Developing Semantics of Verilog HDL in Formal Compositional Design of Mixed Hardware / Software Systems

PhD Thesis

Jordan Dimitrov

Software Technology Research Laboratory

De Montfort University

2002

Abstract

The design and analysis of embedded, mixed hardware/software systems, such as PC cards, application specific hardware, m- and e-commerce devices, mobile telecommunication infrastructure and associated software drivers, is hard.

An important issue for correct codesign is the search for *a highly compositional and unifying formal approach* that crosses the hardware/software boundaries and enables us to keep up with the fast growth in the complexity and variety of electronic devices and their associated software.

Hardware/software codesign is a relatively new discipline interconnecting several other fields of research such as Electronics Engineering and Computer Science with the earliest reference to codesign dated back to 1992.

In this thesis, I describe an integrated compositional framework for codesign of mixed hardware/software systems, together with its underpinning theory of semantics and refinement.

My work integrates formal methods into the design process and the focus of the thesis

is on refinement from a formal specification into a formal hardware part and a formal software part.

Central to my methodology is that the synthesis and design start with a single high-level abstract specification which captures the desired behaviour(s) of the system. Decisions are then taken through *correctness preserving* refinement steps.

I have given formal semantics to Verilog — a Hardware Description Language (HDL) conceived in and extensively used by the hardware industry — in both *denotational* (in specification-oriented style) and *operational* terms and my work on Verilog enables me to blend existing and commercially available hardware synthesis tools and methodologies into my formal framework. This has the benefit of linking software development with hardware development in an integrated fashion and therefore span the gap between hardware and software formally.

The equivalence between the two forms of semantics is proven and a set of generic refinement laws is presented. A detailed case-study of a smart card application of the Rivest Shamir Adleman (RSA) encryption algorithm is provided to evaluate my approach.

Declaration

The thesis presented here is mine and original. It is submitted for the degree of Doctor of Philosophy at De Montfort University. The work was undertaken between September 1998 and May 2002.

Leicester, 2002

Contents

Acknowledgements

Many thanks must go to:

- Prof. Hussein Zedan for his inspirational leadership and full support.
- Dr. Antonio Cau for the patience with which he checked and corrected many technical errors in the text.
- Dr. Ben Moszkowski and Jifeng He for hours of discussions and advice.
- My wife Maria for the constant belief in the project.
- All our colleagues at the STRL for the valuable discussions during all these years.

To all of you

Thank You!

Publications

During the duration of the project several publications presented my results to the academic collegium. These are the following:

1. Cau A., Hale R., Dimitrov J., Zedan H., Moszkowski B., Manjunathaiah M., Spivey M. *A Compositional Framework for Hardware/Software Co-Design*, in Camposano R., Wolf W. (eds.) *Design Automation for Embedded Systems*, Kluwer, 2002.

2. Dimitrov J. *Operational Semantics for Verilog*, in Proceedings of APSEC 2001, IEEE, Macau, Dec 2001.

3. Dimitrov J. *Interval Temporal Logic (ITL) semantics for Verilog*, IEE event on Hardware-Software Co-Design, IEE, London, 8th December 2000.

4. Dimitrov J. *Compositional reasoning about events in Interval Temporal Logic*, in Proceeding of JCIS 2000, 675-678, Association for Intelligent Machinery, Atlantic City, Feb–Mar 2000.

Chapter 1

Overview

1.1 Introduction

The design and analysis of embedded, mixed hardware/software systems, such as PC cards, application specific hardware, m- and e-commerce devices, mobile telecommunication infrastructure and associated software drivers, is hard. A major reason for this is the ever increasing complexity of hardware and software systems coupled with the historical divide between hardware and software design. There are many possible reasons for choosing a mixed hardware/software, also called heterogeneous, implementation of a system. Very often conflicting goals and trade-offs have to be considered and the best compromise between them must be found. Some typical aspects which have to be balanced are performance, cost, flexibility, distribution, power consumption, size and fault tolerance.

An important issue for correct codesign is the search for *a highly compositional and unifying formal approach* [34] that crosses the hardware/software boundaries and enables us to keep up with the fast growth in the complexity and variety of electronic devices and their associated software. An approach is *compositional* if it includes any method by which

a) properties of a system as a whole can be inferred from properties of its components, without additional information about the internal structure of those components [19].

b) requirements that a system must meet are transformed into requirements towards its components.

Such an approach will allow us to compose complex systems out of simpler and/or already existing sub-systems, i.e. simplification, re-use, re-engineering and bottom-up design, and at the same time infer requirements and specifications for sub-systems, i.e. top-down design.

More often than not embedded systems work in "hostile" environment as *critical* sub-systems. Thus we need to be able to guarantee their correct behaviour formally; hence the formal underpinning is central. In the software and even in the hardware industry, *simulation* has often been considered synonymous with *verification*. Although many formal techniques are now beginning to find their rightful place, usually the design process still consists of obtaining an implementation from an informal specification without the use

of any formal design techniques. Both hardware and software are then simulated for a number of inputs, an approach known as *co-simulation* [77, 2, 23]. Bugs discovered are removed and the simulation process is repeated over again.

I am convinced that some degree of rigour must be incorporated in the process described above and my work is a step in this direction.

1.2 Scope of the Thesis and Related Work

Hardware/software codesign is a relatively new discipline interconnecting several other fields of research such as electronic engineering and computer science. The earliest reference to codesign is dated 1992 at the First International Workshop on Hardware/Software Codesign [95].

In this thesis, I describe an integrated compositional framework for codesign of mixed hardware/software systems, together with its underpinning theory of semantics and refinement. I advocate *refinement* as my prime design method and I build my work on previous research in refinement of software [88, 13]. Here I focus more on the formal refinement and development of hardware. The codesign process as a whole is soundly based upon my unifying semantics which crosses the boundaries between software and hardware in a seamless way. A unique characteristic of my framework is that it can validate and analyse system's behaviours within a *single* logical formalism, namely Interval Temporal Logic (ITL) [65, 63, 64] and its executable subset, Tempura [64, 93]. Note that the refinement

process is an interactive and thus cannot be fully automated.

My approach is inspired by existing codesign systems, such as SpecC [27], Polis [3] and the Lyngby Co-Synthesis System (LYCOS) [55]. The traditional design flow is that a project starts with *Informal Specification*, also called requirement, which defines the behaviour and the functionality of the product. Immediately after the specification, a designer should split the application into *hardware* and *software* [26]. According to the design flow given in [71], designers have to transform the *Behavioral* description into *Register Transfer Layer (RTL)* using high level synthesis tools. At every step of the design process, simulations and tests are performed to check the correctness of the transformations with respect to the requirements. Although these tests can be automated to a considerable degree, there are many cases when testing only does not provide the necessary level of correctness and trustworthiness. More often than not, crucial test cases are overlooked which, in the case of a critical system, may result in human lives and/or money being lost.

My work integrates formal methods into the design process and the focus of the thesis is on refinement from a formal specification into a formal hardware part and a formal software part.

Existing codesign systems, such as Polis [3] and LYCOS [55], include some formal verification capability, which is most often achieved by use of an external tool, such as a model-checker. The model-checker can only be used when the design is already quite concrete and such an approach cannot maintain the integrity of the whole design. In

contrast, my approach enforces correctness of the design process by working entirely within a formal system.

There have been successful hardware/software verification efforts in academia and more recently in industry. The majority have used model-checking techniques [17, 49, 38], but also for example functional calculi [8, 44] and Abstract State Machines [7], and recently more powerful tools such as Higher-Order Logic (HOL) [31] have been gaining ground.

There is an increasing industrial interest in ITL, for example Verisity has adopted concepts from ITL in their *Temporal e* language [37]. IBM has introduced a temporal logic called *Sugar* [4] containing ITL-like operators which are targeted at making the logic more usable to design engineers.

1.3 Original Contribution

Central to my methodology is that a synthesis and design start with a single high-level abstract specification which captures the desired behaviour(s) of the system. Design decisions are then taken through *correctness preserving* refinement steps.

I choose ITL as my main behaviour describing formalism. All desired system's properties can be *compositionally* verified using the ITL's compositional proof rules in *assumption/commitment* style (see section 3.2) which make no distinction between software or hardware. Again using sound refinement, the ITL specification can then be composi-

tionally refined into a set of executable Tempura modules and then it can be simulated and analysed within Tempura, i.e. simulation and refinement are tightly integrated in my approach whereas behavioural patterns that emerge during the simulation can help with choosing the next refinement step.

Then I can analyse the set of executable Tempura formulas by a set of techniques, including quantitative and statistical data gathering [66], and based on this, I can select which Tempura formula (or module) will be implemented in hardware and which in software. The result of this phase is a partitioning into two clusters of modules. These will be considered best implemented in hardware and software, respectively.

The interface(s) between these modules will largely depend on the target architecture and its underlying application. Fundamental to my approach is the ability to formulate and compositionally prove an *interface theorem* by using the ITL proof rules, even before the hardware/software partitioning decision has been taken and in contrast to the commonly used ad-hoc techniques. By adopting a unifying formalism such as ITL I have increased the confidence level in the partitioning process.

I have given formal semantics to Verilog [30, 71, 80, 29, 1], my *Hardware Description Language (HDL)* of choice, in both *denotational* (in the form of specification oriented) and *operational* terms [21, 22]. These two reflect the duality of the usage of specification languages, i.e. I need to both describe *properties* and *machines* which implement, or compute, these properties. I have used these semantics for Verilog to formally underpin the refinement transformation from an abstract ITL/Tempura specification into Verilog as

well as the refinement from behavioural to RTL specification within Verilog itself.

My work on Verilog enables me to blend some commercially available hardware synthesis tools and methodologies into my formal framework. This has the benefit of linking software development with hardware development in an integrated fashion and therefore span the gap between hardware and software formally.

The unifying semantics for Verilog allows me to break through yet another barrier — the semantic differences between the Behavioural and the RTL parts of the hardware development process. I can now formally prove if an RTL specification refines, i.e. implements, its Behavioural specification.

On a purely technical side, I show how I can incorporate memory variables into Tempura, i.e. I conservatively extend the language of Tempura into $\mathcal{L}_T+$ [1] and prove some properties about these memory variables.

My Operational semantics for Verilog, unlike most of the proposed semantics in the literature, captures both behavioural and RTL constructs and is *fully parallel*. I prove the equivalence between the operational and the denotational semantics and this guarantees the uniformity of my work.

[1] see section 3.4.3 on page 42

1.4 Thesis Outline

Chapter 2 gives a review of the subject, chapter 3 focuses on my computational model which serves as an architecture for hardware/software codesign and within it section 3.2 describes my specification language of ITL/Tempura and section 3.6 presents a set of practical refinement rules. In chapter 4 I develop my denotational semantics for Verilog, with the operational semantics given in chapter 5 and, crucially, the equivalence between these two formalisms is presented in chapter 6. Finally a case study of smart card application is given in chapter 7.

Chapter 2

Hardware/Software Codesign: A Review

I give an overview of hardware/software codesign as a discipline. Some historical remarks put the subject into context and show the progress of the development in the field. I give a critique of the early ways of constructing mixed hardware/software systems and I show how current projects demand new approaches and pose new challenges. I end with some cutting-edge techniques.

2.1 The Early Approach

Hardware/software codesign as a term was invented in 1992 at the First International Workshop on Hardware/Software Codesign [95]. Figure 2.1 gives a typical design flow

widely adopted at that time.

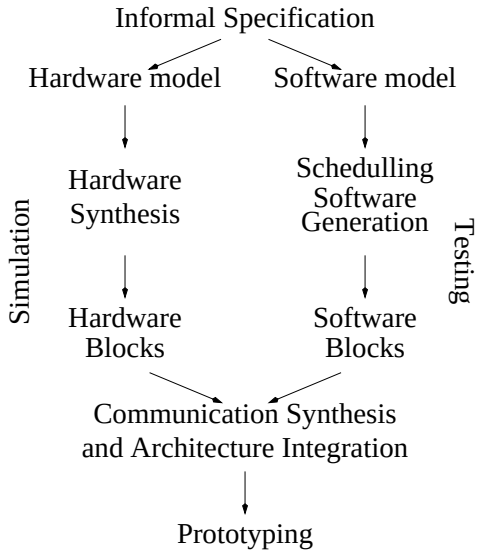


Figure 2.1: Traditional Design Flow in Codesign

According to this traditional design flow, a project starts with an Informal Specification, also called requirement, which defines the behaviour and the functionality of the product. Immediately after the specification, a designer would split the application into hardware and software which implies that several design decisions must have been taken at this point. These include:

- the underlying architecture has been fixed;
- the functionality and the behaviour of the whole system have been split between the software and the hardware into different modules;
- the communication protocol between the system modules has been chosen.

Once the functionality of the system has been split between the hardware and the software parts, the development of these starts. It goes through modelling, hardware synthesis or software generation, and, as a result of that, hardware/software blocks are obtained.

This is followed by merging system blocks using the architectural and communicational paradigms chosen at the point of modelling and only after that we can build a working prototype of the system as a whole.

Many problems were encountered while using this design flow. Most of them came from the very early design decisions taken in accordance to it. As I mentioned above, the communication between the different system modules and the architecture of the whole product are fixed early in the design process without any validation of their suitability. The hardware/software split is also decided and engineered without any check. Most of these crucial choices are underpinned with intuition and best practices rather than rigid reasoning.

Despite the undergoing testing and simulation during the development of the hardware and software modules, the first place where the system, as a whole, is assembled and tested and/or simulated is at the point of the communication synthesis and architecture integration. Only at this moment, very late in the design cycle, one is able to check if the design decisions taken at the very beginning are correct with respect to the original system's specification which, in the mean time, may have been changed.

2.2 The Faster, the Better

Historically, hardware/software codesign was mainly viewed as a methodology for developing hardware accelerators, say in a Fast Fourier Transformation (FFT) application [9]. The typical architecture of these early days [45] is sketched in Figure 2.2.

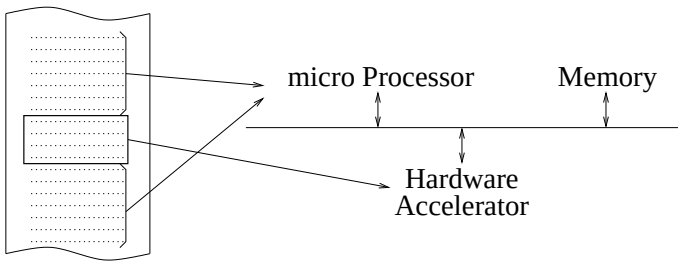


Figure 2.2: Accelerator Architecture

Here we have a piece of software code that needs to be accelerated. The design process consists of finding these parts of the code that are most suitable for acceleration and placing them into a hardware accelerator. In this setup, the program is very simple, sequential piece of code, the underlying architecture is a simple bus with co-processor communication and the hardware/software partitioning is *static*. The challenge here was to evaluate the best configuration, in terms of speed of execution, between the code on the micro processor and the hardware co-processor. Viewed this way, the problem reduces down to granularity and optimisation and has already been solved by large in previous research as part of the LYCOS project [47, 48, 55, 96]. LYCOS relies on, what essentially are granules of computation, *Basic Scheduling Blocks (BSBs)* that may be moved

between hardware and software. Then, the partitioning algorithm PACE is used to obtain fast functional partitions.

2.3 Embedded Systems

An obvious application of hardware/software codesign is development of embedded, or even distributed embedded systems [11, 46, 51, 52]. A typical application would involve *heterogeneous components, multi-language* and *multi-model specifications*, as well as complex target *architectures*. Figure 2.3 gives an idea of a typical application involving such a model.

