\psi_s^m(P') \parallel \psi_m^f(Q') \\
= & \{ \text{ communication laws } \} \\
 & \neg m^\top; (\neg s^\top; \Delta \sqcap \texttt{skip}); P'; m!; Q'; f!; \neg m^\top; \Delta_0; (\epsilon_s^f \parallel \psi_s^m(P') \parallel \psi_m^f(Q')) \\
= & \{ \text{ definition of } i\epsilon,\ i\psi \text{ and law of } \Delta_0 \text{ } \} \\
 & \neg m^\top; (\neg s^\top; \Delta \sqcap \texttt{skip}); P'; m!; Q'; f!; (i\epsilon_s^f \parallel i\psi_s^m(P') \parallel \psi_m^f(Q')) \\
= & \{ \text{ definition of } \texttt{forever} \text{ } \} \\
 & \texttt{forever } \neg m^\top; (\neg s^\top; \Delta \sqcap \texttt{skip}); P'; m!; Q'; f!
\end{array}$$

We can now prove the desired refinement:

$$\begin{array}{ll}
 & i\epsilon_s^f \parallel i\psi_s^f(P;Q) \\
\sqsubseteq & \{ \text{ definition of } i\psi \text{ and proved inequality } \} \\
 & \neg f^\top; \texttt{forever } \neg m^\top; (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P; m!; Q; f! \\
= & \{ \text{ laws of merge from section 6.12 } \} \\
 & \mathsf{Merge} \parallel (\neg f^\top; \texttt{forever } \neg m^\top; (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P'; m!; Q'; f!) \\
= & \{ \text{ above claim } \} \\
 & \mathsf{Merge} \parallel (\neg f^\top; (i\epsilon_s^f \parallel i\psi_s^m(P') \parallel \psi_m^f(Q')) \\
= & \{ \text{ definition of } i\psi \text{ and associativity of } \parallel \text{ } \} \\
 & \mathsf{Merge} \parallel i\epsilon_s^f \parallel i\psi_s^m(P') \parallel i\psi_m^f(Q')
\end{array}$$

□

10.3.2 Conditional

Theorem 1.2: Conditional statements can be thus decomposed:

$$\epsilon_s^f \quad \vdash \quad \psi_s^f(P \lhd b \rhd Q) \sqsubseteq \psi_{s_P}^{f_P}(P') \parallel \psi_{s_Q}^{f_Q}(Q') \parallel \mathsf{Merge} \parallel \mathcal{I}\text{nterface}$$

where

$$\begin{array}{rlll} \mathcal{I}\text{nterface} = & \texttt{assign} & s_P = s \wedge b & \parallel \\ & \texttt{assign} & s_Q = s \wedge \neg b & \parallel \\ & \texttt{assign} & f = f_P \vee f_Q & \end{array}$$

Proof: Again the proof proceeds by first proving two properties and then combining the results to complete the theorem.

Result 1

$$\begin{array}{cl} & i\epsilon_s^f \parallel \psi_{s_P}^{f_P}(P') \parallel \psi_{s_Q}^{f_Q}(Q') \parallel \mathcal{I}\text{nterface} \\ = & \{ \text{ communication laws } \} \\ & (\neg s^\top; \neg s_P^\top; \neg s_Q^\top; \Delta \sqcap \texttt{skip}); s!; \\ & \qquad (f?; \epsilon_s^f \parallel \psi_{s_P}^{f_P}(P') \parallel \psi_{s_Q}^{f_Q}(Q') \parallel \mathcal{I}\text{nterface}) \\ = & \{ \text{ communication laws } \} \\ & (\neg s^\top; \neg s_P^\top; \neg s_Q^\top; \Delta \sqcap \texttt{skip}); s!; (s_P!; P'; f_P!) \lhd b \rhd (s_Q!; Q'; f_Q!); \\ & \qquad (\neg s^\top; \Delta \sqcap \texttt{skip}); (\epsilon_s^f \parallel \psi_{s_P}^{f_P}(P') \parallel \psi_{s_Q}^{f_Q}(Q') \parallel \mathcal{I}\text{nterface}) \\ = & \{ \text{ definition of loops } \} \\ & \texttt{forever} \;\; (\neg s^\top; \neg s_P^\top; \neg s_Q^\top; \Delta \sqcap \texttt{skip}); s!; (s_P!; P'; f_P!) \lhd b \rhd (s_Q!; Q'; f_Q!) \\ \sqsupseteq & \{ \text{ variable hiding } \} \\ & \texttt{forever} \;\; (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P' \lhd b \rhd Q'; f! \end{array}$$

Result 2: This result is just stated. The proof can be rather easily constructed using the communication laws.

$$i\epsilon_s^f \parallel \psi_s^f(P \lhd b \rhd Q) \quad = \quad \texttt{forever} \;\; (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P \lhd b \rhd Q; f!$$

Finally, we combine the two results to complete the desired proof.

$$\begin{array}{cl} & i\epsilon_s^f \parallel i\psi_s^f(P \lhd b \rhd Q) \\ = & \{ \text{ result 2 } \} \\ & (\neg f^\top; \Delta \sqcap \texttt{skip}); \texttt{forever} \;\; (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P \lhd b \rhd Q; f! \\ \sqsubseteq & \{ \text{ merge laws } \} \\ & \mathsf{Merge} \parallel (\neg f^\top; \Delta \sqcap \texttt{skip}); \texttt{forever} \;\; (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P' \lhd b \rhd Q'; f! \\ \sqsubseteq & \{ \text{ result 1 } \} \\ & (\neg f^\top; \Delta \sqcap \texttt{skip}); (i\epsilon_s^f \parallel \psi_{s_P}^{f_P}(P') \parallel \psi_{s_Q}^{f_Q}(Q') \parallel \mathcal{I}\text{nterface} \parallel \mathsf{Merge}) \\ \sqsubseteq & \{ \text{ definition of } i\epsilon \text{ } \} \\ & i\epsilon_s^f \parallel i\psi_{s_P}^{f_P}(P') \parallel i\psi_{s_Q}^{f_Q}(Q') \parallel \mathcal{I}\text{nterface} \parallel \mathsf{Merge} \end{array}$$

□

10.3.3 Loops

Theorem 1.3: Loops can be decomposed into their constituent parts.

$$\epsilon_s^f \quad \vdash \quad \psi_s^f(P * b) \sqsubseteq \psi_{s_P}^{f_P}(P) \parallel \mathcal{I}\text{nterface}$$

where

$$\begin{array}{rll} \mathcal{I}\text{nterface} = & \texttt{assign} & s_P = s \vee (f_P \wedge b) \quad \parallel \\ & \texttt{assign} & f = f_P \wedge \neg b \end{array}$$

Proof: First of all, we note that:

$$\begin{array}{rl} & \neg f^\top; s_P!; (\psi_{s_P}^{f_P}(P) \parallel f?; \Delta_0; \epsilon_s^f \parallel \mathcal{I}\text{nterface}) \\ = & \neg f^\top; (s_P!; P; f_P!) * b; f!; (i\epsilon_s^f \parallel \psi_{s_P}^{f_P}(P) \parallel \mathcal{I}\text{nterface}) \end{array}$$

The outline of the proof is given below. Let us refer to the left hand side of the equality as α:

$$\begin{array}{rl} & \alpha \\ = & \neg f^\top; s_P!; (\psi_{s_P}^{f_P}(P) \parallel f?; \Delta_0; \epsilon_s^f \parallel \mathcal{I}\text{nterface}) \\ = & \{ \text{ laws of communication and loops } \} \\ & \neg f^\top; s_P!; P; f_P!; \alpha \lhd b \rhd (f!; (i\epsilon_s^f \parallel \psi_{s_P}^{f_P}(P) \parallel \mathcal{I}\text{nterface})) \\ = & \{ \text{ laws of loops } \} \\ & \neg f^\top; (s_P!; P; f_P!) * b; f!; (i\epsilon_s^f \parallel \psi_{s_P}^{f_P}(P) \parallel \mathcal{I}\text{nterface}) \end{array}$$

This can now be used in the main proof of the theorem. The theorem is proved by showing that:

$$i\epsilon_s^f \parallel \psi_s^f(P * b) \quad \sqsubseteq \quad i\epsilon_s^f \parallel \psi_{s_P}^{f_P} \parallel \mathcal{I}\text{nterface}$$

The desired result would then follow immediately.

This inequality is proved as follows:

$$\begin{array}{rl} & \{ \text{ RHS (right hand side of inequality) } \} \\ = & i\epsilon_s^f \parallel \psi_{s_P}^{f_P} \parallel \mathcal{I}\text{nterface} \\ = & \{ \text{ by communication laws } \} \\ & (\neg s^\top; \neg s_P^\top; \Delta \ \sqcap \texttt{skip}); s!; s_P!; P; f_P!; \alpha \lhd b \rhd (f!; \text{RHS}) \\ = & \{ \text{ by the result just given } \} \\ & (\neg s^\top; \neg s_P^\top; \Delta \ \sqcap \texttt{skip}); s!; s_P!; P; f_P!; \\ & \qquad (\neg f^\top; (s_P!; P; f_P!) * b; f!; \text{RHS}) \lhd b \rhd (f!; \text{RHS}) \\ = & \{ \text{ by distribution of sequential composition in conditionals } \} \\ & (\neg s^\top; \neg s_P^\top; \Delta \ \sqcap \texttt{skip}); s!; s_P!; P; f_P!; ((s_P!; P; f_P!) * b; \lhd b \rhd \texttt{skip}); f!; \text{RHS} \\ = & \{ \text{ law of loops } \} \end{array}$$

$$
\begin{array}{ll}
& (\neg s^{\top}; \neg s_P^{\top}; \Delta \ \sqcap \texttt{skip}); s!; (s_P!; P; f_P!) * b; f!; \text{RHS} \\
= & \{ \text{ definition of forever loops } \} \\
& \texttt{forever} \ \ (\neg s^{\top}; \neg s_P^{\top}; \Delta \ \sqcap \texttt{skip}); s!; (s_P!; P; f_P!) * b; f! \\
\sqsupseteq & \{ \text{ hiding variables } \} \\
& \texttt{forever} \ \ (\neg s^{\top}; \Delta \ \sqcap \texttt{skip}); s!; P * b; f!
\end{array}
$$

But, on the other hand:

$$
\begin{array}{ll}
& i\epsilon_s^f \parallel \psi_s^f(P * b) \\
= & \{ \text{ communication laws } \} \\
& (\neg s^{\top}; \Delta \ \sqcap \ \texttt{skip}); s!; P * b; f!; (i\epsilon_s^f \parallel \psi_s^f(P * b)) \\
= & \{ \text{ definition of loops } \} \\
& \texttt{forever} \ (\neg s^{\top}; \Delta \ \sqcap \ \texttt{skip}); s!; P * b; f!
\end{array}
$$

$\square$

10.4 Compilation

Using these refinements, we can now define a compilation process:

$$
\begin{array}{rcl}
\Psi_s^f(P; Q) & \stackrel{def}{=} & \Psi_s^m(P) \parallel \Psi_m^f(Q) \parallel \mathsf{Merge} \\
\Psi_s^f(P \triangleleft b \triangleright Q) & \stackrel{def}{=} & \Psi_{s_P}^{f_P}(P') \parallel \Psi_{s_Q}^{f_Q}(Q') \parallel \mathsf{Merge} \parallel \mathcal{I}\text{nterface}_C \\
\Psi_s^f(P * b) & \stackrel{def}{=} & \Psi_{s_P}^{f_P}(P) \parallel \mathcal{I}\text{nterface}_L \\
\Psi_s^f(P) & \stackrel{def}{=} & \psi_s^f(P) \text{ otherwise}
\end{array}
$$

We know that the individual steps of the compilation process are correct. However, it is not yet clear whether the environment conditions can flow through the compilation. In other words, when we split sequential composition into two parts, we showed that the environment conditions for both processes was guaranteed. But is this still the case if we refine the processes further?

Lemma 2.1: Provided that s and f are signals, if $\int f \leq \int s \gg 1$ and $\int f \leq \int s \leq \int f + 0.5$ are valid duration formula, then so is $i\epsilon_s^f$.

Proof: Since the inequality holds for all prefix time intervals, and s and f are both signals, we can use duration calculus reasoning to conclude that:

$$\square((\lceil s \rceil \wedge l = 0.5); \mathbf{true}; (\lceil s \rceil \wedge l = 0.5) \Rightarrow l = 1; \mathbf{true}; \lceil f \rceil; \mathbf{true})$$

This allows us to deduce that $s!; \Delta; s!; \Delta \Rightarrow s!; \#1; f?; \Delta_0; s!; \Delta$.

But s is a signal, and hence satisfies $(\neg s^{\top}; \Delta \sqcap \texttt{skip}); \texttt{forever}\ s!; \Delta$.

$$
\begin{array}{ll}
& \texttt{forever}\ s!; \Delta \\
= & \{ \text{ definition of } \texttt{forever} \text{ loops } \} \\
& s!; \Delta; s!; \Delta; \texttt{forever}\ s!; \Delta \\
\Rightarrow & \{ \text{ by implication just given } \} \\
& s!; \#1; f?; \Delta_0; s!; \Delta; \texttt{forever}\ s!; \Delta
\end{array}
$$

$$\begin{array}{cl} = & \{ \text{ definition of } \texttt{forever} \text{ loops } \} \\ & s!; \#1; f?; \Delta_0; \texttt{forever}\ s!; \Delta \\ \Rightarrow & \{ \text{ definition of } \texttt{forever} \text{ loops } \} \\ & \texttt{forever}\ s!; \#1; f?; \Delta_0 \end{array}$$

Hence, from the fact that s is a signal, we can conclude the desired result:

$$\begin{array}{cl} & (\neg s^\top; \Delta \sqcap \texttt{skip}); \texttt{forever}\ s!; \Delta \\ \Rightarrow & (\neg s^\top; \Delta \sqcap \texttt{skip}); \texttt{forever}\ s!; \#1; f?; \Delta_0 \\ = & i\epsilon_s^f \end{array}$$

□

Lemma 2.2: $i\epsilon_s^f \Rightarrow \int s \leq \int f + 0.5$.

Proof: The proof of this lemma follows by induction on the number of times that the environment loop is performed. We first note that $i\epsilon_s^f$ can be rewritten as:

$$(\neg s^\top; \Delta \sqcap \texttt{skip}); s!; \#1; \texttt{forever}\ (f?; \Delta_0; s!; \#1)$$

Using the law $f? \ = \ f^\top \sqcap (\neg f^\top; \#1; f?)$ and distributivity of non-deterministic choice, it can be shown that this program is equivalent to:

$$(\neg s^\top; \Delta \sqcap \texttt{skip}); s!; \#1; \texttt{forever} \left(\begin{array}{ll} & f^\top; s!; \#1 \\ \sqcap & f^\top; \Delta; s!; \#1 \\ \sqcap & \neg f^\top; \#1; f?; \Delta_0; s!; \#1 \end{array} \right)$$

Using the laws of loops this is equivalent to:

$$(\neg s^\top; \Delta \sqcap \texttt{skip}); s!; \#1; \texttt{forever} \left(\begin{array}{ll} & f^\top; s!; \#1 \\ \sqcap & \neg s^\top; f^\top; \Delta; s!; \#1 \\ \sqcap & \neg s^\top; \neg f^\top; \#1; f?; \Delta_0; s!; \#1 \end{array} \right)$$

The semantic interpretation of this program takes the form:

$$P' \vee \exists n : \mathbb{N} \cdot P; Q^n; Q'$$

where P' corresponds to the partial execution of $(\neg s^\top; \Delta \sqcap \texttt{skip}); s!; \#1$, and P to its full execution. Similarly, Q' and Q correspond to the partial and complete execution of the loop body.

$$\begin{array}{rcl} P & \Rightarrow & \lfloor \neg s \rfloor; (\lceil s \rceil \wedge l = 0.5); \lfloor \neg s \rfloor \\ Q & \Rightarrow & (\mathbf{true}; \lceil f \rceil; \mathbf{true} \ \wedge \ \lfloor \neg s \rfloor; (\lceil s \rceil \wedge l = 0.5)); \lfloor \neg s \rfloor \\ P' & \Rightarrow & \lfloor \neg s \rfloor \vee \lfloor \neg s \rfloor; (\lceil s \rceil \wedge l = 0.5); \lfloor \neg s \rfloor \\ Q' & \Rightarrow & \lfloor \neg s \rfloor \vee (\lfloor \neg s \rfloor; (\lceil s \rceil \wedge l = 0.5); \lfloor \neg s \rfloor \ \wedge \ \mathbf{true}; \lceil f \rceil; \mathbf{true}) \end{array}$$

Since $P' \Rightarrow \int s = 0.5$, it immediately follows that $P' \Rightarrow \int s \leq \int f + 0.5$.

We can also show, by induction on n, that $P; Q^n$ implies this invariant. An outline of the inductive case is given below:

$$\begin{array}{cl} & P; Q^{n+1} \\ = & P; Q^n; Q \\ \Rightarrow & (\int s \leq \int f + 0.5); Q \\ \Rightarrow & (\int s \leq \int f + 0.5); (\int s = 0.5 \wedge \int f \geq 0.5) \\ \Rightarrow & \int s \leq \int f + 0.5 \end{array}$$

Finally, we can use this result to show $P; Q^n; Q' \Rightarrow \int s \leq \int f + 0.5$.

$$
\begin{array}{ll}
 & P; Q^n; Q' \\
\Rightarrow & (\int s \leq \int f + 0.5); Q' \\
\Rightarrow & (\int s \leq \int f + 0.5); \int s = 0 \vee \\
 & (\int s \leq \int f + 0.5); (\int s = 0.5 \wedge \int f \geq 0.5) \\
\Rightarrow & \int s \leq \int f + 0.5
\end{array}
$$

This completes the required proof.

$\square$

Lemma 2.3: Provided that $\mathsf{dur}(P)$:

$$i\psi_s^f(P) \Rightarrow \int f \leq \int s \ \wedge \ \int f \leq \int s \gg 1$$

Proof: Note that $i\psi_s^f(P)$ is a refinement of $(\neg f^\top; \Delta \sqcap \mathtt{skip}); \mathtt{forever}\ f!; s?; \Delta$ which is almost identical to $i\epsilon_f^s$.

The proof follows almost identically to that of lemma 2.2 except that, unlike the environment condition, $i\psi_s^f(P)$ cannot signal on f as soon as it receives a signal on s (since P must take some time to execute). This allows us to gain that extra 0.5 time unit.

$\square$

Lemma 2.4: The environment conditions follow along the compilation process:

$$\Psi_s^f(P) \Rightarrow \int f \leq \int s$$

Proof: The proof uses structural induction on the program:

In the base case, P cannot be decomposed any further, and hence $\Psi_s^f(P) = \psi_s^f(P)$. Therefore, by lemma 2.3, we can conclude that $\int f \leq \int s$.

Inductive case: We proceed by considering the three possible cases: $P = Q; R$, $P = Q \triangleleft b \triangleright R$ and $P = Q * b$.

Sequential composition: $P = Q; R$

$$
\begin{array}{ll}
 & \Psi_s^f(P) \\
= & \{ \text{ by definition of } \Psi \ \} \\
 & \Psi_s^m(Q) \parallel \Psi_m^f(R) \\
\Rightarrow & \{ \text{ by inductive hypothesis } \} \\
 & \int f \leq \int m \wedge \int m \leq \int s \\
\Rightarrow & \{ \leq \text{ is transitive } \} \\
 & \int f \leq \int s
\end{array}
$$

Conditional: $P = Q \triangleleft b \triangleright R$

$$
\begin{array}{ll}
 & \Psi_s^f(P) \\
= & \{ \text{ by definition of } \Psi \ \} \\
 & \Psi_{s_Q}^{f_Q}(Q) \parallel \Psi_{s_R}^{f_R}(R) \parallel \mathcal{I}\text{nterface}_C
\end{array}
$$

$$
\begin{array}{ll}
\Rightarrow & \{ \text{ by inductive hypothesis } \} \\
& \int f_Q \leq \int s_Q \wedge \int f_R \leq \int s_R \wedge \mathcal{I}\text{nterface}_C \\
\Rightarrow & \{ \text{ by definition of } \mathcal{I}\text{nterface}_C \text{ } \} \\
& \int f_Q \leq \int s_Q \wedge \int f_R \leq \int s_R \wedge \int s_Q + \int s_R = \int s \wedge \\
& \qquad \int f = \int f_Q + \int f_R - \int (f_Q \wedge f_R) \\
\Rightarrow & \{ \text{ by properties of } \leq \text{ and } \int \text{ } \} \\
& \int f \leq \int s
\end{array}
$$

Loops: $P = Q * b$

$$
\begin{array}{ll}
& \Psi_s^f(P) \\
= & \{ \text{ by definition of } \Psi \text{ } \} \\
& \Psi_{s_Q}^{f_Q}(Q) \parallel \mathcal{I}\text{nterface}_L \\
\Rightarrow & \{ \text{ by inductive hypothesis } \} \\
& \int f_Q \leq \int s_Q \wedge \mathcal{I}\text{nterface}_L \\
\Rightarrow & \{ \text{ by definition of } \mathcal{I}\text{nterface}_L \text{ and integral reasoning } \} \\
& \int f_Q \leq \int s_Q \wedge \int f = \int s - (\int s_Q - \int f_Q) - \int (f_Q \wedge b \wedge s) \\
\Rightarrow & \{ \text{ by properties of } \leq \text{ } \} \\
& \int f \leq \int s
\end{array}
$$

This completes the inductive step and hence the result holds by induction.

□

Lemma 2.5: $\Psi_s^f(P) \Rightarrow \int f \leq \int s \gg 1$

Proof: The proof follows almost identically to that of lemma 2.4.

□

Theorem 2: If s is a signal, then:

$$\int s \leq \int f + 0.5 \vdash \psi_s^f(P) \sqsubseteq \Psi_s^f(P)$$

Proof: The proof follows by induction on the structure of the program P.

In the base case, when P is a simple program, $\Psi_s^f(P)$ is just $\psi_s^f(P)$, and hence trivially guarantees correctness.

For the inductive case we consider the different possibilities:

Sequential composition: We need to prove that $\int s \leq \int f + 0.5 \vdash \psi_s^f(Q;R) \sqsubseteq \Psi_s^f(Q;R)$

Assume that $\int s \leq \int f + 0.5$.

By lemma 2.4, $\Psi_s^f(Q;R)$ guarantees that $\int f \leq \int s$ and hence:

$$\int f \leq \int s \leq \int f + 0.5$$

But, by definition of Ψ, and the further application of lemma 2.4:

$$\begin{array}{rcl} \Psi_s^m(Q') & \| & \Psi_m^f(R') \\ \int m & \leq & \int s \\ \int f & \leq & \int m \end{array}$$

Hence, combining the above inequalities with the previous ones:

$$\begin{array}{rcl} \int m & \leq & \int f + 0.5 \\ \int s & \leq & \int m + 0.5 \end{array}$$

By the inductive hypothesis, we thus conclude that:

$$\psi_s^m(Q') \| \psi_m^f(R')$$

Also, by lemma 2.5 we can conclude that $\int f \leq \int s \gg 1$. This, together with the first inequality allows us to apply lemma 2.1, implying that $i\epsilon_s^f$. Thus we can apply theorem 1.1 to conclude that $\psi_s^f(Q;R)$.

Therefore, $\int s \leq \int f + 0.5 \vdash \psi_s^f(P;Q) \sqsubseteq \Psi_s^f(P;Q)$.

Conditional: We need to prove that:
$$\int s \leq \int f + 0.5 \vdash \psi_s^f(Q \triangleleft b \triangleright R) \sqsubseteq \Psi_s^f(Q \triangleleft b \triangleright R)$$

As before, we know that:

$$\int f \leq \int s \leq \int f + 0.5$$

Also, by definition of Ψ and lemma 2.4:

$$\begin{array}{rcl} \Psi_{s_Q}^{f_Q}(Q) \ \| & \Psi_{s_R}^{f_R}(R) & \| \ \mathcal{I}\text{nterface}_C \\ \int f_R & \leq & \int s_Q \\ \int f_R & \leq & \int s_R \end{array}$$

Using simple duration calculus arguments on the interface part, we can conclude that:

$$\begin{array}{rcl} \int s & = & \int s_Q + \int s_R \\ \int f & = & \int f_Q + \int f_R - \int (f_Q \wedge f_R) \end{array}$$

Hence:

$$\begin{array}{rl} & \int s \leq \int f + 0.5 \\ \Rightarrow & \int s_Q + \int s_R \leq \int f_Q + \int f_R + 0.5 - \int (f_Q \wedge f_R) \\ \Rightarrow & \int s_Q \leq \int f_Q + 0.5 - (\int s_R - \int f_R) - \int (f_Q \wedge f_R) \\ \Rightarrow & \int s_Q \leq \int f_Q + 0.5 \end{array}$$

The last step is justified since $\int s_R \geq \int f_R$.

The same argument can be used to show that $\int s_R \leq \int f_R + 0.5$. Hence, we can use the inductive hypothesis to conclude that:

$$\psi_{s_Q}^{f_Q}(Q) \parallel \psi_{s_R}^{f_R}(R) \parallel \mathcal{I}\text{nterface}_C$$

But the first inequality and lemma 2.5 imply (by lemma 2.1) that $i\epsilon_s^f$, and thus by theorem 1.2 we can conclude the desired result:
$\int s \leq \int f + 0.5 \vdash \psi_s^f(Q \triangleleft b \triangleright R) \sqsubseteq \Psi_s^f(Q \triangleleft b \triangleright R)$.

Loops: Finally, we need to prove that $\int s \leq \int f + 0.5 \vdash \psi_s^f(Q * b) \sqsubseteq \Psi_s^f(Q * b)$.

The argument is almost identical to the one given for the conditional statement, except that the equality we need to derive from the interface so as to enable us to complete the proof is that:

$$\int s_P = \int s + \int f_P - \int f - \int s \wedge f_P \wedge b$$

Hence, by induction, we can conclude that $\int s \leq \int f + 0.5 \vdash \psi_s^f(P) \sqsubseteq \Psi_s^f(P)$.
□

Corollary: $i\epsilon_s^f \vdash i\psi_s^f(P) \sqsubseteq \Psi_s^f(P)$.

Proof: Follows immediately from lemma 2.2 and theorem 2.
□

10.5 Conclusions

10.5.1 Other Constructs

The definition of a number of simple constructs into a hardware-like format allows this reasoning to be defined for other instructions. Thus, for example, if we are interested in `while` loops, we can use the following algebraic reasoning to justify a compilation rule. First note that $b * P$ is not a valid program to be used in the transformation since it could possibly take zero time. Hence, we consider $b * P \parallel \#1$:

$$
\begin{array}{ll}
 & b * P \parallel \#1 \\
= & \{ \text{ definition of while loops } \} \\
 & ((P; b * P) \triangleleft b \triangleright \texttt{skip}) \parallel \#1 \\
= & \{ \text{ distribution of parallel composition into conditional } \} \\
 & (P; b * P \parallel \#1) \triangleleft b \triangleright (\texttt{skip} \parallel \#1) \\
= & \{ \text{ laws of } \#1 \text{ with parallel composition and } \mathsf{dur}(P) \ \} \\
 & (P; b * P) \triangleleft b \triangleright \#1 \\
= & \{ \text{ by definition of repeat loops } \} \\
 & (P * b) \triangleleft b \triangleright \#1
\end{array}
$$

Hence, by monotonicity, $\psi_s^f(b * P \parallel \#1) = \psi_s^f(P * b \triangleleft b \triangleright \#1)$. Using the compilation rules for repeat loops and conditional, we can therefore define a new compilation rule for this type of loop:

$$\Psi_s^f(b * P \parallel \#1) \stackrel{def}{=} \Psi_{s_1}^{f_1}(\#1) \parallel \Psi_{s_P}^{f_P}(P') \parallel \mathsf{Merge} \parallel \mathcal{I}\text{nterface}$$

where the interface is defined by:

$$
\begin{array}{rcl}
\mathcal{I}\text{nterface} & \stackrel{def}{=} & \texttt{assign } s_1 = s \wedge \neg b \\
 & \parallel & \texttt{assign } s_* = s \wedge b \\
 & \parallel & \texttt{assign } f = f_1 \vee f_* \\
 & \parallel & \texttt{assign } s_P = s_* \vee (f_P \wedge b) \\
 & \parallel & \texttt{assign } f_* = f_P \wedge \neg b
\end{array}
$$

10.5.2 Basic Instructions

Implementation of a number of basic instructions in terms of continuous assignments can also be easily done. Consider, for example:

$$
\begin{array}{rcl}
\Psi_s^f(\#1) & \stackrel{def}{=} & \texttt{assign}_T \ f = \#1 \ s \\
\Psi_s^f(\#1 \ v = e) & \stackrel{def}{=} & \texttt{assign}_T \ f = \#1 \ s \\
 & \parallel & \texttt{assign}_T \ v^- = \#0.5 \ v \\
 & \parallel & \texttt{assign } v = e \triangleleft f \triangleright v^-
\end{array}
$$

For a definition $\Psi_s^f(P) \stackrel{def}{=} Q$, it is enough to verify that $i\epsilon_s^f \vdash \psi_s^f(P) \sqsubseteq Q$. The result of the corollary can then be extended to cater for these compilation rules. Hence, these laws can be verified, allowing a *total* compilation of a program written in terms of these instructions and the given constructs into continuous assignments.

10.5.3 Single Runs

Finally, what if we are interested in running a compiled program just once? It is easy to see that $i\epsilon_s^f \sqsubseteq \texttt{initial } s!$. Also, we can prove that:

$$\texttt{initial } s! \parallel \psi_s^f(P) = \texttt{initial } s!; P; f!$$

Hence, for a single run of the program, we simply add an environment satisfying the desired property: `initial` $s!$.

10.5.4 Overall Comments

This chapter applies to Verilog a number of techniques already established in the hardware compilation community [KW88, May90, Spi97, PL91], giving us a number of compilation rules which translate a sequential program into a parallel one. Most of the proof steps involve a number of applications of simple laws of Verilog, and would thus benefit from machine verification.

One interesting result of the approach advocated here is the separation placed between the control and data paths, which is clearly visible from the compilation procedure.