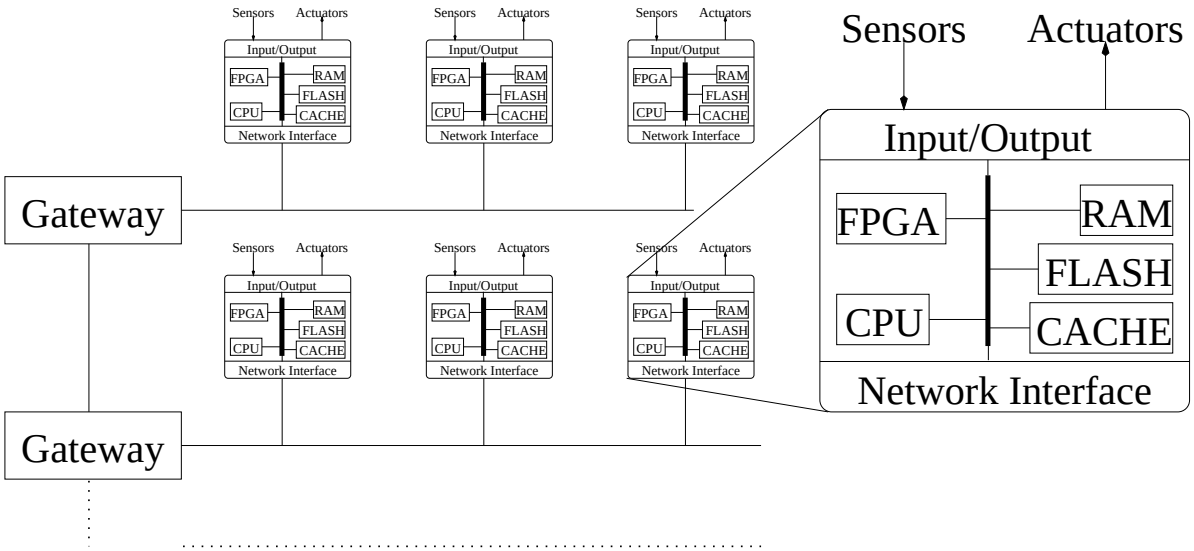


Figure 2.3: Distributed Embedded System

All nodes are interconnected and co-operate in a tandem. Sensors and actuators can

be connected to both digital and analogue input/output sub-systems. The internal components of a node communicate via a network, e.g. a bus. Field Programmable Gate Arrays (FPGAs) are normally reconfigured on the fly, thus making it possible for the system to adapt to changes in its environment. Network Interface provides connectivity with other nodes and is a basis for the distributed computational model.

Many applications have very complex architectures. As shown in Figure 2.3, different tasks can be distributed between many networked *Systems on Chip* and often actuators controlled by one of the nodes depend on information collected by sensors at others.

2.3.1 Mixed Hardware/Software Components

There are many possible reasons for choosing a mixed hardware/software, also called heterogeneous, implementation of a system [74, 24]. Very often there are conflicting goals, and trade-offs have to be made to find the best compromise between them. Some typical aspects which have to be balanced are:

Performance More often than not an embedded system must perform in hard real time. Depending on the application, there might not be any processors available with sufficient performance. Then it becomes necessary to design hardware, where the parallelism can be exploited to gain enough computational power. A typical example would be a Joint Photographic Experts Group (JPEG) encoder [12].

Cost Constructing custom built hardware or using high-end microprocessors is usually

quite expensive, and for a system there is no need to perform quicker than the requirements. Implementing parts of a system in software can significantly reduce the product cost. Using just such an argument many hardware companies introduced the so called "soft modems", also known as Host Controlled Modems[1], which lack the crucial Universal Asynchronous Receiver Transmitter (UART) hardware component and rely on the host computer to perform the UART functions via specialised modem drivers.

Flexibility If a part of the behaviour is likely to be modified after the system is in operation, or if several versions of the same system are planned, it is important to allow changes to be made as easily as possible. This might be an argument for choosing to implement the parts which need to be changed in software and/or reconfigurable FPGAs. Many embedded devices in telecommunication applications use firmware and they are normally being upgraded regularly.

Distribution In some situations the use of a heterogeneous architecture is dictated by the environment. For instance, it could be the case that the sensors and actuators of the system are geographically dispersed, which motivates a distributed architecture where the computing resources are best placed close to the related parts of the environment. Automotive applications follow similar distribution.

[1]Initially, all soft modems supported MS Windows only, hence the alias WinModems. However, this trend has now been broken and LinModems for Linux are also known.

Power Consumption Modern microprocessors run at ever-increasing clock frequencies, and since the frequency is a factor which contributes very much to the power consumption, considerable savings can be made by using an Application Specific Integrated Circuits (ASIC) that runs at a lower clock frequency, but still reaches the same performance through increased parallelism. Mobile computing has very strict requirements on the consumption.

Weight and/or Size Many embedded systems are in some sense portable, and then the weight and/or the size of the product becomes important. This is, for instance, the case for mobile telephones, but also in aerospace and automotive applications. If weight is an issue, it can be desirable to integrate as much functionality as possible into a smaller piece of hardware.

Fault Tolerance Many embedded systems are safety-critical, and they must function, at least partially, even under severe disturbances. Therefore, it might be necessary to duplicate functionality, and implement the same part in several different technologies to reduce the risk of systematic errors which might make all components of a certain kind faulty.

The list is by no means complete. Many other parameters could be considered for particular applications which shows the complex character of the development process from the very early stages to the very latest.

2.3.2 Multi-Language and Multi-Model Specifications

The choice of the underlying paradigm is a crucial one for many embedded applications. Nowadays, designers can select from many languages, both for describing hardware and software, levels of abstraction and communication paradigms [11]. Often, the development process involves shifts from one to another. Let us look briefly into some typical examples when a shift in the paradigms is bound to occur during the development of the system.

- Consider an implementation of a popular network protocol [18]. If the underlying physical transport layer is asynchronous and unreliable and in the same time the application requires synchronous channel for communication, then the obvious transition from synchronous to asynchronous communication has to occur at some stage of the design process.

- Typically, designing a system in Verilog HDL [71, 29] starts with a high level *Behavioural* description. However, the desired specification is a fairly low level *RTL* design and the final goal is an even lower level *Structural* design. The transition between RTL and Structural, or gate level, has already been automated and poses little concern. On the other hand, the shift between a Behavioural and RTL specification is normally underpinned by best practice and intuition.

Obviously, the problems of choosing the correct paradigm and effectively switching between different levels of abstraction are significant. The success of a project often

depends on the *correctness* of these transitions.

2.3.3 Architecture

Architecture selection is an important part of the development process and many researchers position it at the moment of system integration [25]. The designer must carefully select the following aspects of application's architecture.

- A set of components, albeit hardware or software, each one of which has its behaviour and interface specified in a common formalism.
- An inter-networking media, such as channels, buses, network controllers.
- A well defined protocol by which all components communicate and co-operate.

Each one of these elements is crucial and its selection must be verifiable.

One may view the architecture selection as a mapping process which takes the functionality of the system and maps it down to a set of (predefined) components. This normally is justified by the desired manageability of the mapping process. In other cases, the architecture may be limited to a library of predefined components due to vendor restrictions or interfacing constraints. Memory hierarchy or an I/O subsystem design based on standard components has been successfully automated using this technique. Different approaches work on retargetable compilation [79], or on a very abstract formulation of partitioning for co-design [42, 43, 69, 81]. The structure of the application specific hardware components, on the other hand, is generally much less constrained.

I would advocate the idea of stepwise calculative refinement of the system, which in effect helps me to obtain a provable[2] architecture.

2.4 Design Constraints and Requirements

In addition to the aspects of components design given in section 2.3.1, there are some other design constraints that one should consider when developing and/or maintaining a mixed hardware/software system. All of these are direct consequences of a compositional approach which should be adopted if one wishes to explore the full benefits of the codesign.

Legacy Systems The benefits of the compositional codesign become highly visible when it comes to evolving legacy systems [84, 85, 24] and bringing them up to date with evolving needs and/or evolving technology. Let us consider the following example for illustrative purposes. Suppose we were given some requirements which can only be met by using ASIC module because the available processors are too slow and/or too expensive for a software implementation. The system is developed using compositional codesign and deployed. With time, there are two possible scenarios which could lead to a need for redesign.

- Relaxed initial timing requirements.

[2] with respect to a formal requirement

- Improved and/or cheaper Central Processing Unit (CPU) technology.

Any of these could trigger the removal of the customised hardware component and replacing it with equivalent software package. This inevitably gives a cheaper and a smaller final product.

Obviously, the need for replacing a software module with a quicker hardware counterpart is also possible.

Time to Market It is commonly accepted that designing with reusable components reduces the necessary time to market [72]. In this context, compositional co-design has a huge role to play since it allows us to easily incorporate components into both hardware and software contexts.

Of course, the list of the above design requirements could be extended. The variety and complexity of those and the previously listed in section 2.3.1 tell us that the initial methods used when codesign meant little more than simple hardware accelerator are inadequate nowadays.

2.5 Modern Trends in Codesign

The newest trends in the field of hardware/software codesign include some of the following issues. All of them attempt to tackle the design of embedded systems and improve on aspects like flexibility and adaptability, or explore new technologies for building more

compact systems.

2.5.1 Temporal and Spatial Partitioning

Consider again the simple idea of a hardware accelerator given in section 2.2. Here we will build on this via a dynamic reconfiguration "on the fly" of parts of the software code into an FPGA accelerator. As shown on Figure 2.4, we would like to use the reconfigurable FPGA dynamically and build the accelerator for the code when it is needed rather than at compile time. This idea does not give anything new in terms of a underlying architecture but does improve on flexibility. Obvious benefits are

- improved utilisation of the reconfigurable datapath in the FPGA,
- reduced size of the real hardware and
- lower cost and power consumption as a result of reduced hardware.

Most importantly, this allows the system to adapt to specific environments by selecting parts of the software code to be implemented in hardware during the runtime and hence improving on response time.

The example in Figure 2.4 gives how parts of a program code are being temporally loaded at different times $t_0 < t_1 < t_2$ into the FPGA accelerator.

Of course the achieved flexibility does not come for free. The whole concept relies heavily on a scheduler that decides which part of the code to be compiled and loaded

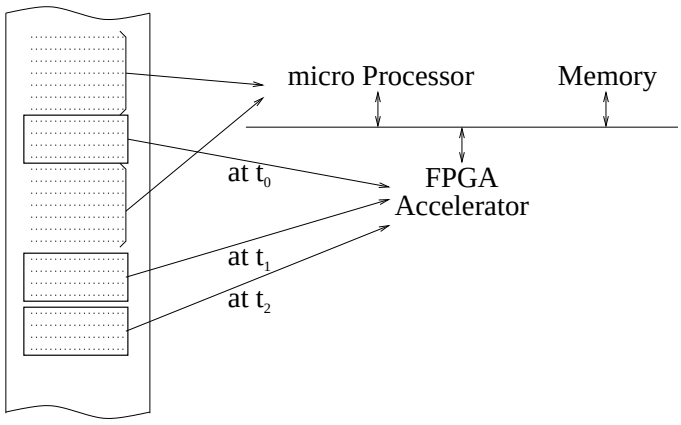


Figure 2.4: Temporal and Spatial Partitioning

into the FPGA “on the fly”. The particularly interesting area of research in reconfigurable processors [52] implements this idea. They are considered attractive due to their ability to adapt to a particular program and, thus improve its timing properties. The challenge in this context is to construct a compiler that utilises this feature in full.

2.5.2 System on Chip

System on chips [29, 82, 24, 39, 83] are in effect complete systems on a single piece of silicon. The normally separate pieces such as the CPU, memory controller, main memory, I/O control, and the various buses and interconnects, are placed on a chip. Main benefits are higher integration, reduced power consumption and size. Because of limited space on the chip though, the tradeoff of functionality implemented in hardware versus equivalent software package becomes ever so important. Therefore hardware/software codesign is the prime methodology for system on chip development.

Interesting aspect of todays system on chip design is that the traditional bus-based architectures are being replaced by “networks on chip” with associated protocols and network controllers.

2.6 Summary

As a roundup of my survey on co-design I can simply say that the design methods used in the early days are no longer applicable. The assumption that a specification is a “simple” and sequential program is no longer valid. Many applications can be heterogeneous and very complex.

The main goal of codesign is no longer acceleration only. It can be power consumption, size, available space on the chip, flexibility, adaptability, compositionality and maintainability. Architectures are no longer simple and/or bus-based either. They can be very complex networks and, especially with distributed applications, can have the topology of the Internet [24].

What is therefore needed is a unifying methodology within which

- the whole system is derived from a single logical representation,
- the system is compositionally refined,
- the gaps between various abstraction levels are bridged and
- simulation and verification are integrated.

In the following chapters I design and evaluate such a methodology.

Chapter 3

A Unifying Methodology for Codesign

My development strategy is presented together with the underlying formalisms of ITL and Tempura. Several different development methods can be derived from the methodology described here. One can mix different language platforms, hardware technologies and existing industrial tools in order to achieve flexibility and adaptability. I also present some of my fundamental results in incorporating memory variables into Tempura. The well published system for compositional verification and refinement is also given for completeness.

3.1 A Strategy for Codesign

The process of modelling a system, albeit sequential or concurrent, timed or untimed, needs a suitable computational model. I take the view that a computation defines math-

ematically an abstract architecture upon which applications will execute. A *system* is a collection of *agents* (which is my unit of computation), possibly executing concurrently and communicating (a)synchronously via communication links. Systems can themselves be viewed as single agents and composed into larger systems. Systems may have timing constraints imposed at three levels; system wide communication deadlines, agent deadlines and sub-computation deadlines (within the computation of an individual agent).

At any instant in time a system can be thought of as having a unique *state*. The system state is defined by the state variables of the system and, for concurrent system, by the values in the communication links. *Computation* is defined as any process that results in a change of system state. An agent is described by a computation which may transform a private data-space and may read and write to communication links during execution. The computation may have both minimum and maximum execution times imposed.

It is important to note that when I talk about *system* I do not make any distinction between software or hardware. I simply talk of a set of *agents* collaborating to achieve the desired behaviour. Some of those agents may be realised (or implemented) in software and some in hardware.

Fundamental to my proposed investigation is that a synthesis and design methodology should start with a high-level abstract specification which describes the desired behaviour(s) and interface(s) of the system under consideration. The target system is derived via design decisions made through *correctness preserving* refinement steps. My proposed development methodology is depicted in Figure 3.1 below.

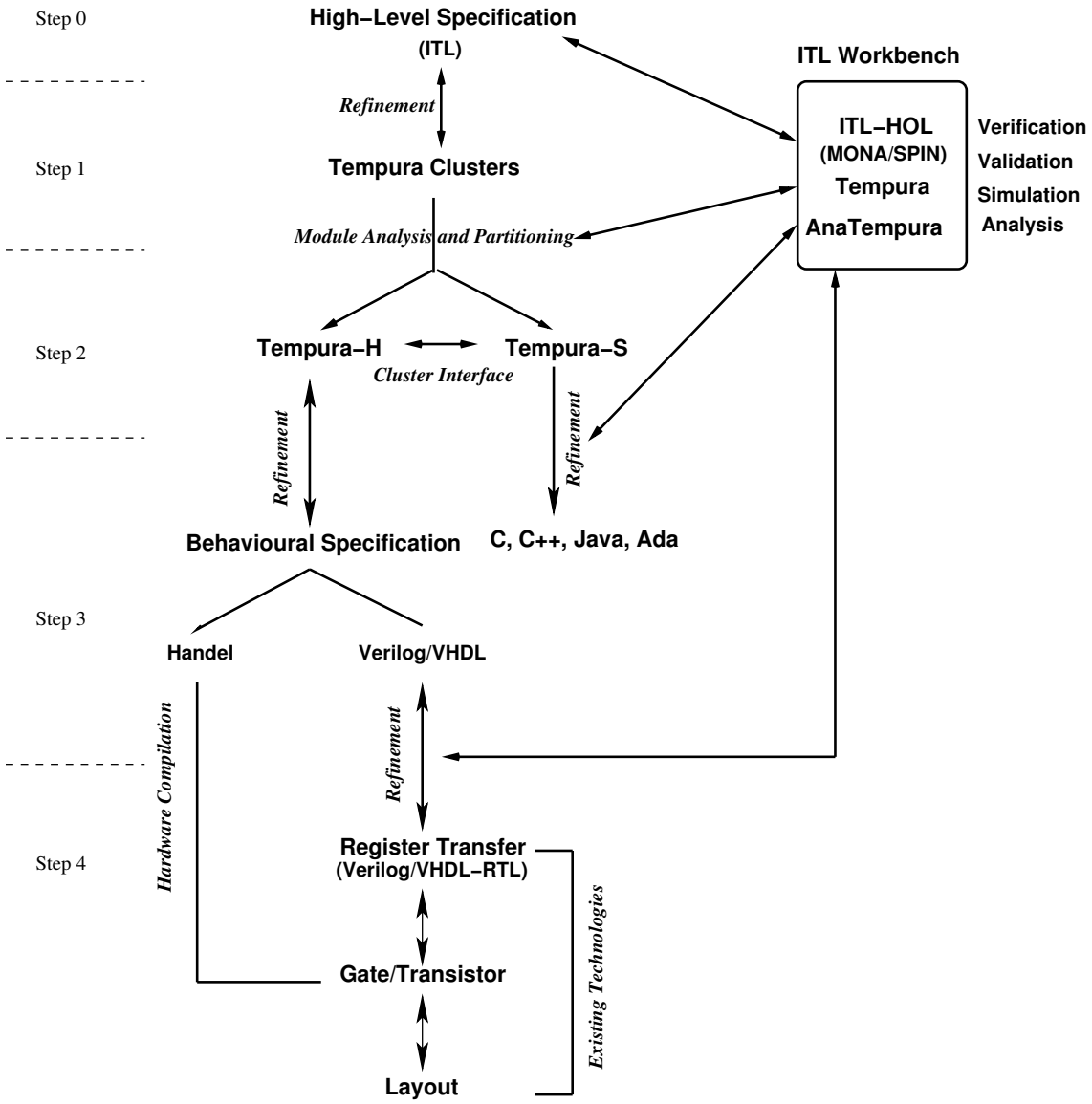


Figure 3.1: The Development Methodology

The design process begins with a high-level abstract specification written in Interval Temporal Logic (ITL) depicted in Figure 3.1 as Step 0. Properties of interest can be *compositionally* verified using the ITL's compositional proof rules in *assumption/commitment* style (see section 3.2). At this level I make no distinction between software or hardware.