The method used here is very similar to the compilation procedure used with Occam in [May90] and Handel in [HJ94, Pag93]. The transformation depends heavily on the timing constraints — unlike the approach usually taken by commercial synthesis tools which usually synchronise using global clock and reset signals [Pal96, SSMT93]. The main difference between the compilation of Verilog programs we define with that of Occam or Handel is the fact that timing control can be explicitly expressed in Verilog. It is thus not acceptable to assume that immediate assignments take a whole time unit to execute (as is done in the case of Occam and Handel). It was however necessary to impose the constraint that all compiled programs take some time to execute. This limitation obviously allows us to compile only a subset of Verilog programs. However, clever use of algebraic laws can allow the designer to modify code so as to enable compilation. How much of this can be done automatically and efficiently by the compiler itself is still an open question.

The results in this chapter can be seen as one possible projection of a program into a highly parallel format. The orthogonal projection into a sequential format is precisely what a simulator does. The next chapter treats this question more formally by deriving an operational semantics of Verilog from the continuation semantics. The transitions of the operational semantics can then be interpreted as sequential instructions, effectively transforming parallel programs into a sequential form.

Chapter 11

An Operational Semantics of Verilog

11.1 Introduction

As is the case with the majority of hardware description languages, one of the high priorities in the design of Verilog was ease of simulation. In certain cases, this was given precedence over giving the language a straightforward behavioural interpretation which can be explained without having to go into the simulation cycle semantics. This is one of the reasons behind the complexity and intricacy of the denotational semantics given to the language.

On the other hand, this kind of approach to language design can sometimes mean that an elegant operational semantics of the language is possible. In such cases, one possible alternative approach to giving a denotational semantics to such a language would be to derive it from the operational semantics. Unfortunately, in the case of complex simulation semantics, this is not necessarily a trivial task. The resulting denotational semantics may be complete with respect to the operational ones but far too complex to be of any practical use.

Looking at the problem from a different perspective, we can first define a denotational semantics of the subset of the language in which we are interested, from which an operational semantics can then be derived. The resulting semantics can then be used to implement as a simulator guaranteed to match our original interpretation of the language. This is the approach taken here.

Transitions in the operational semantics are first given an algebraic interpretation, effectively giving an algebraic semantics to the language. A number of transition rules then follow immediately from the laws given in chapter 6, the correctness of which guarantees that the denotational semantics give a solution to the algebraic semantics. It is then shown that the only solution to these equations is, in fact, the denotational one, proving the equivalence of the two definitions.

Stopping at this point guarantees a correct operational semantics. However, if we implement the transition laws as a simulator we have no guarantee that it will always terminate when asked to give the behaviour of a program for a length of time. Presence of a transition law like $P \longrightarrow P$ may be executed repeatedly by the simulator without reaching a state in which simulation time is advanced any further. We thus finally prove that any execution order allowed by the operational semantics must eventually lead to simulation time being advanced.

The language for which we will derive the operational semantics is a subset of the basic language. In [HJ98], C.A.R. Hoare and Jifeng He show how an operational semantics can be determined from a denotational semantics via an algebraic semantics. To prove the equivalence of the semantics we adopt an approach similar to the one used in this book and other similar work [Hoa94] but avoid giving a complete algebraic semantics. However, the algebraic laws given in chapter 6 will be found very useful in avoiding reproof of certain properties. What is innovative in our presentation is the relevance of intermediate values in the semantics.

11.2 The Program Semantics

11.2.1 Infinite Tail Programs

It will be convenient to discuss the denotational semantics without using continuations. If all modules are reduced to `initial` ones[1], we can prove that, for any program P:

$$[\![\texttt{initial}\ P]\!] = [\![\texttt{initial}\ P;\ \texttt{END}]\!]$$

where $[\![\texttt{END}]\!]_W(D) \stackrel{def}{=} \mathsf{Const}(W)$.

Any program ending with the above instruction will be called an infinite tail program. For such programs P, $[\![P]\!]_W(D)$ is the same for any duration formula D. We will be thus justified in writing the semantic interpretation operator for infinite tail programs simply as $[\![P]\!]_W$.

This statement obeys the following law, which we will need later:

$$\texttt{END = \#1; END}$$

11.2.2 Finite Loops

On the other hand, when using the simulation semantics we are only interested in finite time prefix behaviour.

Since all while loop bodies must take time to execute, we can prove that:

$$l \leq n \Rightarrow [\![P]\!] = [\![P_n]\!]$$

where P_n is the result of replacing all loops (`while b do` Q) in P by finite loops (`while`$_n$ `b do Q`):

$$\begin{array}{rcl} \texttt{while}_0\ \texttt{do}\ P & \stackrel{def}{=} & \texttt{skip} \\ \texttt{while}_{n+1}\ \texttt{do}\ P & \stackrel{def}{=} & \texttt{if b then (}P\texttt{; while}_n\ \texttt{do}\ P\texttt{) else skip} \end{array}$$

This enables us to reduce any program to a finite one, for whatever the length of time we would like to monitor the behaviour. This simplifies certain proofs.

[1] We will not include continuous assignments in our presentation and this is therefore justified.

11.3 The Operational Semantics

To state the operational semantics, we use the following notation:

$$(s, P) \longrightarrow (s', Q)$$

This is read as 'executing P in state s will change the storage to s' and executable program to Q'. P and Q range over valid Verilog programs, s and s' are states storing the values of the variables and $\longrightarrow$ is a transition relation.

Formally, a state is a function from variable names to values. Given a state s and variable v in its domain, $s(v)$ is the value of v in that state. For an expression e built from variables in the domain of state s, $s[v \leftarrow e]$ represents a new state matching s in all variables except v, which takes the value of expression e as evaluated in s. To make the presentation more readable, we will also interpret states as assignment statements. The assignment statement s refers to the assignment of all variables v in the domain of s to $s(v)$.

Two transition relations are defined: immediate changes (written as $\xrightarrow{0}$) and time increasing changes (written as $\xrightarrow{1}$). These will then be combined to give the complete operational semantics of Verilog. The simulation semantics can be informally described as follows:

1. While there are any immediate changes possible, choose one, perform it and repeat this step.
2. Perform a time increasing change along all parallel threads.
3. Go back to step 1.

Note that for any valid program in the subset of Verilog for which we defined the semantics, step 1 eventually terminates since all loop bodies must take time to execute.

11.3.1 Immediate Transitions

Certain transitions can take place without simulation time being advanced. These transitions will be written as $(s, P) \xrightarrow{0} (s', Q)$, and read as '$P$ in state s executes to Q with state s' in zero time'.

Rather than simply coming up with the operational semantics, we give a formal definition to the transition notation and derive the operational semantics from the denotational semantics.

$$(s, P) \xrightarrow{0} (s', Q) \stackrel{def}{=} s; P = s'; Q$$

For sequential programs P, Q and R, we derive the following transition relation:

$$
\begin{array}{rcll}
(s, \texttt{skip};\ P) & \xrightarrow{0} & (s, P) & \text{[OP–skip]} \\
(s, \#0;\ P) & \xrightarrow{0} & (s, P) & \text{[OP–\#0]} \\
(s, v = e;\ P) & \xrightarrow{0} & (s[v \leftarrow e], P) & \text{[OP–:=]} \\
\multicolumn{4}{l}{\text{Provided that } s(b) = true} \\
(s, (\texttt{if } b \texttt{ then } P \texttt{ else } Q);\ R) & \xrightarrow{0} & (s, P;\ R) & \text{[OP–cond(T)]} \\
\multicolumn{4}{l}{\text{Provided that } s(b) = false} \\
(s, (\texttt{if } b \texttt{ then } P \texttt{ else } Q);\ R) & \xrightarrow{0} & (s, Q;\ R) & \text{[OP–cond(F)]} \\
\multicolumn{4}{l}{\text{Provided that } s(b) = true} \\
(s, (\texttt{while } b \texttt{ do } P);\ R) & \xrightarrow{0} & (s, P;\ (\texttt{while } b \texttt{ do } P);\ R) & \text{[OP–while(T)]} \\
\multicolumn{4}{l}{\text{Provided that } s(b) = false} \\
(s, (\texttt{while } b \texttt{ do } P);\ R) & \xrightarrow{0} & (s, R) & \text{[OP–while(F)]} \\
\multicolumn{4}{l}{\text{Provided that } s(v) = true} \\
(s, \texttt{wait } v;\ R) & \xrightarrow{0} & (s, R) & \text{[OP–wait(T)]}
\end{array}
$$

Note that all these transition rules hold when interpreted as algebraic laws.

Definition: A sequential Verilog program P in state S is said to be *stable* if none of the defined zero transition laws are applicable to it.

$$stable(s, P) \stackrel{def}{=} \nexists t,\ Q \cdot (s, P) \xrightarrow{0} (t, Q)$$

Definition: The multiple zero transition relation $\xrightarrow{0}^*$ is defined to be the reflexive transitive closure of $\xrightarrow{0}$.

Definition: A program P in state s is said to stablise to program Q in state t, if there is a sequence of zero transitions which transform (s, P) to (t, Q) and Q in t is stable:

$$(s, P) \stackrel{0}{\longmapsto} (t, Q) \stackrel{def}{=} (s, P) \xrightarrow{0}^* (t, Q) \wedge stable(t, Q)$$

Proposition 0: Interpreted in the same way as $\xrightarrow{0}$, stablising zero transitions are also sound.

In other words, if $(s, P) \stackrel{0}{\longmapsto} (t, Q)$, then $s; P = t; Q$.

Lemma 0: If P is a sequential program with no loops, such that $\mathsf{dur}(P)$, then for any state s, there is a program Q and state t such that $(s, P) \stackrel{0}{\longmapsto} (t, Q)$.

The lemma can be proved using induction on the structure of P and the definition of $\mathsf{dur}(P)$.

Corollary 0: $(s, P; (\texttt{while } b \texttt{ do } Q); R)$ stablises, provided that $(s, P; R)$ stablises.

Since we know that $\mathsf{dur}(Q)$ (from the fact that Q appears as the body of a loop), we can use case analysis on whether b is true or not at the start of the loop and Lemma 0 to prove the statement.

Lemma 1: If P is a sequential program (of the form P';`END`), we can find a program Q and state t, such that: $(s, P) \stackrel{0}{\longmapsto} (t, Q)$.

Proof: Note that using Corollary 0, it is enough if we show that P with all the loops removed can reach a stable state.

We define the size of a (loop-free) sequential program $size(P)$ to be:

$$\begin{aligned} size(S) &\stackrel{def}{=} 1 \\ size(\texttt{if b then } P \texttt{ else } Q) &\stackrel{def}{=} 1 + size(P) + size(Q) \\ size(C;Q) &\stackrel{def}{=} size(C) + size(Q) \end{aligned}$$

where S is a single simple instruction program (such as `v=e`, `END` etc) and C is a single compound program (a single instruction or a conditional).

From the following facts:

- on non-stable program and state pairs, $\xrightarrow{0}$ is, by definition, a total function
- the initial input is always of the form `END` or P`; END` with P being a finite program
- any transition on a program of the form P`; END` produces either `END` or a program Q`; END` where $size(Q) < size(P)$

we conclude that in at most $(size(P) - 1)$ transitions, we must eventually hit on a stable program.

□

11.3.2 Extension to Parallel Composition

The ideas just presented can be readily extended to programs using parallel composition. As a convention, we will use bold variables such as $\mathbb{P}$, $\mathbb{Q}$ to represent parallel programs and $\mathbb{P}_i$, $\mathbb{Q}_i$ for their components. The state is also partitioned into parts, each of which is controlled by a single process (but which other processes can read). We refer to the partition controlled by process i as s_i.

Definition: The $\xrightarrow{0}$ relation can be extended to parallel programs as follows:

If $(s_i, \mathbb{P}_i) \xrightarrow{0} (t_i, \mathbb{Q}_i)$, then $(s, \mathbb{P}) \xrightarrow{0} (t, \mathbb{Q})$, where for any $j \neq i$, $\mathbb{P}_j = \mathbb{Q}_j$ and $s_j = t_j$.

The definition for stability of programs remains unchanged: a parallel program $\mathbb{P}$ is said to be stable if it cannot perform any zero transitions.

Corollary: A parallel program $\mathbb{P}$ is stable if and only if all its components are stable:

$$stable(\mathbb{P}) \iff \bigwedge_i stable(\mathbb{P}_i)$$

The definition of multiple zero transitions and stablising zero transitions also remain unchanged.

Lemma 2: All zero transitions interpreted as equations are consequences of the continuation semantics of Verilog.

Proof: All sequential program transitions follow immediately from the algebraic laws of chapter 6. The parallel program transitions then follow from the distributivity of assignment over parallel composition.

□

Lemma 3: For any loop-free $\mathbb{P}$ and state s, application of zero transitions in any order must eventually terminate. That is, by repeatedly applying zero transitions, we must eventually hit upon a stable state.

Proof: The proof is simply an extension to the one given for lemma 1. We extend the definition of the size of a program to parallel programs:

$$size(\mathbb{P}) \quad \stackrel{def}{=} \quad \sum_i \mathbb{P}_i$$

Following precisely the same argument in lemma 1, zero transitions reduce the size of programs. Since all programs end in a stable instruction `END`, in at most $size(\mathbb{P})$ transitions, the program must stablise.

□

11.3.3 Unit Time Transitions

Once no further immediate transitions can take place, simulation time is advanced throughout the system without changing the state. A relation $\stackrel{1}{\longrightarrow}$ is defined to specify how such a transition takes place.

As in the case of the $\stackrel{0}{\longrightarrow}$ relation, we give an interpretation to this new transition to direct the definition of laws:

$$(s, P) \stackrel{1}{\longrightarrow} (t, Q) \stackrel{def}{=} s; P = t; \#1; Q$$

$$\boxed{\begin{array}{rcll} (s, \#(n+1);\ P) & \stackrel{1}{\longrightarrow} & (s, \#n;\ P) & \text{[OP–\#n]} \\ (s, END) & \stackrel{1}{\longrightarrow} & (s, END) & \text{[OP–END]} \\ \multicolumn{4}{l}{\text{If } S(v) = false} \\ (s, \mathtt{wait}\ v;\ P) & \stackrel{1}{\longrightarrow} & (s, \mathtt{wait}\ v;\ P) & \text{[OP–wait(F)]} \end{array}}$$

This relation is extended to parallel processes by using:

If, for all i, $(s, \mathbb{P}_i) \stackrel{1}{\longrightarrow} (s, \mathbb{Q}_i)$, then $(s, \mathbb{P}) \stackrel{1}{\longrightarrow} (s, \mathbb{Q})$.

Lemma 4: For any program $\mathbb{P}$ and state s, such that $stable(s, \mathbb{P})$, there exists a program $\mathbb{Q}$ such that $(s, \mathbb{P}) \stackrel{1}{\longrightarrow} (s, \mathbb{Q})$.

Furthermore, $\mathbb{Q}$ is unique.

Proof: $\mathbb{P}$ is stable if all its components are stable. The unit transition relation is total on stable sequential states and can thus be applied to all components of $\mathbb{P}$. Uniqueness follows from the functionality of the unit transition relation.

□

Lemma 5: All unit time transitions are consequences of the continuation semantics.

Proof: The proof is identical to the one for lemma 2, but uses distributivity of unit delay over parallel composition rather than the distributivity of assignment statements.

$\square$

Definition: A total time step transition is one which evolves a program $\mathbb{P}$ with state s until it becomes stable, upon which a time transition takes place.

$$(s, \mathbb{P}) \twoheadrightarrow (t, \mathbb{Q}) \stackrel{def}{=} \exists \mathbb{R},\ u \cdot (s, \mathbb{P}) \stackrel{0}{\longmapsto} (u, \mathbb{R}) \wedge (u, \mathbb{R}) \stackrel{1}{\longrightarrow} (t, \mathbb{Q})$$

We would like to show that this relation ($\twoheadrightarrow$) is total. The first thing to prove is that $\stackrel{0}{\longmapsto}$ is still total on parallel programs. As in the case of sequential programs, the result is proved by giving a monotonically decreasing integer variant which is bound below.

Consider the following definition of $size'$ (where P, Q and R stand for any program and C stands for a single simple instruction, such as assignment, `END`, etc):

$$\begin{array}{rcl}
size'(C) & \stackrel{def}{=} & 1 \\
size'(\texttt{while } b \texttt{ do } P) & \stackrel{def}{=} & size'(P) + 1 \\
size'(\texttt{if } b \texttt{ then } P \texttt{ else } Q) & \stackrel{def}{=} & size'(P) + size'(Q) + 1 \\
size'(C; P) & \stackrel{def}{=} & \begin{cases} 1 & \text{if } \mathsf{dur}(C) \\ 1 + size'(P) & \text{otherwise} \end{cases} \\
size'((\texttt{if } b \texttt{ then } P \texttt{ else } Q); R) & \stackrel{def}{=} & \begin{cases} size'(P) + size'(Q) + 1 \\ \qquad \text{if } \mathsf{dur}(\texttt{if } b \texttt{ then } P \texttt{ else } Q) \\ size'(P) + size'(Q) + size'(R) \\ \qquad \text{otherwise} \end{cases} \\
size'((\texttt{while } b \texttt{ do } P); Q) & \stackrel{def}{=} & size'(P) + size'(Q) + 1
\end{array}$$

Lemma 6: If $\mathsf{dur}(P)$, then for any program Q, $size'(P;Q) = size'(P)$. Otherwise, if $\mathsf{dur}(P)$ is false, then for any program Q, $size'(P;Q) = size'(P) + size'(Q)$.

Proof: The proof proceeds by structural induction on P.

Base cases: If P is simple instruction:

- If $\mathsf{dur}(P)$, then:

$$\begin{array}{ll}
 & size'(P;Q) \\
= & \{ \text{ definition of } size' \ \} \\
 & 1 \\
= & \{ \text{ definition of } size' \ \} \\
 & size'(P)
\end{array}$$

- If not $\mathsf{dur}(P)$, then:

$$\begin{array}{ll}
 & size'(P;Q) \\
= & \{ \text{ definition of } size' \ \} \\
 & 1 + size'(Q) \\
= & \{ \text{ definition of } size' \ \} \\
 & size'(P) + size'(Q)
\end{array}$$

Inductive cases:

- P is a loop (`while` b `do` R): Note that, by definition, $\mathsf{dur}(P)$ is false:

$$\begin{array}{ll} & size'(P;Q) \\ = & size'(\texttt{while}\ b\ \texttt{do}\ R;Q) \\ = & \{\ \text{definition of } size'\ \} \\ & size'(R) + size'(Q) + 1 \\ = & \{\ \text{definition of } size'\ \} \\ & size'(\texttt{while}\ b\ \texttt{do}\ R) + size'(Q) \\ = & size'(P) + size'(Q) \end{array}$$

- P is a conditional (`if` b `then` R `else` S):

  - Subcase 1: $\mathsf{dur}(R)$ and $\mathsf{dur}(S)$.

$$\begin{array}{ll} & size'(P;Q) \\ = & size'((\texttt{if}\ b\ \texttt{then}\ R\ \texttt{else}\ S);Q) \\ = & \{\ \text{definition of } size' \text{ and } \mathsf{dur}(P)\ \} \\ & size'(R) + size'(S) + 1 \\ = & \{\ \text{definition of } size' \\ & size'(P) \end{array}$$

  - Subcase 2: At least one of $\mathsf{dur}(R)$ and $\mathsf{dur}(S)$ is false (and hence so is $\mathsf{dur}(P)$).

$$\begin{array}{ll} & size'(P;Q) \\ = & size'((\texttt{if}\ b\ \texttt{then}\ R\ \texttt{else}\ S);Q) \\ = & \{\ \text{definition of } size' \text{ and the fact that } \mathsf{dur}(P) \text{ is false}\ \} \\ & size'(R) + size'(S) + size'(Q) + 1 \\ = & \{\ \text{definition of } size'\ \} \\ & size'(\texttt{if}\ b\ \texttt{then}\ R\ \texttt{else}\ S) + size'(Q) \\ = & size'(P) + size'(Q) \end{array}$$

- P is a sequential composition ($P = R;S$):

  - Subcase 1: $\mathsf{dur}(R)$ holds (and hence so does $\mathsf{dur}(P)$).

$$\begin{array}{ll} & size'(P;Q) \\ = & size'(R;S;Q) \\ = & \{\ \text{inductive hypothesis and } \mathsf{dur}(R)\ \} \\ & size'(R) \\ = & \{\ \text{inductive hypothesis and } \mathsf{dur}(R)\ \} \\ & size'(R;S) \\ = & size'(P) \end{array}$$

  - Subcase 2: $\mathsf{dur}(S)$ holds, but not $\mathsf{dur}(R)$ (hence $\mathsf{dur}(P)$ holds).

$$\begin{array}{ll} & size'(P;Q) \\ = & size'(R;S;Q) \\ = & \{\ \text{inductive hypothesis and } \mathsf{dur}(R) \text{ is false}\ \} \\ & size'(R) + size'(S;Q) \\ = & \{\ \text{inductive hypothesis and } \mathsf{dur}(S)\ \} \\ & size'(R) + size'(S) \\ = & \{\ \text{inductive hypothesis and } \mathsf{dur}(R) \text{ is false}\ \} \\ & size'(R;S) \\ = & size'(P) \end{array}$$

  - Subcase 3: Both $\mathsf{dur}(R)$ and $\mathsf{dur}(S)$ are false (therefore $\mathsf{dur}(P)$ is also false).

$$\begin{array}{ll} & size'(P;Q) \\ = & size'(R;S;Q) \\ = & \{ \text{ inductive hypothesis and } \mathsf{dur}(R) \text{ is false } \} \\ & size'(R) + size'(S;Q) \\ = & \{ \text{ inductive hypothesis and } \mathsf{dur}(S) \text{ is false } \} \\ & size'(R) + size'(S) + size'(Q) \\ = & \{ \text{ inductive hypothesis and } \mathsf{dur}(R) \text{ is false } \} \\ & size'(R;S) + size'(Q) \\ = & size'(P) \end{array}$$

□

Lemma 7: $\twoheadrightarrow$ is total.

Proof: We will show that if $(s, P) \xrightarrow{0} (t, Q)$, then $size'(P) > size'(Q)$. Since $size'$ is bounded below by 1, any chain of valid immediate transitions must be finite. Since, by definition the set of time transition ($\xrightarrow{1}$) is a total relation on stable programs, the totality of $\twoheadrightarrow$ will then follow immediately.

We proceed by analysing all possible immediate transitions:

- $(s, \texttt{skip}; P) \xrightarrow{0} (s, P)$:

$$\begin{array}{ll} & size'(\texttt{skip}\ ; P) \\ = & \{ \text{ by definition of } size' \text{ and } \mathsf{dur} \ \} \\ & 1 + size'(P) \\ > & size'(P) \end{array}$$

- $(s, \#0; P) \xrightarrow{0} (s, P)$: The proof is identical to the one in the previous case.

- $(s, v = e; P) \xrightarrow{0} (s[v \leftarrow e], P)$: Again, the proof is identical to the first.

- $(s, \texttt{wait}\ v; P) \xrightarrow{0} (s, P)$: Again, the proof is identical to the first.

- $(s, (\texttt{if}\ b\ \texttt{then}\ Q\ \texttt{else}\ R); S) \xrightarrow{0} (s, Q; S)$:

  Subcase 1: Both $\mathsf{dur}(Q)$ and $\mathsf{dur}(R)$ hold.

$$\begin{array}{ll} & size'((\texttt{if}\ b\ \texttt{then}\ Q\ \texttt{else}\ R); S) \\ = & \{ \text{ since } \mathsf{dur}(\texttt{if}\ b\ \texttt{then}\ Q\ \texttt{else}\ R) \ \} \\ & size'(Q) + size'(R) + 1 \\ = & \{ \ \mathsf{dur}(Q) \ \} \\ & size'(Q;S) + size'(R) + 1 \\ > & size'(Q;S) \end{array}$$

  Subcase 2: $\mathsf{dur}(Q)$ holds but not $\mathsf{dur}(R)$.

$$\begin{array}{ll} & size'((\texttt{if}\ b\ \texttt{then}\ Q\ \texttt{else}\ R); S) \\ = & \{ \text{ since } \mathsf{dur}(\texttt{if}\ b\ \texttt{then}\ Q\ \texttt{else}\ R) \text{ is false } \} \\ & size'(Q) + size'(R) + size'(S) + 1 \\ > & size'(Q) \\ = & \{ \text{ lemma 6 and } \mathsf{dur}(Q) \ \} \\ & size'(Q;S) \end{array}$$

  Subcase 3: $\mathsf{dur}(R)$ holds but not $\mathsf{dur}(Q)$.

$$\begin{array}{rl}
& size'((\texttt{if}\ b\ \texttt{then}\ Q\ \texttt{else}\ R); S) \\
= & \{ \text{ since } \texttt{dur}(\texttt{if}\ b\ \texttt{then}\ Q\ \texttt{else}\ R) \text{ is false } \} \\
& size'(Q) + size'(R) + size'(S) + 1 \\
> & size'(Q) + size'(S) \\
= & \{ \text{ lemma 6 and } \texttt{dur}(Q) \text{ is false } \} \\
& size'(Q; S)
\end{array}$$

- $(s, (\texttt{if}\ b\ \texttt{then}\ Q\ \texttt{else}\ R); S) \xrightarrow{0} (s, R; S)$: The proof is symmetric to the one given in the previous case.

- $(s, (\texttt{while}\ b\ \texttt{do}\ Q); R) \xrightarrow{0} (s, R)$:

$$\begin{array}{rl}
& size'((\texttt{while}\ b\ \texttt{do}\ Q); R) \\
= & \{ \text{ by definition of } size' \} \\
& size'(Q) + size'(R) + 1 \\
> & size'(R)
\end{array}$$

- $(s, (\texttt{while}\ b\ \texttt{do}\ Q); R) \xrightarrow{0} (s, Q; (\texttt{while}\ b\ \texttt{do}\ Q); R)$: Note that $\texttt{dur}(Q)$ must hold, since Q is a loop body.

$$\begin{array}{rl}
& size'((\texttt{while}\ b\ \texttt{do}\ Q); R) \\
= & \{ \text{ by definition of } size' \} \\
& size'(Q) + size'(R) + 1 \\
> & size'(Q) \\
& \{ \text{ lemma 6 and } \texttt{dur}(Q) \} \\
= & size'(Q; (\texttt{while}\ b\ \texttt{do}\ Q); R)
\end{array}$$

$\square$

11.4 Equivalence of Semantics

11.4.1 An Informal Account

The operational semantics give a set of algebraic equations which programs will satisfy. These can be seen as a number of simultaneous equations which an interpretation of the language should satisfy. Showing that the continuation semantics in fact satisfies the equations is similar to fitting a number of constants into a set of equations and showing that the equalities do in fact hold. However, a question immediately arises — are there other interpretations which satisfy these simultaneous equations? This is the question we now set out to answer.

11.4.2 Formalising the Problem

Since the laws given by the interpretation of the operational semantics are laws of the denotational interpretation of the semantics, the family of interpretations satisfying the laws is refined by the interpretation $[\![P]\!]$.

To prove equivalence, we would like to be able to show the reverse: that all the semantic interpretations satisfying the transition laws are a just a refinement of the denotational semantics. In effect, we would be showing that there is only one unique interpretation satisfying the laws: that given by the denotational semantics.

Different approaches can be taken towards proving this property. We choose to show that any meaningful temporal interpretation of the language satisfying the algebraic laws is in fact a refinement of the denotational semantics. An alternative approach would have been to show that the algebraic laws have a unique solution.

But what do we mean by a meaningful temporal interpretation of the language? The trivial language interpretation which maps all programs to *true* obviously satisfies the algebraic laws. So does the interpretation identical to the one given in chapter 5 but changing the semantics of `#n` to:

$$\begin{array}{rcl} [\![\#\mathtt{n}]\!]_W(D) & \stackrel{def}{=} & (l \leq 2n \wedge \mathsf{Const}(W)) \vee \\ & & (l = 2n \wedge \mathsf{Const}(W)) \mathbin{;} D \end{array}$$

The solution we adopt is to define a relation on programs, $\leq_n$, such that $P \leq_n Q$ if Q refines the behaviour of P over the first n time units. We can then simply say that P is refined by Q ($P \leq Q$) if, for any number n, $P \leq_n Q$. Two programs are then said to be equivalent if they are refinements of each other.

Using this technique, we can define a refinement relation for both the operational and denotational semantics. The two semantics are then identical if the equivalence relations defined by them are the same. By proving that whenever two state-program pairs are equivalent in the denotational sense then they must also be equivalent in the operational sense (theorem 1, case 2) we are showing that both semantics partition the set of all possible state-program pairs in the same way and thus, the only solution to the algebraic laws is the denotational semantics given in chapter 5.

11.4.3 Refinement Based on the Denotational Semantics

Defining the timed refinement relation using the denotational semantics is quite straightforward. Obviously, a program may only be a refinement of another if both start with some particular configuration of the variables:

Definition: We define the statement 'in the denotational interpretation, program Q starting in state t refines P starting in state s for the first n time units', written as $(s, P) \sqsubseteq^{D}_{n} (t, Q)$ as:

$$\begin{array}{rcl} (s, P) \sqsubseteq^{D}_{n} (t, Q) & \stackrel{def}{=} & l = n \Rightarrow \\ & & (\exists \overleftarrow{vars}, \overrightarrow{vars} \cdot [\![\mathrm{t;Q}]\!]) \Rightarrow (\exists \overleftarrow{vars}, \overrightarrow{vars} \cdot [\![\mathrm{s;P}]\!]) \end{array}$$

Definition: We simply say that using the denotational semantics, program Q starting in state t refines P starting in state s, if the refinement holds for any time prefix:

$$(s, P) \sqsubseteq^{D} (t, Q) \quad \stackrel{def}{=} \quad \forall n : \mathbb{N} \cdot (s, P) \sqsubseteq^{D}_{n} (t, Q)$$

Definition: Program P is refined by program Q (with respect to the denotational semantics) if, whatever the starting state s, (s, P) is refined by (s, Q):

$$P \sqsubseteq^{D} Q \quad \stackrel{def}{=} \quad \forall s \cdot (s, P) \sqsubseteq^{D} (t, Q)$$

Definition: We can now define what we mean by denotational equivalence:

$$P \overset{\mathcal{D}}{=} Q \quad \overset{def}{=} \quad (P \sqsubseteq^{\mathcal{D}} Q) \wedge (Q \sqsubseteq^{\mathcal{D}} P)$$

Or more succinctly, $\overset{\mathcal{D}}{=} \;\overset{def}{=}\; \sqsubseteq^{\mathcal{D}} \cap \sqsubseteq^{\mathcal{D}\,-1}$, where R^{-1} is the inverse of relation R.