Using a sound refinement calculus, the ITL specification can then be refined into a set of Tempura modules (an executable subset of ITL) and simulated and analysed (using Tempura, a part of the ITL Workbench). During this process, various design decisions are made, for example synchronous vs. asynchronous or sequential vs. parallel (Step 1).

This is followed by a 'module analysis' phase in which a set of quantitative and statistical data may be obtained (in [66] various techniques are given which can be utilised). The result of this phase is a partitioning into two clusters of modules, namely *Tempura-H* for the hardware part and *Tempura-S* for the software part. These are best realised in hardware and software implementation, respectively. The interface(s) between these clusters will depend on the target architecture and in turn can be verified compositionally using the ITL proof rules (Step 2).

The hardware and software parts are then refined into behavioural specification and a program in any HDL or programming language respectively as per Step 3. I mention Handel [41, 76] and Verilog as possible HDLs and C/C++, Java and Ada as software languages. Based on the particular choice of language and technology, my methodology can be reduced to a particular design method.

At last, Step 4 of my methodology is dedicated to the compilation phase of the devel-

opment. Once the RTL description of the hardware is achieved, I can use commercially available synthesis tools and produce a netlist which after that is being implemented into real piece of silicon. The corresponding step in the software branch has not been depicted for simplicity and to emphasise my focus on the hardware development.

The methodology given above can be specialised into several different design methods. All of them share the steps from Step 0 to Step 2. At Step 3 I can choose several programming and hardware description languages. Each particular method would specialise into the preferred design techniques that will depend on underlying hardware technology, language support and available expertise. For example, the choice between Handel and Verilog may depend on available synthesiser tools, while the choice between C and C++ could be based on the need for object oriented support in the design.

As depicted in Figure 3.1, the abstraction gaps existing between *Behavioural*, *RTL* and *Gate levels* must be bridged using sound refinement/transformation rules. It is fairly easy to define the refinement relation, and therefore derive practical refinement laws as per section 3.5, for the transition between ITL and Tempura since they both have common semantics. For example, if $Spec$ is a specification in ITL and P_T is a Tempura program, then

$$Spec \sqsubseteq P_T, \text{ iff } P_T \supset Spec.$$

Similarly, for the transition between Tempura and Verilog, as well as between the different abstraction levels within Verilog itself, a unifying semantics based on ITL for the

various notations, i.e. *Behavioural*, *RTL* and *Gate* notations, is needed. Such a unifying semantics is detailed in chapters 4 and 5 for my chosen HDL Verilog.

Several applications are *compute* bound as opposed to *control* bound, i.e. the computational complexity of the algorithms used in these applications grows in orders of magnitude with respect to the input size. For instance the 2-D Discrete Cosine Transformation (DCT) algorithm, widely used in many signal and image processing applications, is of $O(N^4)$ complexity for input sizes of $O(N^2)$ (for some size parameter N). *Regularity* in computation is a characterising feature prevalent in many of these algorithms. A model of computation that is well suited for efficient implementation of these algorithms is the *systolic* or *pipelined-parallelism* (see section 7) computational model. The nature (regular) and type (fine-grain) of parallelism in this model makes it particularly suited for implementing algorithms as hardware components [58]. A transformation algebra exists for systematically synthesising these components from high level specification [50].

Using sound refinement/transformation rules, the modules in the *Tempura-S* cluster could be transformed into software components written in popular languages, such as Java, C or C++. Similarly, modules in the *Tempura-H* cluster are further refined into a hardware description language such as Verilog. Alternatively, the *Tempura-H* cluster may be refined to Handel modules which subsequently compiled to netlists (through hardware compilation technology). As depicted in Figure 3.1, refinement calculi may be used to bridge the gap between the various abstraction levels in these technologies.

3.2 Underlying Formalisms

As underlying logic I choose ITL [65, 63, 64]. My choice is based mainly upon the following reasons; it is *simple, flexible* and has an *executable subset* giving the basis for both formal proof of the validity of the system design as well as simulation, animation and rapid prototyping in Tempura [64, 93]. Furthermore, ITL has a complete proof system for both its finite and infinite parts of the logic [62, 60].

A very major advantage of ITL is the ability to reason *compositionally* about specifications via assumption-commitment pairs [65]. This allows me to specify and prove compositional properties of the system in a practical way.

My formalism has to be dual in the sense that I do not only need to specify a system but I also need a framework into which I can reason about behaviours, including undesirable behaviours and avoiding them, as well as a framework into which I can execute, animate and simulate my specification. Therefore, I will use ITL when reasoning about and proving properties of my design while my main specification executable language will be Tempura which, being a subset of ITL, has formal semantics and can be both viewed as a programming language and logic.

However, the differences between ITL and Tempura, inherited by the executability of the later, command certain excess in our exposé. I will present here the syntax and the semantics of both despite Tempura's appurtenance into ITL and I shall justify this approach by the differences in the basic operators of the two languages.

3.3 Interval Temporal Logic

As I mentioned earlier, my proposed approach is based on a single logical framework whose underlying logic is Interval Temporal Logic. In this section I give an introduction to ITL, its syntax and formal semantics. For further reading and many practical examples I will refer the interested reader to [65, 63, 64, 93, 62, 60].

Interval Temporal Logic (ITL) is a flexible notation for both propositional and first order reasoning about intervals (behaviours) found in descriptions of hardware and software systems. It can handle both sequential and parallel composition unlike most temporal logics. It offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and timeliness.

3.3.1 Syntax of ITL

The syntax of ITL is defined in table 3.1 where μ is an integer value, a is a static variable (does not change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol.

Expressions	$e ::=$	$\mu \mid a \mid A \mid g(e_1, \ldots, e_n)$
Formulae	$f ::=$	$p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \exists v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2$

Table 3.1: Syntax of ITL

With these operators I can define the usual temporal operators $\Box$, $\Diamond$ and $\bigcirc$, and the Tempura constructs empty, if then else, etc.

- the predicates: *true* $\widehat{=} 0 = 0$ and *false* $\widehat{=} \neg$*true*.

- disjunction, implication and equivalence: $f_1 \vee f_2 \mathrel{\widehat{=}} \neg(\neg f_1 \wedge \neg f_2)$,

  $f_1 \supset f_2 \mathrel{\widehat{=}} \neg f_1 \vee f_2$ and $f_1 \equiv f_2 \mathrel{\widehat{=}} (f_1 \supset f_2) \wedge (f_2 \supset f_1)$.

- If-Then-Else: $\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2 \mathrel{\widehat{=}} (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$.

- universal quantification: $\forall v \bullet f \mathrel{\widehat{=}} \neg\exists v \bullet \neg f$.

- next, more and empty: $\bigcirc f \mathrel{\widehat{=}} \mathsf{skip} \,;\, f$,

  more $\widehat{=} \bigcirc$*true* and $\mathsf{empty} \mathrel{\widehat{=}} \neg$*more*.

- chop-star: $f^* \mathrel{\widehat{=}} \mathsf{empty} \vee (f \wedge \mathit{more}) \,;\, f^*$.

- infinite and finite: *inf* $\widehat{=}$ *true* ; *false* and *finite* $\widehat{=} \neg$*inf*.

- sometimes and always: $\Diamond f \mathrel{\widehat{=}}$ *finite* ; f and $\Box f \mathrel{\widehat{=}} \neg\Diamond\neg f$.

- some subinterval, some initial subinterval, all subintervals, mostly and keep:

  $\diamondsuit f \mathrel{\widehat{=}} \Diamond(f \,;$ *true*$)$,

  $\diamond\!\!\!\!\cdot\, f \mathrel{\widehat{=}} f \,;$ *true*,

  $\boxed{\mathrm{a}}\, f \mathrel{\widehat{=}} \neg(\diamondsuit \neg f)$,

  $\boxed{\mathrm{m}}\, f \mathrel{\widehat{=}} \Box(\mathit{more} \supset f)$ and

  keep $f \mathrel{\widehat{=}} \boxed{\mathrm{a}}(\mathsf{skip} \supset f)$.

- final state: *fin* $f \mathrel{\widehat{=}} \Box(\mathsf{empty} \supset f)$.

For example, in an interval, if the variable I always equals 1 and in the next state the variable J equals 2 then it follows that the expression $I + J$ equals 3 in the next state:

$$\Box(I = 1) \wedge \bigcirc(J = 2) \quad \supset \quad \bigcirc(I + J = 3)$$

Many more examples can be found in [64] as well as in later chapters were I show how I use ITL and Tempura to specify and reason about Verilog programs.

Types in ITL

There are two basic inbuilt types in ITL (which can be given pure set-theoretic definitions). These are integers $\mathcal{Z}$ (together with standard relations of inequality and equality) and Boolean (*true* and *false*). In addition, the executable subset of ITL (Tempura) has the basic type vector (see table 3.2).

Further types, including reals and matrixes, can be built from these by means of $\times$ and the power set operator, $\mathcal{P}$ similarly to the method adopted in the specification language Z [40].

3.3.2 Semantics of ITL

Every ITL formula is evaluated over a specific time interval which is simply an (in)finite sequence of states. Each state represents a mapping between the set of variables and their values, i.e. the state is a snapshot of the values of the set of the variables. So let me assume

an interval σ with consecutive states $\sigma_0, \sigma_1, \ldots$.

Informal Semantics

There are two main categories in ITL, namely *expressions* and *formulas*. Each constant and static variable of any type keeps its value over σ, while state variables of any type may change.

All *formulas* are evaluated over the whole interval. For example, $f_1 \wedge f_2$ is true over σ, iff f_1 and f_2 are true over σ. Similarly $\exists$ represents the existential quantifier.

More interesting are skip, ";" and "*". While skip is true over every interval with two states, there are three cases when $f_1 \,;\, f_2$ could be true over σ.

1. σ is a finite interval $\sigma_0, \ldots, \sigma_n$ and it can be split into two subintervals $\sigma' = \sigma_0, \ldots, \sigma_k$ and $\sigma'' = \sigma_k, \ldots, \sigma_n$ sharing the common state σ_k for some $0 \leq k \leq n$ and f_1 is true over σ' and f_2 is true over σ''.

2. σ is an infinite interval $\sigma_0, \ldots$ and it can be split into two subintervals $\sigma' = \sigma_0, \ldots, \sigma_k$ and $\sigma'' = \sigma_k, \ldots$ sharing the common state σ_k for some $0 \leq k$ and f_1 is true over σ' and f_2 is true over σ''.

3. σ is an infinite interval $\sigma_0, \ldots$ and f_1 is true over σ.

For "*" I will say that it is the repetitive enclosure of ";" and therefore there are again three cases for the truth value of f^* over σ.

1. σ is a finite interval $\sigma_0, \ldots, \sigma_n$ and it can be split into finite number of subintervals $\sigma^{(j)} = \sigma_{k_j}, \ldots, \sigma_{k_{j+1}}$, where the first index $k_0 = 0$, the last index is n and $0 \leq k_j \leq k_{j+1} \leq n$, sharing the common states σ_{k_j} for all k_j and f is true over all $\sigma^{(j)}$.

2. σ is an infinite interval $\sigma_0, \ldots$ and it can be split into infinite number of finite subintervals $\sigma^{(j)} = \sigma_{k_j}, \ldots, \sigma_{k_{j+1}}$ and $0 = k_0 \leq k_j \leq k_{j+1}$ sharing the common states σ_{k_j} for all $0 \leq k_j$ and f is true over all $\sigma^{(j)}$.

3. σ is an infinite interval $\sigma_0, \ldots$ and it can be split into two subintervals $\sigma' = \sigma_0, \ldots, \sigma_k$ and $\sigma'' = \sigma_k, \ldots$ sharing the common state σ_k for some $0 \leq k$ and f^* is true over σ' and f is true over σ''.

Formal Semantics

In the text below, I will assume that tt and ff are the truth and the false values, $\hat{g}$ is the function that corresponds to the functional symbol g and $\hat{p}$ is the predicate that corresponds to the predicate symbols p. I also write $\sigma \sim_v \sigma'$ if the intervals σ and σ' are identical with the possible exception of their mappings for the variable v and I denote the length of σ with $|\sigma|$.

- $\mathcal{M}_\sigma[\![v]\!] = \sigma_0(v)$.

- $\mathcal{M}_\sigma[\![g(e_1, \ldots, e_n)]\!] = \hat{g}(\mathcal{M}_\sigma[\![e_1]\!], \ldots, \mathcal{M}_\sigma[\![e_n]\!])$.

- $\mathcal{M}_\sigma[\![p(e_1, \ldots, e_n)]\!] =$ tt, iff $\hat{p}(\mathcal{M}_\sigma[\![e_1]\!], \ldots, \mathcal{M}_\sigma[\![e_n]\!])$.

- $\mathcal{M}_\sigma[\![\neg f]\!] =$ tt iff $\mathcal{M}_\sigma[\![f]\!] =$ ff.
- $\mathcal{M}_\sigma[\![f_1 \wedge f_2]\!] =$ tt iff $\mathcal{M}_\sigma[\![f_1]\!] =$ tt and $\mathcal{M}_\sigma[\![f_2]\!] =$ tt.
- $\mathcal{M}_\sigma[\![\exists v \bullet f]\!] =$ tt iff for some σ' s.t. $\sigma \sim_v \sigma'$, $\mathcal{M}_{\sigma'}[\![f]\!] =$ tt.
- $\mathcal{M}_\sigma[\![\mathsf{skip}]\!] =$ tt iff $|\sigma| = 1$, i.e. σ has length 1.
- $\mathcal{M}_\sigma[\![f_1 \ ; \ f_2]\!] =$ tt iff (exists a k, s.t. $\mathcal{M}_{\sigma_0 \ldots \sigma_k}[\![f_1]\!] =$ tt and ((σ is infinite and $\mathcal{M}_{\sigma_k \ldots}[\![f_2]\!] =$ tt) or (σ is finite and $k \leq |\sigma|$ and $\mathcal{M}_{\sigma_k \ldots \sigma_{|\sigma|}}[\![f_2]\!] =$ tt))) or (σ is infinite and $\mathcal{M}_\sigma[\![f_1]\!]$).