Sometimes, we will also refer to $\overset{\mathcal{D}}{=}_n$, which is defined as $\sqsubseteq^{\mathcal{D}}_n \cap \sqsubseteq^{\mathcal{D}\,-1}_n$.

11.4.4 Refinement Based on the Operational Semantics

The operational semantics have no inherent notion of time. It is *our* interpretation of the $\xrightarrow{1}$ relation that alludes to a time unit passing. This is the idea we embody when defining time refinement using operational semantics. The technique used is based on bisimulation as described in [Mil89].

Definition: We define the statement that in the operational semantics, program Q starting in state t refines P starting in state s for the first n time units, written as $(s, P) \sqsubseteq^{\mathcal{O}}_n (t, Q)$ using primitive recursion:

$$\begin{array}{rcl}
(s,P) \sqsubseteq^{\mathcal{O}}_0 (t,Q) & \overset{def}{=} & \textbf{true} \\
(s,P) \sqsubseteq^{\mathcal{O}}_{n+1} (t,Q) & \overset{def}{=} & \exists u, P', Q' \cdot \left(\begin{array}{l} (s,P) \twoheadrightarrow (u,P') \\ (t,Q) \twoheadrightarrow (u,Q') \\ (u,P') \sqsubseteq^{\mathcal{O}}_n (u,Q') \end{array} \right)
\end{array}$$

P and Q must have the same alphabet to be comparable.

Definition: We simply say that using the operational semantics, program Q starting in state t refines P starting in state s, if the refinement holds for any time prefix:

$$(s,P) \sqsubseteq^{\mathcal{O}} (t,Q) \quad \overset{def}{=} \quad \forall n : \mathbb{N} \cdot (s,P) \sqsubseteq^{\mathcal{O}}_n (t,Q)$$

Definition: Program P is refined by program Q (with respect to the operational semantics) if, whatever starting state s, (s,P) is refined by (s,Q):

$$P \sqsubseteq^{\mathcal{O}} Q \quad \overset{def}{=} \quad \forall s \cdot (s,P) \sqsubseteq^{\mathcal{O}} (s,Q)$$

Definition: We can now define what we mean by operational equivalence:

$$P \overset{\mathcal{O}}{=} Q \quad \overset{def}{=} \quad (P \sqsubseteq^{\mathcal{O}} Q) \wedge (Q \sqsubseteq^{\mathcal{O}} P)$$

Or more succinctly, $\overset{\mathcal{O}}{=} \;\overset{def}{=}\; \sqsubseteq^{\mathcal{O}} \cap \sqsubseteq^{\mathcal{O}\,-1}$.

11.4.5 Preliminary Results

Proposition 1: $\sqsubseteq^{\mathcal{O}}_n$ is a partial order (with respect to $\overset{\mathcal{O}}{=}$).

Proposition 2: $\sqsubseteq^{\mathcal{D}}_n$ is a partial order (with respect to $\overset{\mathcal{D}}{=}$).

Lemma 1: For any programs P, Q and states s, t: $(s, P) \sqsubseteq^{\mathcal{D}}_0 (t, Q)$.

Proof: The lemma is proved if we show that for any state s and program P:

$$\exists \overleftarrow{vars}, \overrightarrow{vars} \cdot (l = 0 \wedge [\![s; P]\!]) \quad = \quad l = 0$$

This can be proved using structural induction on P. To avoid the problem with while loops, we can replace $[\![s; P]\!]$ with $[\![s; P_1]\!]$.

□

Lemma 2.1: For any natural number n, programs P, Q and states s, t:

$$(s, P) \xrightarrow{0}{}^{*} (t, Q) \Rightarrow (s, P) \overset{\mathcal{O}}{=}_n (t, Q)$$

Proof: Clearly, by definition of $\sqsubseteq^{\mathcal{O}}_0$, $(s, P) \overset{\mathcal{O}}{=}_0 (t, Q)$ is always true.

Now consider the case $n + 1$, where $n \geq 0$.

$$\begin{array}{ll}
 & \{ \text{ By totality of } \twoheadrightarrow \} \\
 & \exists u, R \cdot (t, Q) \twoheadrightarrow (u, R) \\
\Rightarrow & \{ \text{ definition of } \twoheadrightarrow \text{ and } (s, P) \xrightarrow{0}{}^{*} (t, Q) \} \\
 & \exists u, R \cdot \quad \begin{array}{l} (t, Q) \twoheadrightarrow (u, R) \\ (s, P) \twoheadrightarrow (u, R) \end{array} \\
\Rightarrow & \{ \overset{\mathcal{O}}{=}_n \text{ is reflexive } \} \\
 & \exists u, R \cdot \quad \begin{array}{l} (t, Q) \twoheadrightarrow (u, R) \\ (s, P) \twoheadrightarrow (u, R) \\ (u, R) \overset{\mathcal{O}}{=}_n (u, R) \end{array} \\
\Rightarrow & \{ R = S \} \\
 & \exists u, R, S \cdot \quad \begin{array}{l} (t, Q) \twoheadrightarrow (u, R) \\ (s, P) \twoheadrightarrow (u, S) \\ (u, R) \overset{\mathcal{O}}{=}_n (u, S) \end{array} \\
\Rightarrow & \{ \text{ definition of } \sqsubseteq^{\mathcal{O}}_{n+1} \} \\
 & (s, P) \overset{\mathcal{O}}{=}_{n+1} (t, Q)
\end{array}$$

□

Lemma 2.2: For any natural number n, programs P, Q and states s, t:

$$(s, P) \xrightarrow{0}{}^{*} (t, Q) \Rightarrow (s, P) \overset{\mathcal{D}}{=}_n (t, Q)$$

Proof: From the arguments presented in section 11.3.1, it follows that:

$$(s, P) \xrightarrow{0} (t, Q) \Rightarrow (s, P) \overset{\mathcal{D}}{=}_n (t, Q)$$

By definition of transitive reflexive closure, we know that:

$$(s,P) \xrightarrow{0}^{*} (t,Q) \Rightarrow \exists m : \mathbb{N} \cdot m \geq 0 \wedge (s,P) \xrightarrow{0}^{m} (t,Q)$$

To prove the statement, we use induction on m and the fact that $\stackrel{\mathcal{D}}{=}_n$ is reflexive and transitive.

For the base case, when $m = 0$, we get that $(s,P) = (t,Q)$, and hence $(s,P) \stackrel{\mathcal{D}}{=}_n (t,Q)$.

Let us assume that the result holds for a particular value of m. Now consider the case of $m+1$:

$$\begin{array}{ll}
 & (s,P) \xrightarrow{0}^{m+1} (t,Q) \\
\Rightarrow & \exists u,R \cdot (s,P) \xrightarrow{0} (u,R) \wedge (u,R) \xrightarrow{0}^{m} (t,Q) \\
\Rightarrow & \{ \text{ by the inductive hypothesis } \} \\
 & \exists u,R \cdot (s,P) \xrightarrow{0} (u,R) \wedge (u,R) \stackrel{\mathcal{D}}{=}_n (t,Q) \\
\Rightarrow & \{ \text{ previous arguments in section 11.3.1 } \} \\
 & \exists u,R \cdot (s,P) \stackrel{\mathcal{D}}{=}_n (u,R) \wedge (u,R) \stackrel{\mathcal{D}}{=}_n (t,Q) \\
\Rightarrow & \{ \text{ transitivity of } \stackrel{\mathcal{D}}{=}_n \} \\
 & (s,P) \stackrel{\mathcal{D}}{=}_n (t,Q)
\end{array}$$

□

Lemma 3.1: For any natural number n, programs P, Q and state s:

$$(s,P) \xrightarrow{1} (s,Q) \Rightarrow (s,P) \stackrel{\mathcal{O}}{=}_n (s,\#1;Q)$$

Proof: The proof is similar to that of lemma 2.1. We consider two alternative situations, $n = 0$ and $n > 0$. The result is trivially true in the first case. We now consider the case $n+1$, where $n \geq 0$.

$$\begin{array}{ll}
 & \{ \text{ definition of } \twoheadrightarrow \} \\
 & (s,\#1;Q) \twoheadrightarrow (s,Q) \\
\Rightarrow & \{ \text{ premise and definition of } \twoheadrightarrow \} \\
 & (s,P) \twoheadrightarrow (s,Q) \ \wedge \ (s,\#1;Q) \twoheadrightarrow (s,Q) \\
\Rightarrow & \{ \text{ reflexivity of } \stackrel{\mathcal{O}}{=}_n \} \\
 & (s,P) \twoheadrightarrow (s,Q) \ \wedge \ (s,\#1;Q) \twoheadrightarrow (s,Q) \\
 & (s,Q) \stackrel{\mathcal{O}}{=}_n (s,Q) \\
\Rightarrow & \{ \ u = s,\ R = Q \text{ and } S = Q \ \} \\
 & \quad \exists u,R,S \cdot \quad \begin{array}{l} (s,P) \twoheadrightarrow (u,R) \\ (s,\#1;Q) \twoheadrightarrow (u,S) \\ (u,R) \stackrel{\mathcal{O}}{=}_n (u,S) \end{array} \\
\Rightarrow & \{ \text{ definition of } \stackrel{\mathcal{O}}{=}_{n+1} \} \\
 & (s,P) \stackrel{\mathcal{O}}{=}_{n+1} (s,\#1;Q)
\end{array}$$

□

Lemma 3.2: For any natural number n, programs P, Q and state s:

$$(s, P) \xrightarrow{1} (s, Q) \Rightarrow (s, P) \stackrel{\mathcal{D}}{=}_n (s, \#1; Q)$$

Proof: The proof follows directly from the argument given in section 11.3.3 and the definition of $\stackrel{\mathcal{D}}{=}_n$. $\square$

Lemma 4: For any natural number n, programs P, Q and states s, t:

$$(s, P) \sqsubseteq^{\mathcal{D}}_n (s, Q) \iff (s, \#1; P) \sqsubseteq^{\mathcal{D}}_{n+1} (s, \#1; Q)$$

Proof: The proof is based on the definition of the denotational semantics and a number of laws of duration calculus:

$$\begin{array}{cl}
 & (s, P) \sqsubseteq^{\mathcal{D}}_n (s, Q) \\
\iff & \{ \text{ by definition of } \sqsubseteq^{\mathcal{D}}_n \} \\
 & \exists \overleftarrow{vars}, \overrightarrow{vars} \cdot (l = n \wedge [\![s; Q]\!]) \Rightarrow \exists \overleftarrow{vars}, \overrightarrow{vars} \cdot [\![s; P]\!] \\
\iff & \{ \text{ monotonicity of } ⨾ \} \\
 & \qquad \begin{array}{rll} & \exists \overleftarrow{vars}, \overrightarrow{vars} \cdot & (l = 1 \wedge \mathsf{Const}(vars)) ⨾ \\ & & (l = n \wedge [\![s; Q]\!]) \\ \Rightarrow & \exists \overleftarrow{vars}, \overrightarrow{vars} \cdot & (l = 1 \wedge \mathsf{Const}(vars)) ⨾ [\![s; P]\!] \end{array} \\
\iff & \{ \text{ duration calculus and denotational semantics } \} \\
 & \qquad \begin{array}{rll} & \exists \overleftarrow{vars}, \overrightarrow{vars} \cdot & (\lceil\rceil \wedge \overrightarrow{vars} = s(vars)) ⨾ \\ & & (l = 1 \wedge \mathsf{Const}(vars)) ⨾ \\ & & (l = n \wedge [\![Q]\!]) \\ \Rightarrow & \exists \overleftarrow{vars}, \overrightarrow{vars} \cdot & (\lceil\rceil \wedge \overrightarrow{vars} = s(vars)) ⨾ \\ & & (l = 1 \wedge \mathsf{Const}(vars)) ⨾ [\![P]\!] \end{array} \\
\iff & \{ \text{ denotational semantics } \} \\
 & \exists \overleftarrow{vars}, \overrightarrow{vars} \cdot ([\![s; \#1; Q]\!] \wedge l = n + 1) \Rightarrow \exists \overleftarrow{vars}, \overrightarrow{vars} \cdot [\![s; \#1; P]\!] \\
\iff & \{ \text{ by definition of } \sqsubseteq^{\mathcal{D}}_{n+1} \} \\
 & (s, \#1; P) \sqsubseteq^{\mathcal{D}}_{n+1} (s, \#1; Q)
\end{array}$$

$\square$

Lemma 5: If $(s, P) \sqsubseteq^{\mathcal{D}}_n (t, Q)$ and $n > 0$ then:

$$\begin{array}{rcl} \exists u, P', Q' & \cdot & (s, P) \twoheadrightarrow (u, P') \\ & & (t, Q) \twoheadrightarrow (u, Q') \end{array}$$

Proof: Since $\twoheadrightarrow$ is total, we know that for some P', Q', u and v:

$$\begin{array}{c} (s, P) \twoheadrightarrow (u, P') \\ (t, Q) \twoheadrightarrow (v, Q') \end{array}$$

But we can decompose $\twoheadrightarrow$ into a number of $\xrightarrow{0}$ transitions and a $\xrightarrow{1}$ transition:

$$
\begin{aligned}
(s, P) &\xrightarrow{0}{}^{*} (u, P'') \xrightarrow{1} (u, P') \\
(t, Q) &\xrightarrow{0}{}^{*} (v, Q'') \xrightarrow{1} (v, Q')
\end{aligned}
$$

By lemmata 3.2 and 2.2, we can thus conclude that:

$$
\begin{aligned}
(u, P'') &\stackrel{\mathcal{D}}{=}_n (u, \#1; P') \\
(v, Q'') &\stackrel{\mathcal{D}}{=}_n (v, \#1; Q') \\
(u, P'') &\stackrel{\mathcal{D}}{=}_n (s, P) \\
(v, Q'') &\stackrel{\mathcal{D}}{=}_n (t, Q)
\end{aligned}
$$

Hence, by transitivity of $\stackrel{\mathcal{D}}{=}_n$:

$$
\begin{aligned}
(s, P) &\stackrel{\mathcal{D}}{=}_n (u, \#1; P') \\
(t, Q) &\stackrel{\mathcal{D}}{=}_n (v, \#1; Q')
\end{aligned}
$$

But, from the lemma premise we know that $(s, P) \sqsubseteq^{\mathcal{D}}_n (t, Q)$ and thus:

$$
(u, \#1; P') \quad \sqsubseteq^{\mathcal{D}}_n \quad (v, \#1; Q')
$$

Now, if $u \neq v$, the only way in which the above refinement can be satisfied is if:

$$
l = n \wedge [\![\#1; Q']\!] \quad = \quad \mathbf{false}
$$

But it is easy to see that for any program P, $(l = n \wedge [\![P]\!])$ is never **false**. Hence $u = v$.