I would like to note here that the essential operator “chop-star” is expressible, i.e. I can infer its formal semantics from the definitions above.

3.4 Tempura

As it will become clear later in chapter 4, Tempura, being a strict subset of ITL, is expressive enough for the purposes of Verilog’s semantics. I will give here the syntax and the formal semantics of this executable subset.

3.4.1 Syntax of Tempura

The syntax of Tempura is defined in table 3.2 where μ is an integer value, a, a_1, etc. are integer static variables (don’t change within an interval), A, A_1, etc. are integer state

variables (can change within an interval), $|\mathbf{v}|$ is length of a vector, b, b_1, etc. are boolean static variables, B, B_1, etc. are boolean state variables, p is a predicate symbol, l, l_1, etc. are vector static variables, L, L_1, etc. are vector state variables, $[e_1, \ldots, e_n]$ is a vector of n expressions, $\mathbf{v}[s]$ is an element of a vector, v can be any static or state (including scalar, boolean or vector) variable, g_s is a function symbol with scalar range, $g_{\mathbf{v}}$ is a function symbol with vector range and n is a natural value. I will use *integer* and *list* as synonyms

Scalar	$s ::=$	$\mu \mid a \mid A \mid g_s(e_1, \ldots, e_n) \mid \lvert\mathbf{v}\rvert$
Boolean	$\mathbf{b} ::=$	$b \mid B \mid \mathsf{empty} \mid p(e_1, \ldots, e_n) \mid \neg\mathbf{b} \mid \mathbf{b_1} \wedge \mathbf{b_2}$
Vector	$\mathbf{v} ::=$	$l \mid L \mid [e_1, \ldots, e_n] \mid g_{\mathbf{v}}(e_1, \ldots, e_n)$
Expression	$e ::=$	$s \mid \mathbf{b} \mid \mathbf{v} \mid \mathbf{v}[s]$
Formulae	$f ::=$	$\mathbf{b} \mid f_1 \wedge f_2 \mid \exists v \bullet f \mid \mathsf{skip} \mid f_1 \,;\, f_2 \mid \mathsf{if}\ \mathbf{b}\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2$

Table 3.2: Syntax of Tempura

for *scalar* and *vector* henceforth. Also, when I talk about variable I will understand both integer and list variable.

Any boolean expression is a *state* formula. In Tempura, *negation* "$\neg$" is defined over state formulas only. The operator "empty" is a special state formula.

The current Tempura tool [93] includes some other constructs for *input*, *output* and constructs for *random* numbers which we will omit here for simplicity. I will denote the language of Tempura defined by table 3.2 with $\mathcal{L}_T$. Many interesting operators can be expressed with this minimalistic basic set. Some of them are:

- operator next over formulae: $\bigcirc f \mathrel{\widehat{=}} \mathsf{skip} \,;\, f$.

- operator next over expressions: $X = \bigcirc e$, iff $\exists x \bullet X = x \wedge \bigcirc(x = e)$ where X and

x are state and static variables respectively.

- operator if-then: $\mathsf{if}\ \mathbf{b}\ \mathsf{then}\ f \mathrel{\widehat{=}} \mathsf{if}\ \mathbf{b}\ \mathsf{then}\ f\ \mathsf{else}\ \mathit{true}$.
- operator more: $\mathit{more} \mathrel{\widehat{=}} \neg\mathsf{empty}$.
- operator weak next over formulae: $\circledcirc f \mathrel{\widehat{=}} \mathsf{if}\ \mathit{more}\ \mathsf{then} \bigcirc f$.
- operator weak next over expressions: $X = \circledcirc e$, iff $\exists x \bullet X = x \wedge \circledcirc(x = e)$ where X and x are state and static variables correspondingly.
- unit assignment: $X := e \mathrel{\widehat{=}} (\bigcirc X) = e$.
- operator always: $\Box f \mathrel{\widehat{=}} f \wedge \circledcirc \Box f$, $\boxed{\mathsf{a}}\, f \mathrel{\widehat{=}} \Box(f\ ;\ \mathit{true})$.
- operator keep: $\mathit{keep}\ f \mathrel{\widehat{=}} \boxed{\mathsf{a}}(\circledcirc f)$.
- operator gets: $X\ \mathit{gets}\ Y \mathrel{\widehat{=}} \mathit{keep}\ (X := Y)$.
- operator eventually in Tempura: $\diamondsuit \mathbf{b} \mathrel{\widehat{=}} \mathsf{if}\ \mathbf{b}\ \mathsf{then}\ \mathit{true}\ \mathsf{else} \bigcirc \diamondsuit \mathbf{b}$. I would like to emphasise here that the definitions of $\diamond$ in ITL as given in section 3.3 and $\diamondsuit$ in Tempura are very different. In general, $\diamond$ captures finite intervals only, while $\diamondsuit$ includes the infinite behaviour as well.
- operator infinite: $\mathit{inf} \mathrel{\widehat{=}} \diamondsuit \mathit{false}$.
- operator while: $\mathsf{while}\ \mathbf{b}\ \mathsf{do}\ f \mathrel{\widehat{=}} \mathsf{if}\ \mathbf{b}\ \mathsf{then}\ [f\ ;\ (\mathsf{while}\ \mathbf{b}\ \mathsf{do}\ f)]\ \mathsf{else}\ \mathsf{empty}$.
- operator iteration, also called “chopstar”: $f^* \mathrel{\widehat{=}} \mathsf{if}\ \mathit{more}\ \mathsf{then}\ (f\ ;\ f^*)\ \mathsf{else}\ \mathsf{empty}$.

- bounded universal quantifier: $\forall v < s \bullet f(v) \mathrel{\widehat{=}} f(0) \wedge \ldots \wedge f(s-1)$.

Similarly, I can give several useful constructs on lists.

- concatenation: $L = L_1 + L_2$, iff $|L| = |L_1| + |L_2| \wedge \forall i < |L_1| + |L_2| \bullet$ if $i < |L_1|$ then $L[i] = L_1[i]$ else $L[i] = L_2[i - |L_1|]$.

- sublist: $R = L[s_1..s_2]$, iff if $(0 \leq s_1 \wedge s_1 \leq s_2 \wedge s_2 \leq |L|)$ then $\{|R| = s_2 - s_1 \wedge \forall i < (s_2 - s_1) \bullet R[i] = L[s_1 + i]\}$.

- head and tail: $L = [a|R]$, iff $|L| = 1 + |R| \wedge a = L[0] \wedge R = L[1..|L|]$.

It turns out that the primitives given in table 3.2 can generate a rich set of constructs turning $\mathcal{L}_T$ into a *general purpose programming language with sound semantics*. Examples can be found in [93, 64].

3.4.2 Semantics of Tempura

The formal semantics of $\mathcal{L}_T$ can be easily derived from the ITL semantics given in section 3.4 and [93, 64]. Here I will restrict ourselves to the parts of $\mathcal{L}_T$ that are not covered there, i.e. length of a list and list member selection.

- $\mathcal{M}_\sigma[\![|[e_1, \ldots, e_n]|]\!] = \mathcal{M}_\sigma[\![n]\!]$.

- $\mathcal{M}_\sigma[\![[e_1, \ldots, e_n][s]]\!] = \mathcal{M}_\sigma[\![e_s]\!]$.

3.4.3 Tempura with Memory

Having in mind that I will have to reason about memory, i.e. registers in hardware, I will need an appropriate formal concept that can grasp the properties of, and will be my abstraction for, *memory*. I recognise the fact that ITL and Tempura do not have *memory variables* in its basic syntax and semantics. Research [32] has pointed out the need for framing in ITL which would have solved this problem. As it has been shown before, framing is generally considered a difficult issue so I need something quicker and easier to suit my purpose. I will denote these memory variables as follows $\mathcal{A}$, $\mathcal{B}$, $\mathcal{V}$, etc.

For each memory variable of type T, I will have a normal state variable V of the same type T holding the value of $\mathcal{V}$ and a normal boolean state variable $new_{\mathcal{V}}$ which would tell me if $\mathcal{V}$ has changed its value in the transition between the last and the current state.

Typically I would want to use formulas like

$$\exists \mathcal{V} \bullet (f_1 \wedge f_2) \tag{3.1}$$

where $\mathcal{V}$ is a memory variable shared between the parallel formulas f_1 and f_2. Basic memory property is to keep its value unless it has been changed explicitly via an assignment operator ":=" . Normal Tempura variables do not have this feature and are free to change at any state where they are not given a value. For example, consider an interval of three states namely $\sigma = \sigma_0, \sigma_1, \sigma_2$ and a formula $\exists V \bullet (V = 0 \wedge \mathsf{skip}); \mathsf{skip}; (V = 0 \wedge \mathsf{empty})$ which evaluates to true over σ. Obviously, V's value is bounded to 0 in states σ_0 and σ_2 while in

contrast it is completely unspecified and free in σ_1. The obvious way to solve this problem is to use Tempura's construct stable $V \mathrel{\widehat{=}} V$ *gets* V as in $\exists V \bullet V = 0 \wedge$ stable V. However, this is correct only if V's value is not allowed to be changed by a parallel process. In the case of (3.1) though, I cannot use stable $\mathcal{V}$ within f_1 or f_2 because $\mathcal{V}$ is shared between the two concurrent formulas and each one of them may change $\mathcal{V}$'s value at any point.

Fortunately I am able to transform (3.1) into a classical ITL formula. My strategy is to enrich the Tempura syntax by adding a new type of expressions, i.e. any memory variable is now a valid expression and the existential quantifier $\exists$ can quantify over the new memory variables. The language of $\mathcal{L}_T$ with this new type of expressions and formulas will be denoted as $\mathcal{L}_T+$.

The next step is to use Hale's mechanism "inertia" and following [32] I define a translation from a formula f with a memory variable $\mathcal{V}$. For every such variable I will introduce two state variables V and $new_{\mathcal{V}}$. Now I can transform f into

$$\exists \Delta \bullet \phi(\mathcal{V}, V, \Delta, f) \wedge \Box(\Delta = \circledcirc new_{\mathcal{V}}) \wedge \Box(\textsf{if } \neg\Delta \textsf{ then } (V = \circledcirc V)). \tag{3.2}$$

In this context, $f[\mathcal{V}/V]$ means V substitutes $\mathcal{V}$ into f and $state$ stands for a state formula.

$$\phi(\mathcal{V}, V, \Delta, state) \mathrel{\widehat{=}} state[\mathcal{V}/V]$$

$$\phi(\mathcal{V}, V, \Delta, \mathcal{V} := e) \mathrel{\widehat{=}} \Delta = \neg(V = e[\mathcal{V}/V]) \wedge V := e[\mathcal{V}/V]$$

$$\phi(\mathcal{V}, V, \Delta, \mathcal{V}' := e) \mathrel{\widehat{=}} \Delta = \mathit{false} \wedge \mathcal{V}' := e[\mathcal{V}/V]$$

$$\phi(\mathcal{V}, V, \Delta, \mathsf{skip}) \mathrel{\widehat{=}} \Delta = \mathit{false} \wedge \mathsf{skip}$$

$$\phi(\mathcal{V}, V, \Delta, f_1 \,;\, f_2) \mathrel{\widehat{=}} \phi(\mathcal{V}, V, \Delta, f_1) \,;\, \phi(\mathcal{V}, V, \Delta, f_2)$$

$$\phi(\mathcal{V}, V, \Delta, \mathsf{if}\ \mathit{bool}\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2) \mathrel{\widehat{=}} \mathsf{if}\ \mathit{bool}[\mathcal{V}/V]\ \mathsf{then}\ \phi(\mathcal{V}, V, \Delta, f_1)$$
$$\mathsf{else}\ \phi(\mathcal{V}, V, \Delta, f_2)$$

$$\phi(\mathcal{V}, V, \Delta, \exists \mathcal{V} \bullet f) \mathrel{\widehat{=}} \Delta = \mathit{false} \wedge \exists \mathcal{V} \bullet f$$

$$\phi(\mathcal{V}, V, \Delta, \exists V' \bullet f) \mathrel{\widehat{=}} \exists V' \bullet \phi(\mathcal{V}, V, \Delta, f)$$

$$\phi(\mathcal{V}, V, \Delta, f_1 \wedge f_2) \mathrel{\widehat{=}} \exists \Delta_1, \Delta_2 \bullet \Box(\Delta = \Delta_1 \vee \Delta_2) \wedge \phi(\mathcal{V}, V, \Delta_1, f_1) \wedge$$
$$\phi(\mathcal{V}, V, \Delta_2, f_2)$$

After the definition of the translation, I will show that the so constructed memory variables do have the basic "memory" property I needed, i.e. I can accept them as an abstraction for memory. In the following theorem I will assume $\neq$ as a primitive predicate with the appropriate semantics.

Theorem 1 *If $\mathcal{V}$ is a memory variable, then*

$$(\mathsf{skip} \wedge \mathit{true} = \bigcirc new_{\mathcal{V}}) \quad \equiv \quad (\mathcal{V} \neq \bigcirc \mathcal{V} \equiv V \neq \bigcirc V)$$

Proof (Theorem 1) I will start the proof by transforming $\mathsf{skip} \wedge \mathcal{V} \neq \bigcirc \mathcal{V}$ following the

definition above.

$$\exists x, \Delta, \Delta_1, \Delta_2 \bullet \mathsf{skip} \wedge \Box(\Delta = \Delta_1 \vee \Delta_2) \wedge x \neq V \wedge [(\Delta_2 = \mathit{false} \wedge \mathsf{skip}) ; x = V] \wedge$$

$$\Box(\Delta = ⍤new_{\mathcal{V}}) \wedge \Box(\mathsf{if}\ \neg\Delta\ \mathsf{then}\ V = ⍤V)$$

Expanding the $[\ldots]$ brackets leads to

$$\exists x, \Delta, \Delta_1, \Delta_2 \bullet \mathsf{skip} \wedge \Box(\Delta = \Delta_1 \vee \Delta_2) \wedge \Delta_2 = \mathit{false} \wedge x \neq V \wedge (\mathsf{skip} ; x = V) \wedge$$

$$\Delta = \bigcirc new_{\mathcal{V}} \wedge \mathsf{if}\ \neg\Delta\ \mathsf{then}\ V = ⍤V$$

Now I have to remember that $V \neq \bigcirc V \equiv \exists x \bullet x \neq V \wedge (\mathsf{skip} ; x = V)$, therefore I can transform further

$$\mathsf{skip} \wedge V \neq \bigcirc V \wedge \exists \Delta, \Delta_1, \Delta_2 \bullet \Box(\Delta = \Delta_1 \vee \Delta_2) \wedge \Delta_2 = \mathit{false} \wedge$$

$$\Delta = \bigcirc new_{\mathcal{V}} \wedge \mathsf{if}\ \neg\Delta\ \mathsf{then}\ V = ⍤V$$

and if I mark

$$F = \exists \Delta, \Delta_1, \Delta_2 \bullet \Box(\Delta = \Delta_1 \vee \Delta_2) \wedge \Delta_2 = \mathit{false} \wedge \Delta = \bigcirc new_{\mathcal{V}} \wedge \mathsf{if}\ \neg\Delta\ \mathsf{then}\ V = ⍤V,$$

then I can write finally

$$\mathsf{skip} \wedge \mathcal{V} \neq \bigcirc \mathcal{V} \quad \equiv \quad \mathsf{skip} \wedge V \neq \bigcirc V \wedge F$$

Now I will look at F

$$\textit{true} = \bigcirc new_{\mathcal{V}} \quad \supset \quad F \equiv \exists \Delta, \Delta_1, \Delta_2 \bullet \Box(\Delta = \Delta_1 \vee \Delta_2) \wedge \Delta_2 = \textit{false} \wedge \Delta = \textit{true}$$

$$\equiv \textit{true}$$

Alternatively,

$$\textit{false} = \bigcirc new_{\mathcal{V}} \quad \supset \quad F \equiv V = \textcircled{w} V$$

and therefore

$$\textit{true} = \bigcirc new_{\mathcal{V}} \quad \supset \quad \mathsf{skip} \wedge \mathcal{V} \neq \bigcirc \mathcal{V} \equiv \mathsf{skip} \wedge V \neq \bigcirc V$$

while

$$\textit{false} = \bigcirc new_{\mathcal{V}} \quad \supset \quad \mathsf{skip} \wedge \mathcal{V} \neq \bigcirc \mathcal{V} \equiv \mathsf{skip} \wedge V \neq \bigcirc V \wedge V = \textcircled{w} V \equiv \textit{false}$$

From this I can conclude the theorem. ■

The theorem shows that the $new_{\mathcal{V}}$ always picks up the states where $\mathcal{V}$ has changed value, i.e. an event. Also, the equivalence stated by the theorem guarantees that if the value of $\mathcal{V}$ is not explicitly changed, then it will stay the same, i.e. it does memorise the value.

Before I go further I would like to make the important remark that the semantics of $\mathcal{L}_T+$ is expressed in $\mathcal{L}_T$, i.e. $\mathcal{L}_T+$ is a syntactical extension only which preserves the complete axiom system given in [62, 60].

3.5 Compositional Verification

In order to support system development in an optimal way, description techniques for models of specific system views must be intuitively understandable and be precise enough to ensure an unambiguous and consistent description of the system. In addition, such a technique must be *compositional* allowing the modular description and verification of the system.

Compositional verification is provided through an *assumption-commitment* style framework. The following implication illustrates the use of such style with a system Sys:

$$w \wedge As \wedge Sys \supset Co \wedge \mathit{fin}\ w'.$$

This states that if the state formula w is true in the initial state and the assumption As is true over the interval in which Sys is operating, then the commitment Co is also achieved. Furthermore the state formula w' is true in the interval's final state or is vacuously true if the interval does not terminate.

In general, the assumption As and the commitment Co can be arbitrary ITL formulas.

However, when reasoning about a system built out of sequential parts, it is advantageous to consider certain kinds of assumptions and commitments which readily lend themselves to suitable proof rules.

More specifically, I require that As and Co be respective fixpoints of the ITL operators $\boxdot$ and "$*$" as is now shown:

$$As \equiv \boxdot As \ , \qquad Co \equiv Co^* \ .$$

The first equivalence ensures that if the assumption As is true on an interval, it is also true in all subintervals. The second ensures that if zero or more sequential instances of the commitment Co span an interval, Co is also true on the interval itself.

The temporal formula $\Box(K = 1)$ is an example of a suitable assumption while some formulas such as stable K can be used both as assumptions and commitments as these are precisely the fixpoints of the ITL operator *keep*. For assumptions and commitments obeying the above, the following derivable proof rule is sound:

$$\frac{\begin{array}{l} w \wedge As \wedge Sys \ \supset \ Co \wedge \mathit{fin}\ w' \\ w' \wedge As \wedge Sys' \ \supset \ Co \wedge \mathit{fin}\ w'' \end{array}}{w \wedge As \wedge (Sys; Sys') \ \supset \ Co \wedge \mathit{fin}\ w'' \ .} \tag{3.3}$$

Here is an analogous rule for decomposing a proof for zero or more iterations of a formula

Sys:

$$\frac{w \wedge As \wedge Sys \supset Co \wedge \mathit{fin}\ w}{w \wedge As \wedge Sys^* \supset Co \wedge \mathit{fin}\ w\ .} \tag{3.4}$$

Compositional reasoning about *liveness* is also possible.

3.6 Formal Refinement and Analysis

3.6.1 Refinement Calculus

The transformation process in each step (see Figure 3.1) is based on a refinement calculus that allows me to systematically calculate the desired system description. The refinement relation $\sqsubseteq$ is defined on a system: A system $\mathcal{X}$ is *refined* by the system $\mathcal{Y}$, denoted $\mathcal{X} \sqsubseteq \mathcal{Y}$, if and only if $\mathcal{Y} \supset \mathcal{X}$. A set of sound refinement laws have been derived [13] to transform an abstract system specification into concrete systems.

Two observations are in order:

1. Once I have completed the formal specification phase, various properties could be proven about the specification itself. This can provide an extra assurance that the final specification meets the required informal requirements.
2. At each refinement step, I can simulate the resulting (sub-)system. This gives some guidelines on the choice of the subsequent refinement rules.

The following basic law states that the operators in ITL are monotonic with respect to the refinement relation. Monotonicity means that the ITL refinement calculus is compositional.

Law 1 (Monotonicity)

Let f_i be an ITL formula then

$$(\sqsubseteq -1)\ \textit{If}\ f_0 \sqsubseteq f_1\ \textit{and}\ f_1 \sqsubseteq f_2\ \textit{then}\ f_0 \sqsubseteq f_2$$

$$(\sqsubseteq -2)\ \textit{If}\ f_0 \sqsubseteq f_1\ \textit{and}\ f_2 \sqsubseteq f_3\ \textit{then}\ (f_0 \wedge f_2) \sqsubseteq (f_1 \wedge f_3)$$

$$(\sqsubseteq -3)\ \textit{If}\ f_0 \sqsubseteq f_1\ \textit{and}\ f_2 \sqsubseteq f_3\ \textit{then}\ (f_0 \vee f_2) \sqsubseteq (f_1 \vee f_3)$$

$$(\sqsubseteq -4)\ \textit{If}\ f_1 \sqsubseteq f_2\ \textit{then}\ f_0 ; f_1 \sqsubseteq f_0 ; f_2$$

$$(\sqsubseteq -5)\ \textit{If}\ f_1 \sqsubseteq f_2\ \textit{then}\ f_1 ; f_0 \sqsubseteq f_2 ; f_0$$

$$(\sqsubseteq -6)\ \textit{If}\ f_0 \sqsubseteq f_1\ \textit{then}\ f_0^* \sqsubseteq f_1^*$$

$$(\sqsubseteq -7)\ \textit{If}\ f_0 \sqsubseteq f_1\ \textit{then}\ \forall v \bullet f_0 \sqsubseteq \forall v \bullet f_1$$

Following are some useful refinement rules for refining ITL specifications into Tempura code. The conditional is introduced with the following rule.

Rule 1 (If then else)

$$(\textsf{if}\ -1)\quad (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2) \qquad \sqsubseteq \qquad \textsf{if}\ f_0\ \textsf{then}\ f_1\ \textsf{else}\ f_2$$

The following rules describes the characteristics of the chop construct (";"). Chop has empty as a unit, is associative and distributes over nondeterministic choice and condi-

tional

Rule 2 (Chop)

$$(;-1)\quad \textsf{empty}\,;f \quad\equiv\quad f \quad\equiv\quad f\,;\textsf{empty}$$

$$(;-2)\quad (f_1\,;f_2)\,;f_3 \quad\equiv\quad f_1\,;(f_2\,;f_3)$$

$$(;-3)\quad f_1\,;(f_2\vee f_3)\,;f_4 \quad\equiv\quad (f_1\,;f_2\,;f_4)\vee(f_1\,;f_3\,;f_4)$$

$$(;-4)\quad (\textsf{if } f_0 \textsf{ then } f_1 \textsf{ else } f_2)\,;f_3 \quad\equiv\quad \textsf{if } f_0 \textsf{ then } (f_1\,;f_3) \textsf{ else } (f_2\,;f_3)$$

The following rules introduce the while loop and the non-terminating loop

Rule 3 (While)

$$(\textsf{while}-1)\quad (f_0\wedge f_1)^*\wedge \mathit{fin}\,\neg f_0 \quad\sqsubseteq\quad \textsf{while } f_0 \textsf{ do } f_1$$

$$(\textsf{while}-2)\quad f_1^* \quad\sqsubseteq\quad f_1^*\wedge \mathit{inf} \quad\equiv\quad \textsf{while } \mathit{true} \textsf{ do } f_1$$

The following are some rules for the parallel construct $\wedge$.

Rule 4 (Parallel)

$$(\wedge-1)\quad f_0\wedge f_1 \quad\equiv\quad f_1\wedge f_0$$

$$(\wedge-2)\quad f_0\wedge(f_1\vee f_2) \quad\equiv\quad (f_0\wedge f_1)\vee(f_0\wedge f_2)$$

$$(\wedge-3)\quad (f_0\wedge f_1)\wedge f_2 \quad\equiv\quad f_0\wedge(f_1\wedge f_2)$$

$$(\wedge-4)\quad (\textsf{if } f_0 \textsf{ then } f_1 \textsf{ else } f_2)\wedge f_3 \quad\equiv\quad \textsf{if } f_0 \textsf{ then } (f_1\wedge f_3) \textsf{ else } (f_2\wedge f_3)$$

3.6.2 Analysis

Fundamentally, I can use the approach given in [88] to capture a possible behaviour of a running (sub-)system. Once the behaviour is captured then I can assert if such behaviour satisfies a given property, i.e. runtime validation and testing. And as a property is a set of behaviours, *satisfaction* is achieved by checking if the captured system's behaviour is an element of this set. I am not dealing here with the formal verification of properties which requires that all possible behaviours of system satisfy the properties. I am only concerned with validating properties which requires that only interesting behaviours satisfy the properties.

The states of a (sub-)system to be analysed are captured by inserting *assertion points* at suitably chosen places. These divide the system into several *code-chunks*. Properties of interests are then validated for this behaviour.

The general framework for analysis can be described as follows.

1. Establish all desirable properties of the system under consideration and express them in Tempura.
2. Identify suitable places in the code and insert assertional points.
3. Using Tempura, check that the behaviour satisfies the desired properties.

Establishing system properties can be a hard task, however I suggest to follow the main characterisation of properties given above, namely safety, liveness and timing properties. Obviously, some level of understanding of the (sub-)system under consideration is as-

sumed. These properties could be invariants that need to be true at all levels of system's abstraction.

The locations of assertion points could be chosen, for example, at the entry and exit points of a procedure or function. In this case assertions are in fact *pre-* and *post-* conditions, and what I am asserting is: If the system starts at a state satisfying the *pre-* condition then it terminates properly in a state satisfying the *post-* condition.

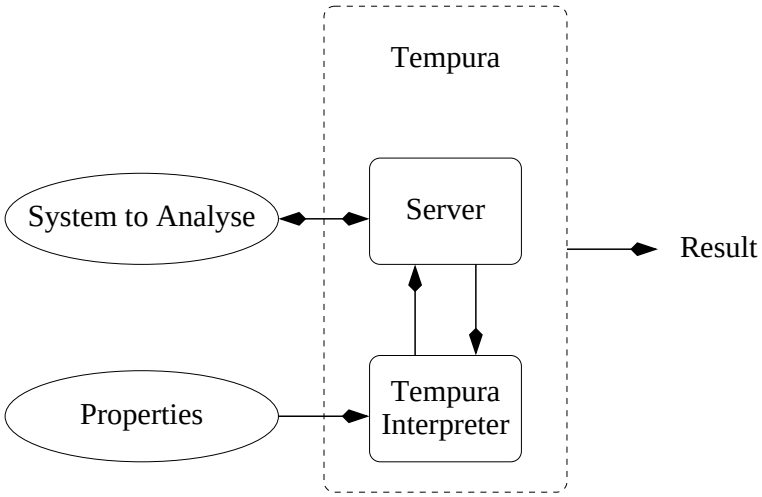


Figure 3.2: Basic Functions

I can use the existing tool Tempura [88], that supports the approach described above. Figure 3.2 shows the general structure of the tool. The inputs are the system description (either source code plus assertion points or an ITL specification) and the properties I want to check. The result of the analysis is whether the properties hold for the system. Optionally the behaviour of the system can be animated. Currently the tool can analyse C, Verilog and Tempura programs. The tool is available from [93] and several examples of the tool in action can be found in [12, 88].

3.7 Summary

I presented my overall strategy for codesign in this chapter. It integrates (co-)simulation with formal verification and refinement. Assumption/commitment pairs of formulas form the base of my methodology and this allows me to achieve a highly compositional formalism.

The foundation of my work is ITL. I give here the formal syntax and semantics for both ITL and its executable subset Tempura. I also argue the need for memory variables in ITL and Tempura and I propose a way to introduce such variables in the formalism. I also prove Theorem 1 which formally supports and justifies the construction of memory variables.

Also, I justify the claim that ITL has some powerful compositional expressiveness through assumption/commitment pairs. I give some compositional proof rules 3.3 and 3.4 and I show how they could be employed to prove some important liveness properties.

I choose stepwise refinement [13] as a major vehicle for development. It is well published and I include it here for completeness reasons.

Simulation and analysis are tightly integrated in my methodology. I present the basic functions of the Tempura tool with a general view of how it could be used to test for the validity some important properties.

Chapter 4

Denotational Semantics for Verilog

After justifying the need for an ITL based semantics for Verilog in the previous chapters, here I discuss the details of constructing such a semantics. I define a powerful and expressive core of the HDL and I give the translation between Verilog and $\mathcal{L}_T+$.

4.1 Introduction

The need for a formal and well founded semantics of a programming or hardware description language is widely accepted. I will consider two types of such semantics: *Denotational* and *Operational*. They reflect the duality of the usage of programming or hardware description languages, i.e. I need to both describe *properties* and *machines* which implement, or compute, these properties. This chapter treats the denotational semantics and Chapter 5 defines the operational semantics of Verilog.

It is a widely accepted that properties are best reasoned about in denotational terms and machines are best described by operational means. Of course, the correspondence between the denotational and operational semantics guarantees the uniformity of my formal reasoning and boosts my confidence in the trustworthiness of my work. The proof of the equivalence between the two semantics is given in chapter 6.

Modern hardware design is largely based on using HDLs and once I have the semantics of the HDL, formal verification comes within my reach. My approach to hardware/software codesign facilitates some existing technologies for hardware synthesis based on Verilog HDL. The transition between a TEMPURA-H specification and a Verilog specification (see figure 3.1 on page 27) must be based on sound technique such as refinement and the definition for the refinement relation must be

$$Spec_{\text{TEMPURA-H}} \sqsubseteq Spec_{\text{Verilog}}, \text{ iff } Spec_{\text{Verilog}} \supset Spec_{\text{TEMPURA-H}} \tag{4.1}$$

Obviously, the need for a sound definition of the last implication drives my desire for denotational, ITL-based, semantics for the Verilog HDL [21].

4.1.1 Verilog Specifics

Current HDLs raise considerable difficulties in understanding their semantics and Verilog is a typical example. There were several attempts [67, 30, 90, 89] for different styles of semantics for Verilog in the literature. Most of them compromise with the complexity

of the language, others choose to use several different semantics for different levels of abstraction. There are two main reasons for the lack of formal semantics of Verilog.

Concurrency Unlike most of the software, hardware is naturally concurrent. In a hardware system all subsystems work in parallel and this level of parallelism must be matched by the HDL of choice.

Reactivity Again in contrast to the software, hardware is reactive. The whole computation in hardware comes as a reaction to changes it the environment. Therefore the notion of event is explicit in hardware design.

Of course, there are some language specifics such as *event cancellation*, *blocking* vs *non-blocking* assignment as well as *continuous assignment*, which has earned Verilog a reputation of a *dirty* language with hard semantics [30].

4.1.2 Semantics Gap within Verilog HDL

As is well known, one can use several abstraction levels when developing a system in Verilog HDL. The *Verilog Formal Equivalence* project at Cambridge has been concentrating on different semantics for the different abstraction levels in Verilog [94, 30]. This differs from my work since I obtain a *unifying* semantics for the language. A major benefit of that is my ability to *compositionally* refine high level ITL specifications down to the RTL subset of Verilog.

Sagdeo and Thomas in [71, 80] give detailed design flows where the main stages are the following two:

1. *Behavioural design* — In a behavioural design one uses procedural constructs such as *begin-end* block, *always statement*, *event* control and so on. The general syntax of a behavioural specification includes *initial* and *always* statements, as well as *function* and *task* declarations. Since the declarations are instructions to the compiler only, I would not consider them here.
2. *RTL design* — RTL is a restricted subset of Verilog HDL. Data structures that can be continuously driven and statements that continuously drive them can be used. Only *continuous assignment* statements are RTL statements.

A third abstraction level is included in [71, 80], namely *Gate Level Verilog* or *Structural design*. The reason I do not consider structural descriptions is that there are commercially available synthesis tools which transform RTL down to netlists and this step has already been automated, via several commercially available synthesisers.