$\square$

11.4.6 Unifying the Operational and Denotational Semantics

Theorem 1: For any natural number n, programs P, Q and states s, t:

$$
(s, P) \sqsubseteq^{\mathcal{O}}_n (t, Q) \iff (s, P) \sqsubseteq^{\mathcal{D}}_n (t, Q)
$$

In other words, $\sqsubseteq^{\mathcal{O}}_n = \sqsubseteq^{\mathcal{D}}_n$

Proof: We prove this theorem using induction on n.

Base case: $n = 0$. Lemma 1 already guarantees this.

Inductive hypothesis: Let us assume that the theorem holds for n:

$$
(s, P) \sqsubseteq^{\mathcal{O}}_n (t, Q) \iff (s, P) \sqsubseteq^{\mathcal{D}}_n (t, Q).
$$

Inductive case: We would now like to show that:

$$
(s, P) \sqsubseteq^{\mathcal{O}}_{n+1} (t, Q) \iff (s, P) \sqsubseteq^{\mathcal{D}}_{n+1} (t, Q).
$$

We now consider the two directions of the proof separately:

- *Case 1:* $\sqsubseteq^{\mathcal{O}}_{n+1} \subseteq \sqsubseteq^{\mathcal{D}}_{n+1}$. In this part we will be proving that the denotational semantics are *complete* with respect to the operational semantics. Equivalently, it can be seen to show that the operational semantics are *sound* with respect to the denotational semantics.

$$
\begin{array}{ll}
 & (s, P) \sqsubseteq^{\mathcal{O}}_{n+1} (t, Q) \\
\Rightarrow & \{ \text{ definition of } \sqsubseteq^{\mathcal{O}}_{n+1} \} \\
 & \exists u, P'', Q'' \cdot \quad \begin{array}{l} (s, P) \twoheadrightarrow (u, P'') \\ (t, Q) \twoheadrightarrow (u, Q'') \\ (u, P'') \sqsubseteq^{\mathcal{O}}_{n} (u, Q'') \end{array} \\
\Rightarrow & \{ \text{ inductive hypothesis } \} \\
 & \exists u, P'', Q'' \cdot \quad \begin{array}{l} (s, P) \twoheadrightarrow (u, P'') \\ (t, Q) \twoheadrightarrow (u, Q'') \\ (u, P'') \sqsubseteq^{\mathcal{D}}_{n} (u, Q'') \end{array} \\
\Rightarrow & \{ \text{ definition of } \twoheadrightarrow \} \\
 & \exists u, P', P'', Q', Q'' \cdot \quad \begin{array}{l} (s, P) \xrightarrow{0}^{*} (u, P') \\ (u, P') \xrightarrow{1} (u, P'') \\ (t, Q) \xrightarrow{0}^{*} (u, Q') \\ (u, Q') \xrightarrow{1} (u, Q'') \\ (u, P'') \sqsubseteq^{\mathcal{D}}_{n} (u, Q'') \end{array} \\
\Rightarrow & \{ \text{ Lemma 4 } \} \\
 & \exists u, P', P'', Q', Q'' \cdot \quad \begin{array}{l} (s, P) \xrightarrow{0}^{*} (u, P') \\ (u, P') \xrightarrow{1} (u, P'') \\ (t, Q) \xrightarrow{0}^{*} (u, Q') \\ (u, Q') \xrightarrow{1} (u, Q'') \\ (u, \#1; P'') \sqsubseteq^{\mathcal{D}}_{n+1} (u, \#1; Q'') \end{array} \\
\Rightarrow & \{ \text{ Lemma 3.2 } \} \\
 & \exists u, P', Q' \cdot \quad \begin{array}{l} (s, P) \xrightarrow{0}^{*} (u, P') \\ (t, Q) \xrightarrow{0}^{*} (u, Q') \\ (u, P') \sqsubseteq^{\mathcal{D}}_{n+1} (u, Q') \end{array} \\
\Rightarrow & \{ \text{ Lemma 2.2 } \} \\
 & (s, P) \sqsubseteq^{\mathcal{D}}_{n+1} (t, Q)
\end{array}
$$

- *Case 2:* $\sqsubseteq^{\mathcal{O}}_{n} \supseteq \sqsubseteq^{\mathcal{D}}_{n}$. The is the dual of the first case, where we show that the denotational semantics are *sound* with respect to the operational semantics (or, equivalently, that the operational semantics are *complete* with respect to the denotational semantics).

$$
\begin{array}{ll}
 & (s, P) \sqsubseteq^{\mathcal{D}}_{n+1} (t, Q) \\
\Rightarrow & \{ \text{ Lemma 5 } \} \\
 & \exists u, P'', Q'' \cdot \quad \begin{array}{l} (s, P) \twoheadrightarrow (u, P'') \\ (t, Q) \twoheadrightarrow (u, Q'') \\ (s, P) \sqsubseteq^{\mathcal{D}}_{n+1} (t, Q) \end{array} \\
\Rightarrow & \{ \text{ definition of } \twoheadrightarrow \text{ and } \xrightarrow{1} \text{ keeps state constant } \}
\end{array}
$$

$$
\begin{array}{lll}
 & \exists u, P', P'', Q', Q'' \cdot & (s,P) \xrightarrow{0}{}^{*} (u,P') \\
 & & (u,P') \xrightarrow{1} (u,P'') \\
 & & (t,Q) \xrightarrow{0}{}^{*} (u,Q') \\
 & & (u,Q') \xrightarrow{1} (u,Q'') \\
 & & (s,P) \stackrel{\boxed{\mathcal{D}}}{=}_{n+1} (t,Q) \\
\Rightarrow & \{ \text{ Lemma 2.2 } \} & \\
 & \exists u, P', P'', Q', Q'' \cdot & (s,P) \twoheadrightarrow (u,P'') \\
 & & (t,Q) \twoheadrightarrow (u,Q'') \\
 & & (u,P') \xrightarrow{1} (u,P'') \\
 & & (u,Q') \xrightarrow{1} (u,Q'') \\
 & & (u,P') \stackrel{\boxed{\mathcal{D}}}{=}_{n+1} (u,Q') \\
\Rightarrow & \{ \text{ Lemma 3.2 twice } \} & \\
 & \exists u, P', P'', Q', Q'' \cdot & (s,P) \twoheadrightarrow (u,P'') \\
 & & (t,Q) \twoheadrightarrow (u,Q'') \\
 & & (u,P') \stackrel{\mathcal{D}}{=}_{n+1} (u, \#1; P'') \\
 & & (u,Q') \stackrel{\mathcal{D}}{=}_{n+1} (u, \#1; Q'') \\
 & & (u,P') \stackrel{\boxed{\mathcal{D}}}{=}_{n+1} (u,Q') \\
\Rightarrow & \{ \text{ transitivity of } \stackrel{\boxed{\mathcal{D}}}{=}_{n} \} & \\
 & \exists u, P'', Q'' \cdot & (s,P) \twoheadrightarrow (u,P'') \\
 & & (t,Q) \twoheadrightarrow (u,Q'') \\
 & & (u, \#1; P'') \stackrel{\boxed{\mathcal{D}}}{=}_{n+1} (u, \#1; Q'') \\
\Rightarrow & \{ \text{ Lemma 4 and information from previous lines } \} & \\
 & \exists u, P'', Q'' \cdot & (s,P) \twoheadrightarrow (u,P'') \\
 & & (t,Q) \twoheadrightarrow (u,Q'') \\
 & & (u,P'') \stackrel{\boxed{\mathcal{D}}}{=}_{n} (u,Q'') \\
\Rightarrow & \{ \text{ inductive hypothesis and definition of } \twoheadrightarrow \} & \\
 & \exists u, P'', Q'' \cdot & (s,P) \twoheadrightarrow (u,P'') \\
 & & (t,Q) \twoheadrightarrow (u,Q'') \\
 & & (u,P'') \stackrel{\boxed{\mathcal{O}}}{=}_{n} (u,Q'') \\
\Rightarrow & \{ \text{ definition of } \stackrel{\boxed{\mathcal{O}}}{=}_{n+1} \} & \\
 & (s,P) \stackrel{\boxed{\mathcal{O}}}{=}_{n+1} (t,Q) &
\end{array}
$$

This completes the inductive case, and hence the proof.

□

Corollary 1: $P \stackrel{\mathcal{D}}{=} Q \iff P \stackrel{\mathcal{O}}{=} Q$.

Proof: This follows almost immediately from Theorem 1.

$$
\begin{array}{ll}
 & \{ \text{ Theorem 1 } \} \\
 & \forall n : \mathbb{N} \cdot \stackrel{\boxed{\mathcal{D}}}{=}_{n} = \stackrel{\boxed{\mathcal{O}}}{=}_{n} \\
\Rightarrow & \{ \text{ definition of } \stackrel{\boxed{\mathcal{D}}}{=} \text{ and } \stackrel{\boxed{\mathcal{O}}}{=} \} \\
 & \stackrel{\boxed{\mathcal{D}}}{=} = \stackrel{\boxed{\mathcal{O}}}{=} \\
\Rightarrow & \{ \text{ definition of } \stackrel{\mathcal{D}}{=} \text{ and } \stackrel{\mathcal{O}}{=} \} \\
 & \stackrel{\mathcal{D}}{=} = \stackrel{\mathcal{O}}{=}
\end{array}
$$

□

11.5 Conclusions

This chapter has unified two strictly different views of the semantics of Verilog. On one hand we have the denotational semantics which allow us to compare mathematical specifications with a Verilog program. On the other hand we have an operational semantics which describes exactly the way in which the language is interpreted. Essentially, the result is a set of laws which decide the transformation of a Verilog program into a sequential one. This complements the results given in the previous chapter, where the transformations parallelised the code as much as possible. The two approaches complement each other and serve to expose better the dual nature of languages which combine sequential with parallel execution.

Part V

Chapter 12

Conclusions and Future Work

12.1 An Overview

The main goal of this thesis is to show how standard industrial HDLs can be used within a formal framework. Different research may have different motivations for the formalisation of a language semantics. One such motivation can be to formally document the behaviour of programs written in the language. Another motive may be that of providing a sound mathematical basis to serve as a tool for program development and verification. Our choice of the subset of Verilog we choose to formalise indicates that our primary aim is more the second than the first. We aimed at identifying a subset of Verilog with a clear semantics so as to enable us to produce more elegant proofs. In particular, hardware acts upon input varying over time to produce a time dependent output. It is thus desirable to be able to talk about time related properties of such systems.

Algebraic reasoning in arithmetic and analysis has been used effectively by engineers for centuries and we believe that this is also the way forward in hardware engineering. However, one would desire a mathematical interpretation of the language by which to judge these laws. This is the main motivation behind the denotational semantics (chapter 5) and the algebraic laws (chapter 6).

In particular, if a decision procedure can be used to apply these laws effectively towards a particular objective, their use becomes much more attractive. One such application is the compilation procedure presented in chapter 10. Another is the interpretation of the language by a simulator. Although such an interpretation is usually seen as an operational semantics, there is no reason why we should not view the operational transitions as algebraic laws in their own right as shown in chapter 11.

12.2 Shortcomings

The evaluation of this thesis would be incomplete if it lacks to mention its shortcomings. This section attempts to express and discuss the main ones.

Formalised subset of Verilog: The subset of Verilog which has been formalised is rather large. However, the side condition that the programs must not perform any concurrent reading and writing on global variables is rather strong. Certain frequently used programs such as (`@clk v=d or clk`) are not allowed. The main problem is that the condition is not syntactically checkable. Conventions could have been defined to (syntactically) make sure that this condition is always satisfied, for example by ensuring that reading is always performed at the rising edge of a global clock, while writing is always performed at the falling edge. Alternatively, the non-deterministic semantics given in section 5.5.4 could be used to get around the problem. These non-deterministic semantics should be analysed more thoroughly in the future.

Unknown and high impedance values: These values (denoted by `x` and `z` respectively) are frequently used by hardware engineers to model tri-state devices. Our basic Verilog semantics fail to handle these values and although section 5.5.3 discusses possible ways to handle such values and devices, they have not been explored any further in the thesis.

Case studies: The case studies analyed in chapters 7 to 9 are rather small. Larger case studies (such as the correctness of a small processor) need to be performed to judge better the utility of the semantics presented in this thesis.

Mechanical proof checking: A number of rather complex theorems have been proved in this thesis which would have benefitted from a run through a mechanical proof checker. This is discussed in more detail in section 12.3.2.

The results presented thus still need considerable refinement before they can find their place on the hardware engineer's workbench. However, we believe that this thesis is a step towards the use of formal tools and techniques in hardware design.

12.3 Future Work

12.3.1 Other Uses of Relational Duration Calculus

Relational Duration Calculus has been developed with the semantics of Verilog in mind. However, the formulation of the calculus is much more general than this and should be found useful in describing other phenomena.

One of the differences between Verilog and VHDL is the concept of delta delays in VHDL, where signal assignments are postponed until the simulator has to execute a `wait` instruction[1]. On the other hand, variable assignments take place immediately.

Consider the following VHDL portion of code:

```
s<=0; wait for 0ns; s<=1; v=1; w=s and v
```

where `s` is a signal and `v, w` are variables. We would expect `w` to be assigned the value `0` (`0 and 1`). Note that `v` is updated immediately, while `s` is not.

This two-level hierarchy can be captured in Relational Duration Calculus thanks to the typing of the relational chop operator:

[1]In a certain sense, this is quite similar to Verilog non-blocking assignments.

$$
\begin{pmatrix} \overrightarrow{s} = 0 \\ \wedge \lceil\rceil \wedge \\ \mathsf{Const}(\{v, w\}) \end{pmatrix} \overset{\{s,v,w\}}{⨟}
\left(\begin{pmatrix} \overrightarrow{s} = 1 \\ \wedge \lceil\rceil \wedge \\ \mathsf{Const}(\{v, w\}) \end{pmatrix} \overset{\{v,w\}}{⨟} \begin{pmatrix} \overrightarrow{v} = 1 \\ \wedge \lceil\rceil \wedge \\ \mathsf{Const}(\{w\}) \end{pmatrix} \overset{\{v,w\}}{⨟} \begin{pmatrix} \overrightarrow{w} = \overleftarrow{v} \wedge \overleftarrow{s} \\ \wedge \lceil\rceil \wedge \\ \mathsf{Const}(\{v\}) \end{pmatrix} \right)
$$

In fact, this technique should also be applicable to other languages with an inherently built synchronisation mechanism such as BSP. Going one step further, one can construct a language with multiple 'synchronisation layers'. VHDL has only two such layers, the local (variables) and delta level (signals), as does BSP. But one may construct a language with a whole family of such layers to aid programming in, say, a network of distributed systems. The semantics of such a language should be elegantly expressible in Relational Duration Calculus.

12.3.2 Mechanised Proofs

The work presented in this thesis can be pushed forward along various paths. One obvious missing aspect is machine verified proofs of the theorems presented. Ideally, one should provide support for machine proofs in Relational Duration Calculus.

Not much work has yet been done in mechanising Duration Calculus proofs. [SS93] embedded Duration Calculus in PVS and proved a number of number of properties of standard case studies. They have also defined a number of tactics, for instance, to automatically prove statements in a decidable subset of Duration Calculus.

[Pac94] investigated the use of JAPE [SB93], a visual proof calculator, to prove a number of hardware implementations of standard case studies correct. However, the proofs were given with respect to a number of Duration Calculus laws, as opposed to the underlying definitions or axioms. A number of tactics useful in transforming some real-time specifications into a format closer to hardware have also been presented there.

Embedding Relational Duration Calculus and the given Verilog semantics into a standard theorem prover and proving the correctness of the algebraic laws would allow the verification of the proofs given in the thesis. The hardware compilation and simulation semantics could then be encoded as proof tactics.

12.3.3 Real-Time Properties and HDLs

An area which is only briefly touched upon in this thesis is the verification of real-time properties of programs written in a hardware description language. The use of Duration Calculus to specify the semantics of Verilog allows us to handle such proofs in a straightforward manner. Unfortunately, we are not aware of much other work done for this particular style of specification in conjunction with HDLs. It would be interesting to study this area in more detail and see what useful results can be achieved.

12.3.4 Extending the Sub-Languages

Another aspect which should ideally be extended is the sub-language which is compiled into hardware-like format. Other constructs (such as `fork ...join`) and instructions (such as `@v`) can be converted into a hardware-like format without too much difficulty. One would still, however, desire a proof of correctness for these additional transformations.

The same can be said of the comparison between the operational and denotational semantics.

12.4 Conclusion

The main aim of this thesis was to study the development of a formal framework for Verilog, as a typical industrial-strength hardware description language. The main achievement was the building of a basis for such a framework, but more work still needs to be done in order to make this framework robust and general enough to be useful in a real-life hardware development environment.

Bibliography

[Bal93] Felice Balarin. Verilog HDL modelling styles for formal verification. In D. Agnew and L. Claesen, editors, *Computer Hardware Description Languages and their Applications (A-32)*, pages 453–466. Elsevier Science Publishers BV (North-Holland), 1993.

[BB92] J.C.M Baeten and J.A. Bergstra. Discrete time process algebra. In W.R. Cleaveland, editor, *Proceedings of CONCUR '92*, number 630 in Lecture Notes in Computer Science, pages 401–420. Springer, 1992.

[BB93] J.C.M Baeten and J.A. Bergstra. Real time process algebra with infinitesimals. Technical Report P9325, Programming Research Group, University of Amsterdam, October 1993.

[BFK95] Peter T. Breuer, Luis Sanchez Fernandez, and Carlos Delgado Kloos. A simple denotational semantics, proof theory and a validation condition generator for unit-delay VHDL. *Formal Methods in System Design*, 7:27–51, 1995.

[BG96] C. Bolchini and C. Ghezzi. Software systems: Languages, models and processes. In Giovanni de Micheli and Mariagiovanna Sami, editors, *Software Hardware Co-design*, volume 310 of *NATO ASI series (Series E — Applied Sciences)*, pages 397–426. Kluwer Academic publishers (Netherlands), 1996.

[BGM95] E. Börger, U. Glässer, and W. Müller. Formal definition of an abstract VHDL '93 simulator by EA-machines. In C. Delgado Kloos and P.T. Breuer, editors, *Formal Semantics for VHDL*, pages 107–139. Kluwer Academic Press Boston/London/Dordrecht, 1995.

[Bla96] Gerald M. Blair. Verilog — accellerating digital design. *IEE Electronics and Communication Engineering Journal*, 9(2):68–72, 1996.

[Boc82] Gregor V. Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers*, C-31(3):223–231, March 1982.

[Bor95] Editor Dominique Borrione. Formal methods in system design, special issue on VHDL semantics. Volume 7, Nos. 1/2, Aug 1995.

[Bou97] Richard J. Boulton. A tool to support formal reasoning about computer languages. In E. Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97), Enschede, The Netherlands*, number 1217 in Lecture Notes in Computer Science, pages 81–95. Springer, April 1997.

[Bry86] Randal E. Bryant. Can a simulator verify a circuit? In G. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 125–136. Elsevier Science Publisher B.V., 1986.

[BS95] Dominique Borrione and Ashraf Salem. Denotational semantics of a synchronous VHDL subset. *Formal Methods in System Design*, 7:53–71, 1995.

[BSC+94] C. Bayol, B. Soulas, F. Corno, P. Prinetto, and D. Borrione. A process algebra interpretation of a verification oriented overlanguage of VHDL. In *EURO-DAC 1994/EURO-VHDL '94*, pages 506–511. IEEE CS Press, 1994.

[BSK93] P.T. Breuer, L. Sánchez, and C. Delgado Kloos. Clean formal semantics for VHDL. In *European Design and Test Conference, Paris*, pages 641–647. IEEE Computer Society Press, 1993.

[Cam90] Albert Camilleri. Simulating hardware specifications within a theorem proving environment. *International Journal of Computer Aided VLSI design*, (2):315–337, 1990.

[CGH+93] Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. A formal specification model for hardware/software codesign. In *Proceedings of International Workshop on Software-Hardware Codesign*, October 1993.

[CGM86] Albert Camilleri, Mike Gordon, and Tom Melham. Hardware verification using higher order logic. Technical Report 91, University of Cambridge Computing Laboratory, 1986.

[CHF96] Ching-Tsun Chou, Jiun-Lang Huang, and Masahiro Fujita. A high-level language for programming complex temporal behaviours and its translation into synchronous circuits. September 1996.

[CP88] Paolo Camurati and Paola Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *IEEE Computer*, pages 8–19, July 1988.

[Dav93] K.C. Davis. A denotational definition of the VHDL simulation kernel. In P. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of the 11th IFIP WG 10.2 International Conference on Computer Hardware Description Languages and their applications CHDL '93*, pages 523–536, 1993.

[Des96a] Chrysalis Symbolic Design. Design for verification strategies optimize advanced verification techniques, December 1996. available from `http://chrys.chrysalis.com/techinfo/`.

[Des96b] Chrysalis Symbolic Design. Symbolic logic technology, January 1996. available from `http://chrys.chrysalis.com/techinfo/`.

[Des96c] Chrysalis Symbolic Design. What is formal verification, January 1996. available from `http://chrys.chrysalis.com/techinfo/`.

[DHP95] Gert Döhmen, Ronald Herrmann, and Hergen Pargmann. Translating VHDL into functional symbolic finite-state models. *Formal Methods in System Design*, 7:125–148, 1995.

[Fou91] Micheal P. Fourman. Proof and design — machine-assisted formal proof as a basis for behavioural design tools. Technical Report LFCS-91-319, Laboratory for the foundations of Computer Science, The University of Edinburgh, 1991.

[Frä95] Martin Fränzle. On appropiateness of the digitality abstraction taken in duration calculus to implementation developement. ProCoS document, Christian-Albrechts-Universität Kiel, Germany, 1995.

[GG95] D.J. Greaves and M.J.C. Gordon. Checking equivalence between synthesised logic and non-synthesisable behavioural prototypes. Research Proposal, 1995. available from `http://www.cl.cam.ac.uk/users/djs1002/verilog.project`.

[GG98] M.J.C. Gordon and A. Ghosh. Language independent RTL semantics. In *Proceedings of IEEE CS Annual Workshop on VLSI: System Level Design, Florida, USA*, 1998.

[Gol92] Robert Goldblatt. *Logics of Time and Computation*, chapter 6: Temporal Logic. Number 7 in CLSI lecture notes. Center for the Study of Language and Information, second edition, 1992.

[Gol94] Steve Golson. State machine design techniques for Verilog and VHDL. *Synopsys Journal of High-Level Design*, September 1994. Available from `www.synopsys.com`).

[Gol96] Ulrich Golze. *VLSI Chip Design with the Hardware Description Language VERILOG: an introduction based on a large RISC processor*. Springer, 1996.

[Goo93a] K.G.W. Goossens. The formalisation of a hardware description language in a proof system: Motivation and applications. In *Proceedings of the XIII Conference of the Brazilian Computer Society, Florianopolis, Brazil*, September 1993.

[Goo93b] K.G.W. Goossens. Structure and behaviour in hardware verification. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its applications, 6th International Workshop, HUG '93, Vancouver, B.C. Canada*, number 780 in Lecture Notes in Computer Science, pages 73–87, 1993.

[Goo95] K.G.W. Goossens. Reasoning about VHDL using operational and observational semantics. Technical Report SI/RR-95/06, Dipartimento di Scienze dell' Informazione, Università degli Studi di Roma, 'La Sapienza', via Salaria, 113-I-00198, Roma, April 1995.

[Gor95] Mike Gordon. The semantic challenge of Verilog HDL. In *Proceedings of the tenth annual IEEE symposium on Logic in Computer Science (LICS '95) San Diego, California*, pages 136–145, June 1995.

[Gor98] M.J.C. Gordon. Event and cycle semantics of hardware description languages v1.4, January 1998. available from `http://www.cl.cam.ac.uk/users/djs1002/verilog.project/`.

[H+85] C.A.R. Hoare et al. Laws of programming — a tutorial paper. Technical Monograph PRG-45, Oxford University Computing Laboratory, 1985.

[Hay82] John P. Hayes. A unified switching theory with applications to VLSI design. *Proceedings of the IEEE*, 70(10):1140–1151, 1982.

[HB92] Jifeng He and Jonathan Bowen. Time interval semantics and implementation of a real-time programming language. In *Proceedings of the 4th Euromicro Workshop in Computing*, pages 173–192, August 1992.

[HB93] Jifeng He and S.M. Brien. Z description of duration calculus. Technical report, Oxford University Computing Laboratory (PRG), 1993.

[He93] Jifeng He. Synchronous realization of circuits. ProCos document OU HJF 17/1, Oxford University Computing Laboratory, Programming Research Group, Wolfson Building, Parks Road, Oxford, October 1993.

[He94] Jifeng He. From CSP to hybrid systems. In A.W. Roscoe, editor, *A Classical Mind*, chapter 11, pages 171–189. Prentice-Hall, 1994.

[Hea93] Thomas Heath. Automating the compilation of software into hardware. Master's thesis, Oxford University Computing Laboratory, Oxford University, 1993.

[Her88] John Herbert. Formal verification of basic memory devices. Technical Report 124, University of Cambridge Computing Laboratory, 1988.

[HH94] C.A.R. Hoare and Jifeng He. Specification and implementation of a flashlight. ProCoS report, Oxford University Computing Laboratory, 1994.

[HHF+94] Jifeng He, C.A.R. Hoare, Martin Fränzle, Markus Müller-Olm, Ernst-Rüdiger Olderog, Michael Schenke, Michael R. Hansen, Anders P. Ravn, and Hans Rischel. Provably correct systems. Technical report, ProCoS, 1994.

[HHS93a] C.A.R. Hoare, Jifeng He, and A. Sampaio. From algebra to operational semantics. *Information Processing Letters*, 45(2):75–80, February 1993.

[HHS93b] C.A.R. Hoare, Jifeng He, and Augusto Sampaio. Normal form approach to compiler design. *ACTA Informatica*, 30:701–739, 1993.

[HI96] Dang Van Hung and Ko Kwang Il. Verification via digitized models of real-time systems. Research UNU/IIST Report No. 54, The United Nations University, International Institute for Software Technology, P.O. Box 3058, Macau, February 1996.

[HJ94] Jifeng He and Zheng Jianping. Simulation approach to provably correct hardware compilation. In *Formal Techniques in Real-Time and Fault Tolerant Systems*, number 863 in Lecture Notes in Computer Science, pages 336–350. Springer-Verlag, 1994.

[HJ98] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall Series in Computer Science, 1998.

[HMC96] Jifeng He, Quentin Miller, and Lei Chen. Algebraic laws for BSP programming. In *Proceedings of EUROPAR 1996*, number 1124 in Lecture Notes in Computer Science, pages 359–368. Springer-Verlag, 1996.

[HMM83] Joseph Halpern, Zohar Manna, and Ben Moszkowski. A hardware semantics based on temporal intervals. In *10th International Colloqium on Automata, Languages and Programming*, number 154 in Lecture Notes in Computer Science, pages 278–291. Springer-Verlag, 1983.

[HO94] Jifeng He and Ernst-Rüdiger Olderog. From real-time specification to clocked circuit. ProCoS document, Department of Computer Science, Technical University of Denmark, DK-2800, Lyngby, Denmark, 1994.

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

[Hoa90] C.A.R. Hoare. A theory for the derivation of C-MOS circuit designs. In *Beauty is our Business*, Texts and monographs in Computer Science. Springer-Verlag, 1990.

[Hoa94] C.A.R. Hoare. Mathematical models for computer science. Lecture Notes for Marktoberdorf Summer School, August 1994.

[Hol97] Clive Holmes. VHDL language course. Technical report, Rutherford Appleton Laboratory, Microelectronics Support Centre, Chilton, Didcot, Oxfordshire, February 1997.

[HP94] C.A.R. Hoare and Ian Page. Hardware and software: The closing gap. *Transputer Communications*, 2(2):69–90, June 1994.

[HRS94] K.M. Hansen, A.P. Ravn, and V. Stravridou. From safety analysis to formal specification. ProCoS document DTH KMH 1/1, Department of Computer Science, Technical University of Denmark, DK-2800, Lyngby, Denmark, February 1994.

[HZ94] M.R. Hansen and Chaochen Zhou. Duration calculus: Logical foundations. *Formal Aspects of Computing*, 6(6A):826–845, 1994.

[HZS92] Michael Hansen, Chaochen Zhou, and Jørgen Straunstrup. A real-time duration semantics for circuits. In *Proceedings of Tau '92: 1992 Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, ACM/SIDGA*, September 1992.

[IEE95] IEEE. *Draft Standard Verilog HDL (IEEE 1364).* 1995.

[Joy89] Jeffrey J. Joyce. A verified compiler for a verified microprocessor. Technical Report 167, University of Cambridge Computer Laboratory, March 1989.

[JS90] Geraint Jones and M. Sheeran. Circuit design in RUBY. In Jørgen Straunstrap, editor, *Formal Methods for VLSI Design*, pages 13–70. Elsevier Science Publications BV, 1990.

[JU90] Mark B. Josephs and Jan Tijmen Udding. An algebra for delay-insensitive circuits. Technical Report WUCS-89-54, Department of Computer Science, Washington University, Campus Box 1045, One Brookings Drive, Saint Louis, MO 63130-4899, March 1990.

[KAJW93] Sunjaya Kumar, James H. Aylor, Barry W. Johnson, and W.A. Wulf. A framework for software-hardware co-design. *IEEE Computer*, 26(12):39–46, December 1993.

[KB95] Carlo Delgado Kloos and Peter T. Breuer. *Formal Semantics for VHDL*. Number 307 in The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1995.

[KLMM95] D. Knapp, T. Ly, D. MacMillen, and R. Miller. Behavioral synthesis methodology for HDL-based specification and validation. In *Proceedings of DAC '95*, 1995.

[Kroon] Thomas Kropf. Benchmark-circuits for hardware-verification v1.2.1. Technical Report SFB 358-C2-4/94, Universität Karlsruhe, 1996 (updated version).

[KW88] K. Keutzer and W. Wolf. Anatomy of a hardware compiler. In David S. Wise, editor, *Proceedings of the SIGPLAN '88 Conference on Programming Lanugage Design and Implementation (SIGPLAN '88)*, pages 95–104. ACM Press, June 1988.

[Lam85] Leslie Lamport. On interprocess communication. Technical Report 8, Systems Research Center (SRC), December 1985.

[Lam91] Leslie Lamport. The temporal logic of actions (TLA). Technical Report 79, Digital Equipment Corporation Systems Research centre, 1991.