The restrictions imposed on RTL specifications imply their synthesisability. However, behavioural descriptions, including event controls and high level language constructs, are generally not synthesisable.

According to the design flow given in [71], designers have to transform the *Behavioural* description into *RTL* using their intuition and expert knowledge. At every step of the design process, simulations and tests are performed to check the correctness of the

transformations with respect to the requirements. Although these tests can be automated to a considerable degree, there are many cases when testing only does not provide the necessary level of correctness. More often than not, crucial test cases are overlooked which, in the case of critical systems, may result in human lives and/or money being lost. Probably the Pentium floating point flaw has the highest profile. According to [10, page 19] this error cost $475 million.

4.2 Syntax of Verilog

Here I define the syntax of the language I consider. Only a handful of convenient but non-essential constructs, such as function and task declarations, are not considered here. All constructs are given in Backus-Naur Form (BNF) style description and $\{\ldots\}^+$ denotes a non-empty repetition and **bool** is a boolean expression. Because of the specifics of Verilog I consider two syntactic categories namely *statement* and *atom*.

$statement$	::=	empty | η | block_assign | non_block_assign | event_trigger | if | while | begin_end
empty	::=	ε
η	::=	@ (e_exp) | # exp
block_assign	::=	$v = \eta\ exp \mid v = exp$
non_block_assign	::=	$v <= \#\ exp_1\ exp_2 \mid v <= exp$
event_trigger	::=	$\rightarrow event$
if	::=	`if` $(\mathbf{bool})\ statement$ `else` $statement$
while	::=	`while` $(\mathbf{bool})\ statement$
begin_end	::=	`begin` $\{statement;\}^+$ `end`

Table 4.1: Syntax of $statement$

A $statement$ is one of the sequential statements of Verilog. These are all statements one may find in a **begin_end** block for example. All statements are given in table 4.1. There e_exp is a boolean expression over event variables, exp is an expression, `bool` is a boolean and $event$ is an event variable normally declared as `event e;` in a Verilog program. The notation for time delays and event controls is standard.

$atom$ ::= assign | always | initial
assign ::= `assign` $v = exp$ | `assign` $\# exp_1\ v = exp_2$
always ::= `always` $statement$
initial ::= `initial` $statement$

Table 4.2: Syntax of $atom$

An $atom$ is the smallest unit of parallelism in Verilog. These are the continuous assignment, its delayed counterpart, always and initial constructs. Both Behavioural and RTL language constructs are included. Typically a Verilog program is a collection of atoms with appropriate variable declarations. All atoms run in parallel and share the variables as well as a common clock. With no loss of generality, we will accept the following general syntax of a Verilog program.

`program` ::= `module name` $(*)$;

$global\ variables$;

$atom_1$;

$atom_2$;

...

$atom_n$;

```
endmodule
```

Here $*$ is a shortcut for $v_1, \ldots, v_n$ and the Verilog operator ";" does not have the same semantics as the ITL chop but it is merely a statement separator. The following is an example of a Verilog program. I will come back to this program in section 4.3.4 where I will give its translation into $\mathcal{L}_T+$.

```
program ::= module example ;
                reg [3 : 0] a, b, i ;
                wire [3 : 0] v ;
                assign #5 v = a + b ;
                always begin
                    #10 ;
                    a <= #1 b ;
                    b <= #1 a ;
                end
                initial begin
                    b = 0 ;
                    a = 3 ;
                    i = 0 ;
                    while (i < 10) begin
                        #15;
```

```
            b = b + 1;
            i = i + 1;
        end
    end
endmodule
```

4.3 Mapping Verilog onto $\mathcal{L}_T+$

I define a function that translates Verilog constructs into $\mathcal{L}_T+$ equivalents, hence giving semantics for Verilog. The obtained semantics follows a declarative style. The language of $\mathcal{L}_T+$ is Tempura enriched with memory variables (see section 3.4.3).

Definition 1 *If* `statement` *is a valid Verilog statement, then* $\|$`statement`$\|$ *gives its* $\mathcal{L}_T+$ *equivalent.*

Suppose I have a Verilog specification. It defines a set of atoms with their variable declarations. Let the following be a Verilog program.

```
program ::= module name (*);
            global variables;
            atom_1;
            atom_2;
            ...
```

$atom_n$;

`endmodule`

Our general idea is to translate all atoms into $\mathcal{L}_T+$ formulas and combine them with the "$\wedge$" connective. In doing this, I will have to bear in mind that any Verilog program implicitly assumes several other properties such as a clock and non-blocking event scheduling, which I will have to explicitly specify as additional parallel atoms. Therefore, I will need to assume that all variables appearing in the left-hand side of a non-blocking assignment are $x_1, \ldots, x_m$. For each variable x_i and each individual atom $atom_j$ where x_i appears in such non-blocking assignment I will need a parallel atom $NB^{x_i}_{atom_j}$ and a memory list variable of non-blocking assignment events $\mathcal{L}^{x_i}_{atom_j}$.

$$
\begin{aligned}
&\|\texttt{program}\| ::= \\
&\quad \exists Atom_1.active, \ldots, Atom_n.active, \\
&\quad \mathcal{G}lobal\ \mathcal{V}ariables, Disable, \\
&\quad \mathcal{T}ime, \mathcal{L}^{x_1}_{atom_j} \ldots \mathcal{L}^{x_m}_{atom_k} \bullet \\
&\quad (\\
&\qquad \mathcal{G}lobal\ \mathcal{V}ariables = \bot \wedge clock(Disable) \wedge \mathit{inf} \wedge \\
&\qquad \Box(Disable = (Atom_1.active \vee \ldots \vee Atom_n.active)) \wedge \\
&\qquad \|atom_1\| \wedge \|atom_2\| \wedge \ldots \wedge \|atom_n\| \wedge \\
&\qquad NB^{x_1}_{atom_j} \wedge \ldots \wedge NB^{x_m}_{atom_k} \\
&\quad)
\end{aligned}
$$

I will simply use this specification as illustration only and later I will give a much more precise example in section 4.3.4 of a real Verilog program. All $\mathcal{G}lobal\ \mathcal{V}ariables$ and the $\mathcal{T}ime$ will be memory variables, whereas for each variable `v` from the global variables in the Verilog `program` I will have a memory variable $\mathcal{V}$ corresponding to it.

$Atom_i.active$ and the $Disable$ variables are normal state variables used for synchronisation and $Atom_i.active$ is a boolean variable which is true when $atom_i$ is active. The bottom " $\perp$ " is used to mark an undefined value.

4.3.1 Preliminaries

I will need to specify two important parts of our $\mathcal{L}_T+$ model. These are a clock and an event catching predicate on memory variables.

Explicit Clock I will have an atom called $clock$ which will keep the time in a global memory variable $\mathcal{T}ime$. I will need global clock for synchronisation.

$$
\begin{array}{l}
clock(Disable) \mathrel{\widehat{=}} \mathcal{T}ime \ = \ 0 \ \wedge \\
\quad (\\
\qquad \textsf{while}\ (Disable)\ \textsf{do skip}; \\
\qquad \mathcal{T}ime := \mathcal{T}ime + 1 \\
\quad)^*
\end{array}
$$

The clock in my specification has one parameter namely the state variable $Disable$ which synchronises all atoms. When $Disable$ is *true* then the clock is simply doing nothing, i.e.

its idle and waiting and some of atoms are busy working. Once all atoms are suspended, then $Disable$ turns into *false* and the clock advances the time.

Events For the memory variables $\mathcal{E}_1, \ldots, \mathcal{E}_n$ I can define the predicate $\sim$ as follows

$$\sim (\mathcal{E}_1, \ldots, \mathcal{E}_n) \mathrel{\widehat{=}} new_{\mathcal{E}_1} \vee \ldots \vee new_{\mathcal{E}_n}.$$

This predicate represents events on the memory variables involved.

4.3.2 *Statement* Semantics

I will start our semantics definition with the specification of all *statements* as given in table 4.1.

Empty Statement The empty statement ε is usually neglected. However, it is very important and I need to give it a semantic meaning. Let me assume that the ε statement, being a $statement$ from table 4.1, is included in an $atom$ with a status variable $Atom.active$. Then, I can write

$$\|\varepsilon\| \mathrel{\widehat{=}} (Atom.active = \textit{false}) \wedge \mathsf{skip}$$

Delays and Event Control Let me consider now the event control statement in Verilog, i.e. me assume here that the declarations in the Verilog program include e_i as variables

and for each of them I have a memory variable $\mathcal{E}_i$ in my $\mathcal{L}_T+$ formula.

$$\begin{aligned}&\|@\ (e_1\texttt{or}\ , \ldots , \texttt{or}\ e_n)\| \mathrel{\widehat{=}} \\ &\qquad \textsf{while}\ \neg \sim (\mathcal{E}_1, \ldots , \mathcal{E}_n)\ \textsf{do}\ (\|\varepsilon\|)\end{aligned}$$

Of course, the event control statement must be included in an $atom$ with a status variable $Atom.active$.

Similarly, the delay statement is a part of an atom with its $Atom.active$ variable.

$$\begin{aligned}&\|\#\ exp\| \mathrel{\widehat{=}} \\ &\qquad \exists t \bullet (\\ &\qquad\quad t = \mathcal{T}ime + exp \wedge \\ &\qquad\quad \textsf{while}\ \mathcal{T}ime < t\ \textsf{do}\ (\|\varepsilon\|) \\ &\qquad)\end{aligned}$$

Blocking Assignment The blocking assignment has two forms and again in each one of them I use the corresponding atom's status variable. The delayed version uses the semantics of η which shows the compositional nature of my denotational semantics.

$$\|\texttt{v}\ = exp\| \mathrel{\widehat{=}} ((Atom.active = \textsf{true}) \wedge \mathcal{V} := exp)$$

$$\|\texttt{v} \ = \eta \ exp\| \mathrel{\widehat{=}} \exists x \bullet (x = exp \wedge (\|\eta\| \ ; ((Atom.active = \textsf{true}) \wedge \mathcal{V} := x)))$$

Non-blocking Assignment With the non-blocking assignment, for every atom $atom$ and every variable v appearing in the lefthand side of a non-blocking assignment, I will create another atom NB^v_{Atom} which will run in parallel to the system and will have a memory list variable of non-blocking assignment events $\mathcal{L}^v_{Atom}$. Each non-blocking assignment in a normal atom will simply be substituted with update to this list, while the atom NB^v_{Atom} will simply execute this list when scheduled.

Again the non-blocking assignment has an immediate and a delayed form. Each one of them may appear in an $atom$, i.e. I have to take care of its status variable when updating the event list of non-blocking assignments $\mathcal{L}^v_{Atom}$. So, the following are the semantics of both forms of non-blocking assignment which assumes the additional atom NB^v_{Atom} running in parallel with the system. Later I give its specification.

$$\|\texttt{v} \ <= exp\| \mathrel{\widehat{=}} ((Atom.active = \textsf{true}) \wedge \mathcal{L}^v_{Atom} := \mathcal{L}^v_{Atom} + [[\mathcal{T}ime, exp]])$$

$$\|\texttt{v} \ <= \# \ exp_1 \ exp_2\| \mathrel{\widehat{=}} ((Atom.active = \textsf{true}) \wedge \mathcal{L}^v_{Atom} := \mathcal{L}^v_{Atom} + [[\mathcal{T}ime + exp_1, exp_2]])$$

As I mentioned earlier, the non-blocking assignment assumes a parallel atom NB^v_{Atom}. I have to specify such an atom for each variable appearing in the lefthand side of a non-blocking assignment.

$$
\begin{array}{l}
NB^v_{Atom} \mathrel{\widehat{=}} (\\
\quad \textsf{if } Disable \textsf{ then skip} \\
\quad \textsf{else } (\\
\quad \mathcal{L}^v_{Atom} := filter(\mathcal{L}^v_{Atom}, \mathcal{T}ime) \wedge \\
\quad \forall i < |\mathcal{L}^v_{Atom}| \bullet \textsf{if } (\mathcal{L}^v_{Atom}[i][0] = \mathcal{T}ime) \textsf{ then } \mathcal{V} := \mathcal{L}^v_{Atom}[i][1]))^*
\end{array}
$$

and $filter$ is defined as follows

$$
\begin{array}{l}
filter(\mathcal{L}, t) \mathrel{\widehat{=}} (\\
\quad \textsf{if } |\mathcal{L}| = 0 \textsf{ then } [] \\
\quad \textsf{else } (\textsf{if } \mathcal{L}[0][0] = t \textsf{ then } [] \textsf{ else } [\mathcal{L}[0]]) + filter(\mathcal{L}[1..|\mathcal{L}|], t))
\end{array}
$$

Event Trigger This construct is very easy. I just change the value of the variable in *some* way.

$$\| \rightarrow e\| \mathrel{\widehat{=}} \|e = e + 1\|$$

If, While and Begin_end Block Standard semantics for these constructs is adopted for obvious reasons. In the following, $\mathbf{b}$ is a boolean expression in the sense of table 3.2.

$$\|\texttt{if}\ (\mathbf{b})\ statement_1\ \texttt{else}\ statement_2\| \mathrel{\widehat{=}}$$

$$\quad \mathsf{if}\ (\|\mathbf{b}\|)\ \mathsf{then}\ \|statement_1\|\ \mathsf{else}\ \|statement_2\|$$

$$\|\texttt{while}\ (\mathbf{b})\ statement\| \mathrel{\widehat{=}}$$

$$\quad \mathsf{while}\ (\|\mathbf{b}\|)\ \mathsf{do}\ \|statement\|$$

$$\|\texttt{begin}\ statement_1\ ;\ (statement)\ \texttt{end}\ \| \mathrel{\widehat{=}}$$

$$\quad (\|statement_1\|\ ;\ \|\texttt{begin}\ (statement)\ \texttt{end}\ \|)$$

4.3.3 *Atom* Semantics

I now turn our attention to the atoms in table 4.2.

Assign As shown in table 4.2, the form of the `assign` statement has two forms and here I will start with the simpler one.

$$\|\texttt{assign}\ \ v = exp(*)\| \mathrel{\widehat{=}}$$

$$\quad (\Box(Atom.active =\!\sim (*)) \wedge$$

$$\quad\ (\mathsf{if}\ \neg Atom.active\ \mathsf{then\ skip\ else}\ \mathcal{V} := exp(*))^*)$$

Informally I have two $\mathcal{L}_T+$ statements, namely $\Box(Atom.active =\sim (*))$ and $(\ldots)^*$ working in parallel. The first one turns the $Atom.active$ variable into *true* any time when there is a change in the variables in the expression on the righthand side of the assignment, i.e. it activates the atom at those time points, while the other statement checks if the atom is active and, if it is, it executes the assignment. Otherwise it stays idle and does a skip only.

The second form of the `assign` statement has a much more complex semantics.

$$
\begin{array}{l}
\|\texttt{assign}\ \#\texttt{exp}_1\ \ v = exp_2(*)\| \mathrel{\widehat{=}} \\
\quad \exists \mathcal{T}, Atom.act_1, Atom.act_2 \bullet \\
\quad\ (\Box(Atom.active = Atom.act_1 \vee Atom.act_2) \wedge \\
\quad\quad (\textsf{if } \sim (*) \textsf{ then } (Atom.act_1 = \textit{true} \wedge \mathcal{T} := \mathcal{T}ime + exp_1) \\
\quad\quad\ \textsf{else } (Atom.act_1 = false \wedge \textsf{skip}) \wedge \\
\quad\quad\ \textsf{if } \mathcal{T} = \mathcal{T}ime \textsf{ then } (Atom.act_2 = \textit{true} \wedge \mathcal{V} := exp_2(*)) \\
\quad\quad\ \textsf{else } (Atom.act_2 = false \wedge \textsf{skip}) \\
\quad\quad)^* \\
\quad\ )
\end{array}
$$

Here I have three parallel statements in the semantics. Again I have an $\textsf{if } \sim (*) \textsf{ then } \ldots$ statement which watches for any changes in the variables in the expression exp_2 and should change occurs, it switches the local variable $Atom.act_1$ to *true* and assigns new

value to the local variable $\mathcal{T}$, i.e. it reschedules the assignment for some time in the future. Otherwise, it keeps $Atom.act_1$ to *false* and does skip, i.e. it stays idle.

The second if $\mathcal{T} = \mathcal{T}ime$ then ... statement watches the time when the assignment is scheduled and executes it when it elapses.

The third statement is $\Box(Atom.active = Atom.act_1 \vee Atom.act_2)$ which keeps the whole atom active if any part of it is active, i.e. it synchronises with the global clock and the other atoms in the system.

Always and Initial The semantics of the `always` atom is *iteration* while the semantics of `initial` is a single execution.

$$\|\texttt{always}\ statement\| \mathrel{\widehat{=}} \|statement\|^*$$

With the `initial` atom I only have to bear in mind one small complication. This is, the `initial` atom is running in parallel with all the rest and therefore I need its status variable looked after even when all statements of the atom itself are long gone. Hence I use $\Box Atom.active = \textit{false}$ sequentially composed at the end of the semantics of the statement.

$$\|\texttt{initial}\ statement\| \mathrel{\widehat{=}} \|statement\| \,;\, \Box(Atom.active = \textit{false})$$

4.3.4 Example

Let me consider again the Verilog program that I introduced in section 4.2.

```
program ::= module example ;
                reg [3 : 0] a, b, i ;
                wire [3 : 0] v ;
                assign #5 v = a + b ;
                always begin
                    #10 ;
                    a <= #1 b ;
                    b <= #1 a ;
                end
                initial begin
                    b = 0 ; a = 3 ;
                    i = 0 ;
                    while (i < 10) begin
                        #15 ;
                        b = b + 1 ;
                        i = i + 1 ;
                    end
                end
```

```
endmodule
```

For `program` I will construct the $\mathcal{L}_T+$ specification which will be its semantics. Knowing the nature of each atom I will name them as $assign$, $always$ and $initial$. Therefore, their corresponding $active$ variables will be named $Assign.active$, $Always.active$, $Initial.active$.

$$
\begin{aligned}
&\|\texttt{program}\| = \\
&\quad \exists Assign.active, Always.active, Initial.active, Disable, \\
&\quad \mathcal{A}, \mathcal{B}, \mathcal{I}, \mathcal{T}ime, \\
&\quad \mathcal{L}^{\mathcal{A}}_{Always}, \mathcal{L}^{\mathcal{B}}_{Always} \bullet \\
&\quad (\mathcal{A} = \mathcal{B} = \mathcal{I} = \mathcal{V} = \bot \wedge \mathcal{L}^{\mathcal{A}}_{Assign} = \mathcal{L}^{\mathcal{B}}_{Assign} = [] \wedge clock(Disable) \wedge \\
&\quad \Box(Disable = (Assign.active \vee Always.active \vee Initial.active)) \wedge \\
&\quad \mathit{inf} \wedge \|\texttt{assign}\| \wedge \|\texttt{always}\| \wedge \|\texttt{initial}\| \wedge \\
&\quad NB^{\mathcal{A}}_{\texttt{always}} \wedge NB^{\mathcal{B}}_{\texttt{always}} \\
&\quad)
\end{aligned}
$$

Having in mind that the $clock$ has been specified previously, I will go straight to the specifications for $\|\texttt{assign}\|$, $\|\texttt{always}\|$, $\|\texttt{initial}\|$, $NB^{\mathcal{A}}_{\texttt{always}}$ and $NB^{\mathcal{B}}_{\texttt{always}}$. According to the definitions above these specification will be as follows.

The `assign` atom takes care of its $active$ variable $Assign.active$ which is a global state variable for the whole formula and is the synchronising link between the atom itself and the clock.

$$
\begin{aligned}
&\|\texttt{assign}\| = \\
&\quad \exists \mathcal{T}, Assign.act_1, Assign.act_2 \bullet \\
&\quad\quad (\Box(Assign.active = (Assign.act_1 \vee Assign.act_2)) \wedge \\
&\quad\quad\quad (\textsf{if } \sim(\mathcal{A}, \mathcal{B}) \textsf{ then } (Atom.act_1 = \textit{true} \wedge \mathcal{T} := \mathcal{T}ime + 5) \\
&\quad\quad\quad\quad \textsf{else } (Atom.act_1 = false \wedge \textsf{skip}) \wedge \\
&\quad\quad\quad\quad \textsf{if } \mathcal{T} = \mathcal{T}ime \textsf{ then } (Atom.act_2 = \textit{true} \wedge \mathcal{V} := \mathcal{A} + \mathcal{B}) \\
&\quad\quad\quad\quad \textsf{else } (Atom.act_2 = false \wedge \textsf{skip}))^*)
\end{aligned}
$$

The next atom is the `always` atom. It also looks after its global $Always.active$ variable and it sequentially composes several other behavioural statements. The following $\mathcal{L}_T+$ specification also sequentially composes their equivalents.

$$
\begin{aligned}
&\|\texttt{always}\| = \\
&\quad (\\
&\quad\quad \exists t \bullet (\\
&\quad\quad\quad t = \mathcal{T}ime + 10 \wedge
\end{aligned}
$$

$$
\begin{array}{l}
\quad \textsf{while } \mathcal{T}ime < t \textsf{ do } ((Always.active = \textit{false}) \wedge \textsf{skip}) \\
); \\
((Always.active = \textit{true}) \wedge \\
\quad \mathcal{L}^{\mathcal{A}}_{Always} := \mathcal{L}^{\mathcal{A}}_{Always} + [[\mathcal{T}ime + 1, \mathcal{B}]] \\
); \\
((Always.active = \textit{true}) \wedge \\
\quad \mathcal{L}^{\mathcal{B}}_{Always} := \mathcal{L}^{\mathcal{B}}_{Always} + [[\mathcal{T}ime + 1, \mathcal{A}]] \\
); \\
)^*
\end{array}
$$

Next comes the semantics for the `initial` atom which also is a sequential composition of several behavioural statements. Those have been translated in $\mathcal{L}_T+$ and their semantics have been sequentially composed.

$$
\begin{array}{l}
\|\texttt{initial}\ \| = \\
\quad (\\
\quad\quad (Initial.active = \textit{true} \wedge \mathcal{B} := 0); \\
\quad\quad (Initial.active = \textit{true} \wedge \mathcal{A} := 3); \\
\quad\quad (Initial.active = \textit{true} \wedge \mathcal{I} := 3);
\end{array}
$$

$$
\begin{array}{l}
\text{while } (\mathcal{I} < 10) \text{ do } (\\
\quad \exists t \bullet (\\
\quad\quad t = \mathcal{T}ime + 15 \wedge \\
\quad\quad\quad \text{while } \mathcal{T}ime < t \text{ do } ((Initial.active = \textit{false}) \wedge \text{skip}) \\
\quad); \\
\quad (Initial.active = \textit{true} \wedge \mathcal{B} := \mathcal{B} + 1); \\
\quad (Initial.active = \textit{true} \wedge \mathcal{I} := \mathcal{I} + 1); \\
); \\
) \,;\, \Box(Initial.active = \textit{false})
\end{array}
$$

Now I will need to turn my attention on the atoms that implement the behaviour of the non-blocking assignments. They are

$$
\begin{array}{l}
NB^{\mathcal{A}}_{\texttt{always}} = (\\
\quad \text{if } Disable \text{ then skip} \\
\quad \text{else } (\mathcal{L}^{\mathcal{A}}_{\texttt{always}} := filter(\mathcal{L}^{\mathcal{A}}_{\texttt{always}}, \mathcal{T}ime) \wedge \\
\quad \forall i < |\mathcal{L}^{\mathcal{A}}_{\texttt{always}}| \bullet \text{if } (\mathcal{L}^{\mathcal{A}}_{\texttt{always}}[i][0] = \mathcal{T}ime) \text{ then } \mathcal{A} := \mathcal{L}^{\mathcal{A}}_{\texttt{always}}[i][1]) \\
)^*
\end{array}
$$

Of course the same stands for $NB^{\mathcal{B}}_{\texttt{always}}$

$$
\begin{aligned}
NB^{\mathcal{B}}_{\texttt{always}} = (\ & \\
& \textsf{if } Disable \textsf{ then skip} \\
& \textsf{else } (\mathcal{L}^{\mathcal{B}}_{\texttt{always}} := filter(\mathcal{L}^{\mathcal{B}}_{\texttt{always}}, \mathcal{T}ime) \wedge \\
& \forall i < |\mathcal{L}^{\mathcal{B}}_{\texttt{always}}| \bullet \textsf{if } (\mathcal{L}^{\mathcal{B}}_{\texttt{always}}[i][0] = \mathcal{T}ime) \textsf{ then } \mathcal{B} := \mathcal{L}^{\mathcal{B}}_{\texttt{always}}[i][1]) \\
)^*
\end{aligned}
$$

and the function $filter$ has been defined previously.

Now, after I have obtained the denotational semantics for `program` , I can show some simple properties about it. In the following I will use $\sigma \models f$ instead of $\mathcal{M}_\sigma[\![f]\!] = \text{tt}$ as given in section 3.3.2. For example, I can quickly prove the following implication

$$
\begin{aligned}
\sim(\mathcal{A}, \mathcal{B}) \wedge \textsf{skip} ; \boxed{a} \neg \sim(\mathcal{A}, \mathcal{B}) \wedge \Box\Diamond \mathcal{T}ime := \mathcal{T}ime + 1 \wedge & \\
\|\texttt{assign}\| \quad \supset \quad \Diamond \mathcal{V} := \mathcal{A} + \mathcal{B} &
\end{aligned}
$$

which says that if $\mathcal{A}$ or $\mathcal{B}$ change and then they stay the unchanged for sufficiently long time and if the time continues to tick, then the `assign` statement assigns the appropriate values to the appropriate variable. The argument is very simple. Let me say that prerequisite holds for an interval σ. This means that in the first state $\sigma_0 \models \sim(\mathcal{A}, \mathcal{B})$ which leads

to

$$\sigma_0 \models \textsf{if } \sim(\mathcal{A},\mathcal{B}) \textsf{ then } (Atom.act_1 = \textit{true} \wedge \mathcal{T} := \mathcal{T}ime + 5)$$
$$\textsf{else } (Atom.act_1 = false \wedge \textsf{skip}) \equiv$$
$$(Atom.act_1 = \textit{true} \wedge \mathcal{T} := \mathcal{T}ime + 5)$$

However, since in the prerequisite I also have $\textsf{skip} ; \boxed{a} \neg \sim(\mathcal{A},\mathcal{B})$, I can say that for the rest of the interval

$$\sigma_1, \ldots \models \textsf{if } \sim(\mathcal{A},\mathcal{B}) \textsf{ then } (Atom.act_1 = \textit{true} \wedge \mathcal{T} := \mathcal{T}ime + 5)$$
$$\textsf{else } (Atom.act_1 = false \wedge \textsf{skip}) \equiv$$
$$(Atom.act_1 = false \wedge \textsf{skip})$$

So, I know that $\sigma_1 \models \mathcal{T} = \mathcal{T}ime + 5$, $\mathcal{T}$ is a local variable and it is not changed anywhere else, i.e. its value will be stable and $\sigma \models \Box\Diamond\mathcal{T}ime := \mathcal{T}ime + 1$, therefore I can conclude $\sigma \models \Diamond\mathcal{T} = \mathcal{T}ime$, i.e.

$$\sigma \models \Diamond(\textsf{if } \mathcal{T} = \mathcal{T}ime \textsf{ then } (Atom.act_2 = \textit{true} \wedge \mathcal{V} := \mathcal{A} + \mathcal{B})$$

$$\textsf{else } (Atom.act_2 = false \wedge \textsf{skip}) \quad \equiv$$

$$(Atom.act_2 = \textit{true} \wedge \mathcal{V} := \mathcal{A} + \mathcal{B}))$$

which proves the implication.

This simple exercise demonstrates that the semantics for the `assign` statement indeed assigns the appropriate expression to the appropriate variable. The usefulness of the denotational semantics is self-evident when properties of Verilog programs need to be proven.

4.4 Summary

In this chapter I have defined the syntax of Verilog HDL and I gave the denotational semantics for it by translating all Verilog programs into $\mathcal{L}_T+$. All constructs of the language are treated and a small example is provided. At the end I proved a simple property about the denotational semantics of the sample program from the example. This supports the usefulness of my denotational semantics.

Chapter 5

Operational Semantics for Verilog

Here I present my operational semantics for Verilog and I support my findings with several healthiness conditions. I conclude with an example of simulation which represents a run of the operational semantics over a simple Verilog program.

5.1 Introduction

As I mentioned earlier, the duality of a programming or hardware description language consists of the necessity for describing both properties and machines that compute these properties. This results in duality of the underlying semantics for my language of choice, in my case, the Verilog HDL. In chapter 4 I gave a denotational semantics for Verilog based on $\mathcal{L}_T+$, my conservative extension of Tempura while here I will focus on the other aspect of the language — the operational semantics of Verilog. Of course, later on

in chapter 6 I will show the uniformity in my two approaches by proving the equivalence between the denotational and the operational semantics.

Many *difficult* language constructs are treated in both the Behavioural and RTL parts. The approach used allows quick expansion of the treated language up to the limits of full Verilog as described in [80, 71] and beyond.

The semantics is readily implementable into a real simulator and, unlike most current Verilog simulators, it is *fully* parallel which eliminates all side effects caused by implementing parallelism via non-determinism in Verilog. I have also included a small example which supports the usefulness and the practical benefit of my semantics.

The correctness of the proposed semantics is backed by several healthiness conditions. These conditions are by no means complete and can easily be extended.

A recent paper [89] on semantics for Verilog has considered a true subset of the language. In fact, the subset described there is smaller than Behavioural Verilog and does not mention RTL at all.

The Verilog Formal Equivalence Project at [94, 30] studies different semantics at different abstraction levels of Verilog HDL. Although the framework there is consistent and sound, the differences in the semantics results in differences in the specification languages and this hampers the refinement. The subset V0 studied in this project is a considerable advance including Behavioural as well as RTL constructs. However, repeatedly I see lack of attention devoted to very difficult specifics of Verilog like *delayed continuous assignment* and *delayed procedural assignment*.

5.2 My Contribution

Here I consider a powerful and expressive core of Verilog given in the tables 4.1 on page 58 and 4.2 on page 59. Only a handful of convenient but non-essential constructs such as `case` -like statements and function and task declarations are omitted. I would like to add here that these constructs have very trivial semantics and can be incorporated easily into my framework. In contrast, all significant *Behavioural* and *RTL* constructs are included and this gives a solid basis for the whole language with no semantic gaps.

A major benefit of an operational style semantics for Verilog is the applicability of my work as a *roadmap* for a simulator of the language. The semantics described here is *fully* parallel which differs from most of the available simulators. The Verilog simulators "in-use" at the moment implement parallelism via non-determinism. This approach has obvious side effects and all texts on Verilog strongly discourage the active use of the sequential nature of the simulators. A truly parallel operational semantics to Verilog eliminates these side effects.

As well as giving an example of simulation later, I seek reassurance in the validity of my work here by proving some *healthiness conditions* and in addition to this I show the correspondence between the denotational and the operational semantics given here. The healthiness conditions given at the end of the chapter are by no means complete and can be expanded if necessary.

5.3 Structural Operational Semantics

Following Plotkin [68] and Mosses [59] I use the notions of Structural Operational Semantics (SOS) and Labelled Transition Systems (LTS) to describe the semantics of Verilog. Conventional SOS employs LTS which in turn is a structure of a configuration set, a set of terminal configurations, a set of labels and a transition relation. In my framework we define a *configuration*, and a *step function* which represents the transition relation.

The absence of an explicit set of terminal configurations represents my belief that hardware, which Verilog models, has no "terminal" state. Computation in hardware is modelled as a sequence of configurations and the sole aim is to achieve a "stable" configuration. If such a configuration is achieved, the system simply keeps the state unchanged (stuttering). In this sense, "terminal" configuration for hardware is equivalent to power failure.

I also note that instead of a set of labels which determines the transition relation, I use two functions namely $head$ and $tail$. Their purpose is to play the role of a parser for the language. Fundamental to this approach is that it facilitates a possible implementation of this operational semantics into a real simulator.