[Lam93] L. Lamport. Hybrid systems in TLA^+. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102. Springer-Verlag, 1993.

[LL95] Yanbing Li and Miriam Leeser. HML: An innovative hardware description language and its translation to VHDL. Technical Report EE-CE9-95-2, School of Electrical Engineering, Cornell University, Ithaca, 1995.

[LMS86] Roger Lipsett, Erich Marchner, and Moe Shahdad. VHDL — the language. *IEEE Design and Test*, 3(2):28–41, April 1986.

[LS93] Leslie Lamport and Merz Stephan. Hybrid systems in TLA^+. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, number 736 in Lecture Notes in Computer Science, pages 77–102. Springer-Verlag, 1993.

[LS94] Leslie Lamport and Merz Stephan. Specifying and verifying fault tolerant systems. In *Formal Techniques in Real-Time and Fault Tolerant Systems*, number 863 in Lecture Notes in Computer Science, pages 41–76. Springer-Verlag, 1994.

[Man81] Zohar Manna. Verification of sequential programs using temporal axiomatization. Technical Report STAN-CS-81-877, Department of Computer Science, Stanford University, September 1981.

[May90] D. May. Compiling occam into silicon. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, University of Texas at Austin Year of Programming Series, chapter 3, pages 87–106. Addison-Wesley Publishing Company, 1990.

[Maz71] Antoni W. Mazwikiewicz. Proving algorithms by tail functions. *Information and Control*, 18:220–226, 1971.

[Mel87] Thomas Melham. Abstraction mechanisms for hardware verification. Technical Report 106, University of Cambridge Computer Laboratory, 1987.

[Mil83] G.J. Milne. Circal: A calculus for circuit description. *Integration VLSI Journal*, 1(2,3), 1983.

[Mil85a] G.J. Milne. Circal and the representation of communication, concurrency and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, 1985.

[Mil85b] G.J. Milne. Simulation and verification: Related techniques for hardware analysis. In C.J. Koomen and T. Moto-oka, editors, *Proceedings of the 7th International Conference on Computer Hardware Design Language Applications (CHDL '85), Tokyo*, pages 404–417, 1985.

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

[Mil91] G.J. Milne. Formal description and verification of hardware timing. *IEEE transactions on Computers*, 40(7):811–826, July 1991.

[Mil94] G.J. Milne. *Formal Specification and Verification of Digital Systems.* McGraw-Hill, 1994.

[MM83] Ben Moszkowski and Zohar Manna. Reasoning in interval temporal logic. Technical Report STAN-CS-83-969, Department of Computer Science, Stanford University, July 1983.

[Mol89] F. Moller. The semantics of CIRCAL. Technical Report HDF-3-89, Department of Computer Science, University of Strathclyde, Glasgow, Scotland, 1989.

[Moo94] J. Strother Moore. A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. *Formal Aspects of Computation*, 6:60–91, 1994.

[Mos84] Ben Moszkowski. Executing temporal logic programs. Technical Report 55, University of Cambridge Computer Laboratory, 1984.

[Mos85] Ben Moszkowski. A temporal logic for multilevel reasoning about hardware. *Computer*, 2(18):10–19, February 1985.

[Mos86] Ben Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[MP81] Zohar Manna and Amir Pnueli. Verification of concurrent programs: the temporal framework. In Robert S. Boyer and J. Strother Moore, editors, *The Correctness Problem in Computer Science*, Int. Lecture Series in Computer Science, pages 215–273. Academic Press, 1981.

[MP87] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. Technical Report STAN-CS-87-1186, Department of Computer Science, Stanford University, October 1987.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 377–408. ACM Press, 1990.

[MP94] Zohar Manna and Amir Pnueli. Models for reactivity. In *3rd International School and Symposium on Formal Techniques in Real Time Systems and Fault Tolerant Systems, Lübeck Germany*, 1994.

[MR93] Paulo C. Masiero and Anders P. Ravn. Refinement of real-time specifications. ProCoS II document, Department of Computer Science, Technical University of Denmark, DK-2800, Lyngby, Denmark, 1993.

[MW84] Zohar Manna and P.L. Wolper. Synthesis of computer processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.

[NH84] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Nor96] Anders Nordstrom. Formal verification — a viable alternative to simulation? In *Proceedings of International Verilog HDL Conference*, February 1996.

[OC95] Serafin Olcoz and Jose Manuel Colom. A colored Petri net model of VHDL. *Formal Methods in System Design*, 7:101–123, 1995.

[Ope93] Open Verilog International. *Verilog Hardware Description Language Reference Manual (Version 2.0)*. Open Verilog, March 1993.

[Pac94] Gordon J. Pace. Duration calculus: From parallel specifications to clocked circuits. Master's thesis, Oxford University Computing Laboratory, 1994.

[Pag93] Ian Page. Automatic systems synthesis: A case study. Technical report, Oxford University Computing Laboratory (PRG), 1993. Available from `http://www.comlab.ox.ac.uk` in `/oucl/hwcomp.html`.

[Pag96] Ian Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[Pag97] Ian Page. Hardware-software co-synthesis research at Oxford. UMIST Vacation School on Hardware/Software Codesign, 1997.

[Pal96] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Prentice Hall, New York, 1996.

[Per91] D.L. Perry. *VHDL*. McGraw-Hill, 1991.

[PH98] Gordon J. Pace and Jifeng He. Formal reasoning with Verilog HDL. In *Proceedings of the Workshop on Formal Techniques in Hardware and Hardware-like Systems, Marstrand, Sweden*, June 1998.

[PL91] Ian Page and Wayne Luk. Compiling occam into field-programmable gate arrays. In Wayne Luk and Will Moore, editors, *FPGAs*, pages 271–283. Abingdon EE&CS books, 1991.

[Poo95] Rob Pooley. Integrating behavioural and simulation modelling. Technical Report ECS-CSG-8-95, Computer Systems Group, Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh, March 1995.

[PR95] P.K. Pandya and Y.S. Ramakrishna. A recursive mean-value calculus. Technical Report TR-95/3, Tata Institute of Fundamental Research, Bombay, India, 1995.

[Pro97] The Verilog Formal Equivalence (VFE) Project. Synthesizable Verilog: Syntax and semantics, 1997. available from `http://www.cl.cam.ac.uk/users/djs1002/verilog.project`.

[Pyg92] C.H. Pygott. Will proof replace simulation? In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 21–33. Prentice-Hall International, 1992.

[Qiw96] Xu Qiwen. Semantics and verification of extended phase transition systems in duration calculus. Research UNU/IIST Report No. 72, The United Nations University, International Institute for Software Technology, P.O. Box 3058, Macau, June 1996.

[Rav95] Anders P. Ravn. Design of embedded real-time computing systems. Technical Report ID-TR:1995-170, Department of Computer Science, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, October 1995.

[RH86] A.W. Roscoe and C.A.R. Hoare. The laws of occam programming. Technical Monograph PRG-53, Oxford University Computing Laboratory, 1986.

[RH88] A.W. Roscoe and C.A.R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60(2):177–229, September 1988.

[Ris92] Hans Rischel. A duration calculus proof of Fisher's mutual exclusion protocol. ProCoS document DTH HR 4/1, Department of Computer Science, Technical University of Denmark, DK-2800, Lyngby, Denmark, December 1992.

[RK95] Ralf Reetz and Thomas Kropf. A flowgraph semantics of VHDL: Toward a VHDL verification workbench in HOL. *Formal Methods in System Design*, 7:73–99, 1995.

[Rus95] David M. Russinoff. A formalization of a subset of VHDL in the Boyer-Moore logic. *Formal Methods in System Design*, 7:7–25, 1995.

[Sam93] A. Sampaio. An algebraic approach to compiler design. Technical Monograph PRG-110, Oxford University Computing Laboratory, 1993.

[SB93] Bernard Sufrin and Richard Bornat. The gist of JAPE. Technical report, Oxford University Computing Laboratory (PRG), 1993. Available from `http://www.comlab.ox.ac.uk/` in `oucl/bernard.sufrin/jape.shtml`.

[Sch96] Michael Schenke. Transformational design of real-time systems. *Acta Informatica*, 1996. To appear.

[Sch97] Michael Schenke. Modal logic and the duration calculus. To appear, 1997.

[SD97] Michael Schenke and Michael Dossis. Provably correct hardware compilation using timing diagrams. Available from `http://semantic.Informatik.Uni-Oldenburg.DE/persons/michael.schenke/`, 1997.

[SHL94] Morten Ulrik Sørensen, Odd Eric Hansen, and Hans Henrik Løvengreen. Combining temporal specification techniques. ProCoS document DTU MUS 1/2, Department of Computer Science, Technical University of Denmark, DK-2800, Lyngby, Denmark, May 1994.

[Smi96] Douglas J. Smith. VHDL and Verilog compared and contrasted — plus modelled example written in VHDL, Verilog and C. In *Proceedings of the 33rd Design Automation Conference (Las Vegas NV, USA)*, pages 771–776, 1996.

[SMSV83] R.L. Schwartz, P.M. Melliar-Smith, and F.H. Vogt. An interval based temporal logic. In *Proceedings of the second annual ACM symposium on Principles of Distributed Computing, Montreal, Quebec, Canada*, pages 173–186, 1983.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual (2nd edition)*. Prentice Hall International, 1992.

[Spi97] Michael Spivey. Deriving a hardware compiler from operational semantics. *submitted to ACM TOPLAS*, 1997.

[SS93] Jens U. Skakkebaek and N. Shankar. A duration calculus proof checker: Using PVS as a semantic framework. Technical Report SRI-CSL-93-10, SRI, 1993.

[SS94] Jens U. Skakkebaek and N. Shankar. Towards a duration calculus proof assistant in PVS. In H. Langmaack, W.P. Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault Tolerant Systems*, number 863 in Lecture Notes in Computer Science, pages 660–679. Springer-Verlag, 1994.

[SSMT93] E Sternheim, R. Singh, R. Madhavan, and Y. Trivedi. *Digital Design and Synthesis with Verilog HDL*. Automata Publishing Company, 1993.

[Ste97] Daryl Stewart. Modelling Verilog port connections, 1997. Available from `http://www.cl.cam.ac.uk/` in `/users/djs1002/verilog.project/`.

[SW74] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Programming Research Group, Oxford University Computing Laboratory, 1974.

[Tan84] A.S. Tanenbaum. *Structured Computer Organization (3rd edition)*. Prentice-Hall International, 1984.

[Tan88] Tong Gao Tang. A temporal logic language for hardware simulation, specification and verification. Technical Report CMU-CS-88-194, Department of Computer Science, CMU, Pittsburgh, September 1988.

[Tas90] John Peter Van Tassel. The semantics of VHDL with VAL and HOL: Towards practical verification tools. Technical Report 196, University of Cambridge Computer Laboratory, June 1990.

[Tas92] John Peter Van Tassel. A formalization of the VHDL simulation cycle. Technical Report 249, University of Cambridge Computer Laboratory, March 1992.

[TH91] John Peter Van Tassel and David Hemmendinger. Toward formal verification of VHDL specifications. In Luc Claesen, editor, *Applied Formal Methods for VLSI design*, pages 399–408. Elsevier Science Publications, 1991.

[Tho95] Donald E. Thomas. *The Verilog HDL*. Kluwer Academic, Boston, Massachusetts, second edition, 1995.

[Win86] Glynn Winskel. Models and logic of MOS circuits. In *International Summer School on Logic of Programming and Calculi of Discrete Systems*, Marktoberdorf, Germany, 1986.

[Win87] Glynn Winskel. Relating two models of hardware. In D.H. Pitt, A. Poigné, and D.E. Rydeheard, editors, *Category Theory and Computer Science (Edinburgh, U.K., September 1987)*, number 283 in Lecture Notes in Computer Science, pages 98–113. Springer-Verlag, 1987.

[WMS91] Philip A. Wilsey, Timothy J. McBrayer, and David Sims. Towards a formal model of VLSI systems compatible with VHDL. In A. Halaas and P.B. Denyer, editors, *VLSI '91*, pages 225–236. Elsevier Science Publishers BV (North-Holland), Amsterdam, The Netherlands, August 1991.

[Wol93] W.H. Wolf. Software hardware co-design. *Special Issue of IEEE Design and Test*, 10(3):5, September 1993.

[XJZP94] Yu Xinyao, Wang Ji, Chaochen Zhou, and Paritosh K. Pandya. Formal design of hybrid systems. Research UNU/IIST Report No. 19, The United Nations University, International Institute for Software Technology, P.O. Box 3058, Macau, 1994.

[Yoe90] Michael Yoeli. *Formal Verification of hardware design*, chapter 6: Temporal Logic, pages 159–165. The Institute of Electrical and Electronics Engineers, Inc, 1990.

[ZH92] Chaochen Zhou and C.A.R. Hoare. A model for synchronous switching circuits and its theory of correctness. *Formal Methods in System Design*, 1(1):7–28, July 1992.

[ZH96a] Chaochen Zhou and Michael R. Hansen. Chopping a point. Technical report, Department of Information Technology, Technical University of Denmark, March 1996.

[ZH96b] Chaochen Zhou and Michael R. Hansen. Chopping a point. In He Jifeng, John Cooke, and Peter Wallis, editors, *BCS-FACS 7th Refinement Workshop*, Electronic Workshops in Computing, pages 256–266. Springer-Verlag, 1996.

[ZHK96] Ping Zhou, Jozef Hooman, and Ruurd Kuipei. Compositional verification of real-time systems with explicit clock temporal logic. *Formal Aspects of Computing*, 8(3):294–323, 1996.

[Zho93] Chaochen Zhou. Duration calculi: An overview. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Formal Methods in Programming and their Applications*, number 735 in Lecture Notes in Computer Science, pages 256–266. Springer-Verlag, 1993.

[ZHR91] Chaochen Zhou, C.A.R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[ZHRR91] Chaochen Zhou, Michael R. Hansen, Anders Ravn, and Hans Rischel. Duration specifications for shared processors. In J. Vytopil, editor, *Formal Techniques in Real Time and Fault Tolerant Systems*, number 571 in Lecture Notes in Computer Science, pages 21–32. Springer-Verlag, 1991.

[ZHX95a] Chaochen Zhou, Dang Van Hung, and Li Xiaoshan. A duration calculus with infinite intervals. Research UNU/IIST Report No. 40, The United Nations University, International Institute for Software Technology, P.O. Box 3058, Macau, 1995.

[ZHX95b] Chaochen Zhou, Dang Van Hung, and Li Xiaoshan. A duration calculus with infinite intervals. In Horst Reichel, editor, *Fundamentals of Computation*, number 965 in Lecture Notes in Computer Science, pages 16–41. Springer-Verlag, 1995.

[ZRH93] Chaochen Zhou, A.P. Ravn, and M.R. Hansen. An extended duration calculus for hybrid real-time systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rishel, editors, *Hybrid Systems*, number 736 in Lecture Notes in Computer Science, pages 36–59. Springer-Verlag, 1993.

[ZX94] Chaochen Zhou and Li Xiaoshan. A mean-value duration calculus. In A.W. Roscoe, editor, *A Classical Mind*. Prentice-Hall, 1994.

[ZX95] Chaochen Zhou and Li Xiaoshan. A mean-value duration calculus. Research UNU/IIST Report No. 5, The United Nations University, International Institute for Software Technology, P.O. Box 3058, Macau, March 1995.