5.4 Operational Semantics of Verilog

The syntax of Verilog was given by tables 4.1 and 4.2. However, in the following, I will need yet another syntactic category, namely an $\mathcal{A}tom$ which is needed in the technical

$\mathcal{A}$tom ::= $statement$ | atom | $statement$; $\mathcal{A}tom$

Table 5.1: Syntax of $\mathcal{A}tom$

details of the definition of operational semantics. In essence, an $\mathcal{A}tom$ represents an $atom$ being executed. It gives me a partially completed $atom$ stripped out of the statements that have been finished.

As mentioned earlier, I need to define a set of configurations.

Definition 2 A configuration will be denoted by

$$(T, ((\mathcal{A}tom_1, bool_1), .., (\mathcal{A}tom_n, bool_n)), V, (E_1, .., E_n))$$

where:

- T is a *time variable*, i.e. a clock
- $\mathcal{A}tom_i$ is what is left of the *i*-th atom to be executed
- $bool_i$ is a boolean expression indicating if $\mathcal{A}tom_i$ will schedule any events for the next configuration
- $V = \{(variable, value, new)\}$ is a set variables and
  - $variable$ is a name,
  - $value$ is its associated current value and
  - new is a boolean indicating if this particular variable has a new value.

- $E_i = (E_i^A, E_i^{NB})$ is a tuple of *Active* and *Non-Blocking* events for the *i*-th atom
- $E_i^j = \{(when, what)\}$ a finite set of tuples representing the event and
  - $when$ is a boolean expression, a guard of the event and
  - $what$ is Verilog statement to be executed at the event.

In a configuration, the time variable and the set V form the system's state. It might be of interest to mention here the intuition behind an event of the type $(when, \varepsilon)$ which represents an empty event. For each configuration I define an interpretation of the variables, such that $\|variable\| = value$, where $(variable, value, new) \in V$. This allows me to calculate the expressions in Verilog. Also, $\|.\|$ defines a set function $\uparrow (E)$ giving the set of enabled events in E. An event $e \in E$ is enabled in a configuration c, iff $\|when(e)\| =$ ***true*** and it is disabled otherwise. I denote $\uparrow (E) = \{e | e \in E \wedge \|when(e)\| =$ ***true***$\}$ and $\downarrow (E) = E \setminus \uparrow (E)$.

For every Verilog program with atoms $atom_i$ I need to define a start configuration. This would represent the configuration at time 0 and is as follows

$$(0, ((atom_1, \textit{true}), \ldots, (atom_n, \textit{true})), \{ \perp \}, (\emptyset^2, \ldots, \emptyset^2)).$$

Here $\{ \perp \}$ is a set of variables with ***undefined*** values. It represents the `x` value in a Verilog simulator.

I will need two functions $head$ and $tail$ to define the step. The function $head$ gives

me the current statement to be executed. In the following η is defined in table 4.1.

$head : \mathcal{A}tom \rightarrow \{\varepsilon,$ block_assign, non_block_assign, $\eta,$ `assign` $\}$

$head(\varepsilon) \mathrel{\widehat{=}} \varepsilon$

$head($`assign` **body**$) \mathrel{\widehat{=}}$ `assign` , where **body** is the body of the assign statement

$head($`always` $statement) \mathrel{\widehat{=}} head(statement)$

$head($`initial` $statement) \mathrel{\widehat{=}} head(statement)$

$head(statement\ ; \mathcal{A}tom) \mathrel{\widehat{=}} head(statement)$

$head(\eta) \mathrel{\widehat{=}} \eta$

$head(v = exp) \mathrel{\widehat{=}} v = exp$

$head(v = \eta\ exp) \mathrel{\widehat{=}} v = \eta\ exp$

$head(v <= exp) \mathrel{\widehat{=}} v <= exp$

$head(v <= \#exp_1\ exp_2) \mathrel{\widehat{=}} v <= \#exp_1\ exp_2$

$head(\rightarrow event) \mathrel{\widehat{=}} event = event + 1$

$head($`if` $($`bool`$)$ $statement_1$ `else` $statement_2) \mathrel{\widehat{=}}$

$$\mathrel{\widehat{=}} \begin{cases} head(statement_1), & \text{iff } \|\texttt{bool}\| = \textit{true} \\ head(statement_2), & \text{otherwise} \end{cases}$$

$$head(\texttt{while}\ (\texttt{bool})\ statement) \mathrel{\widehat{=}} \begin{cases} head(statement), & \text{iff } \|\texttt{bool}\| = \textit{true} \\ \varepsilon, & \text{otherwise} \end{cases}$$

$head($`begin` $statement_1\ ;\ \{statement\}$ `end` $) \mathrel{\widehat{=}} head(statement_1)$

When the current statement has been identified I will need to remove it and this is what $tail$ does. It simply takes the rest of the $\mathcal{A}tom$ and prepares it for the next iteration.

$tail : \mathcal{A}tom \rightarrow \mathcal{A}tom$

$tail(\varepsilon) \mathrel{\widehat{=}} \varepsilon$

$tail(\texttt{assign}\ \textbf{body}) \mathrel{\widehat{=}} \texttt{assign}$, where **body** is the body of the assign statement

$tail(\texttt{always}\ \ statement) \mathrel{\widehat{=}} tail(statement)\ ;\ \texttt{always}\ \ statement$

$tail(\texttt{initial}\ \ statement) \mathrel{\widehat{=}} tail(statement)$

$tail(statement\ ;\ \mathcal{A}tom) \mathrel{\widehat{=}} tail(statement)\ ;\ \mathcal{A}tom$

$tail(\eta) \mathrel{\widehat{=}} \varepsilon$

$tail(v = exp) \mathrel{\widehat{=}} \varepsilon$

$tail(v = \eta\ exp) \mathrel{\widehat{=}} \varepsilon$

$tail(v <= exp) \mathrel{\widehat{=}} \varepsilon$

$tail(v <= \#exp_1\ exp_2) \mathrel{\widehat{=}} \varepsilon$

$tail(\rightarrow event) \mathrel{\widehat{=}} \varepsilon$

$tail(\texttt{if}\ \ (\texttt{bool})\ \ statement_1\ \texttt{else}\ \ statement_2) \mathrel{\widehat{=}}$

$$\mathrel{\widehat{=}} \begin{cases} tail(statement_1), & \text{iff } \|\texttt{bool}\| = \textit{true} \\ tail(statement_2), & \text{otherwise} \end{cases}$$

$$tail(\texttt{while}\ (\texttt{bool})\ statement) \mathrel{\widehat{=}}$$

$$\mathrel{\widehat{=}} \begin{cases} tail(statement)\,;\texttt{while}\ (\texttt{bool})\ statement, & \text{iff } \|\texttt{bool}\| = \textit{true} \\ \varepsilon, & \text{otherwise} \end{cases}$$

$$tail(\texttt{begin}\ statement_1\ ;\ \{statement\}\ \texttt{end}\ ) \mathrel{\widehat{=}} tail(statement_1)\ ;\ \{statement\}$$

The $head$ and $tail$ functions are in effect a *parser* for Verilog.

Let me now have a configuration

$$c = (T, ((\mathcal{A}tom_1, bool_1), \ldots, (\mathcal{A}tom_n, bool_n)), V, (E_1, \ldots, E_n)).$$

For c I have also an interpretation $\|.\|$ defined by the set V and a set function $\uparrow(E)$ giving the set of enabled events in E. $\uparrow(.)$ uses $\|.\|$ to evaluate the event guards $when(.)$. For that setup I will define four step tests.

- $\tau_0 \mathrel{\widehat{=}} \uparrow(\bigcup_{i,j} E_i^j) = \emptyset \wedge \bigvee_i \|bool_i\|$
- $\tau_1 \mathrel{\widehat{=}} \uparrow(\bigcup_{i,j} E_i^j) = \emptyset \wedge \bigwedge_i \neg\|bool_i\|$
- $\tau_2 \mathrel{\widehat{=}} \uparrow(\bigcup_i E_i^A) = \emptyset \wedge \uparrow(\bigcup_i E_i^{NB}) \neq \emptyset$
- $\tau_3 \mathrel{\widehat{=}} \uparrow(\bigcup_i E_i^A) \neq \emptyset$

The step tests determine the type of each step. As it will become clear later, τ_0 is true in the start configuration only, τ_1 picks up time advancing steps, τ_2 is for the activation

of events associated with non-blocking assignments and τ_3 is for the actual computation steps.

Before I define the transition step, I will give some notation. If v_i are all variables in $exp(v_0, \ldots, v_n)$, then I will write $exp(*)$ instead. The special predicate $\sim$ is true over variables which have just changed their values and is defined as follows:

- $\sim (v_0) \Leftrightarrow new_{v_0} = \mathit{true}$, where $(v_0, value, new_{v_0}) \in V$.
- $\sim (v_0, \ldots, v_n) \Leftrightarrow \sim (v_0, \ldots, v_{n-1}) \vee \sim (v_n)$.

The transition step function $Step(c)$ must find a successor for c and is defined as follows.

- If $\tau_1 = true$, then

$$Step(c) = (T + 1, ((\mathcal{A}tom_1, bool_1), \ldots, (\mathcal{A}tom_n, bool_n)), V, (E_1, \ldots, E_n))$$

- If $\tau_2 = true$, then

$$Step(c) = (T, ((\mathcal{A}tom_1, bool_1), \ldots, (\mathcal{A}tom_n, bool_n)), V, (E'_1, \ldots, E'_n)),$$

  where $E'_i = (E_i^A \cup \uparrow (E_i^{NB}), \downarrow (E_i^{NB}))$

- If $\tau_0 \vee \tau_3 = true$, then

$$Step(c) = (T, ((\mathcal{A}tom'_1, bool'_1), \ldots, (\mathcal{A}tom'_n, bool'_n)), V', (E'_1, \ldots, E'_n)),$$

where

$$\mathcal{A}tom'_i = \begin{cases} \mathcal{A}tom_i, & \text{if } \neg\|bool_i\| \\ tail(\mathcal{A}tom_i), & \text{otherwise} \end{cases}$$

and

- if $\neg\|bool_i\|$, then $bool'_i = bool_i$
- if $\|bool_i\| \wedge head(\mathcal{A}tom_i) \eqq \varepsilon$[1], then $bool'_i = false$
- if $\|bool_i\| \wedge head(\mathcal{A}tom_i) \eqq$ `assign` $v = exp(*)$, then $bool'_i =\sim (*)$
- if $\|bool_i\| \wedge head(\mathcal{A}tom_i) \eqq$ `assign` $\# exp_1(*_1)\ v = exp(*)$, then $bool'_i =\sim (*)$
- if $\|bool_i\| \wedge head(\mathcal{A}tom_i) \eqq \# exp$, then $bool'_i = (T = \|T + exp\|)$
- if $\|bool_i\| \wedge head(\mathcal{A}tom_i) \eqq @\ e\_exp(*)$, then $bool'_i =\sim (*)$
- if $\|bool_i\| \wedge head(\mathcal{A}tom_i) \eqq v = exp$, then $bool'_i = (T = \|T\|)$
- if $\|bool_i\| \wedge head(\mathcal{A}tom_i) \eqq v = \# exp_1\ exp$, then $bool'_i = (T = \|T + exp_1\|)$
- if $\|bool_i\| \wedge head(\mathcal{A}tom_i) \eqq v = @\ (exp_1(*))\ exp$, then $bool'_i =\sim (*)$
- if $\|bool_i\| \wedge head(\mathcal{A}tom_i) \eqq v <= exp$, then $bool'_i = (T = \|T\|)$
- if $\|bool_i\| \wedge head(\mathcal{A}tom_i) \eqq v <= \#exp_1\ exp_2$, then $bool'_i = (T = \|T\|)$

In the previous, all expressions like $T = \|T + exp\|$ are syntactic, i.e. I calculate the constant $\hat{t} = \|T + exp\|$ using the interpretation $\|.\|$ for the set V, and then I

[1] $\eqq$ means "graphically equal"

construct the string $T = \hat{t}$.

I calculate V' from V and $(E_1, \ldots, E_n)$. For every triple $(variable, value', new')$ I define

$$(value', new') = \begin{cases} (\|exp\|, true), & \text{if } \exists e \bullet e \in \uparrow (\bigcup_i E_i^A) \wedge \\ & what(e) = variable = exp \wedge \\ & value \neq \|exp\| \\ (value, false), & \text{otherwise} \end{cases}$$

At last, I need to define E_i' which consists of the pair of $E_i^{A'}$ and $E_i^{NB'}$. I will need to consider several cases when constructing $E_i^{A'}$. If $\neg\|bool_i\|$, then $E_i^{A'}$ will keep the old $\downarrow (E_i^A)$, i.e. $E_i^{A'} = \downarrow (E_i^A)$. However, if $\|bool_i\|$, then $E_i^{A'} = \downarrow (E_i^A) \cup S$ where S will be given below.

- if $head(\mathcal{A}tom_i) = \varepsilon$, then $S = \{(\mathit{false}, \varepsilon)\}$
- if $head(\mathcal{A}tom_i) =$ `assign` $v = exp(*)$, then $S = \{(\sim (*), v = exp(*))\}$
- if $head(\mathcal{A}tom_i) =$ `assign` $\# exp_1(*_1)\ v = exp(*)$, then $S = \{(\sim (*), t_j = \|T + exp_1\|), (T = t_j, v = exp(*))\}$
- if $head(\mathcal{A}tom_i) = \# exp$, then $S = \{(T = \|T + exp\|, \varepsilon)\}$
- if $head(\mathcal{A}tom_i) = @\ e\_exp(*)$, then $S = \{(\sim (*), \varepsilon)\}$
- if $head(\mathcal{A}tom_i) = v = exp$, then $S = \{(T = \|T\|, v = \|exp\|)\}$

- if $head(\mathcal{A}tom_i) = v = \#\ exp_1\ exp$, then $S = \{(T = \|T + exp_1\|, v = \|exp\|)\}$
- if $head(\mathcal{A}tom_i) = v = @\ (exp_1(*))\ exp$, then $S = \{(\sim (*), v = \|exp\|)\}$
- if $head(\mathcal{A}tom_i) = v <= exp$, then $S = \{(T = \|T\|, \varepsilon)\}$
- if $head(\mathcal{A}tom_i) = v <= \#exp_1\ exp_2$, then $S = \{(T = \|T\|, \varepsilon)\}$

At the end I need to define E_i^{NB}. Similarly to the definition of $E_i^{A'}$, I will consider several cases. If $\neg\|bool_i\|$, then $E_i^{A'} = E_i^{NB}$ and if $\|bool_i\|$, then $E_i^{A'} = E_i^{NB} \cup S$, where S will be defined below.

$$S = \begin{cases} \{(T = \|T\|, v = \|exp\|)\}, & \text{if } head(\mathcal{A}tom_i) = v <= exp, \\ \{(T = \|T + exp_1\|, & \\ \quad v = \|exp\|)\} & \text{if } head(\mathcal{A}tom_i) = v <= \#\ exp_1\ exp \\ \emptyset, & \text{otherwise} \end{cases}$$

Let me give some intuition for the definition of $Step$. The cases when $\tau_1 \vee \tau_2 =$ *true* are clear. The first one is a time advancing step and in the second we simply activate all enabled events from the non-blocking assignment list.

There is more action in the actual computation step when $\tau_0 \vee \tau_3 =$ *true*. The intuition behind $bool_i =$ *true* is that $atom_i$ is active and it schedules events for the next configuration. Therefore, when $\neg\|bool_i\|$ I simply execute those events which the atom has scheduled in the past and are enabled at the moment.

However, when $\|bool_i\|$ I need to consume the current statement of the atom in $head$.

That is why the new $\mathcal{A}tom$ gets the tail of the current $\mathcal{A}tom$ and pending on $head$ I determine the new boolean $bool_i'$ and the new event lists $E_i{}'$.

Let me illustrate the intuition when $\|bool_i\|$ and $head(\mathcal{A}tom_i) = \texttt{assign}\ v = exp(*)$. In this case I have that the current atom is active, i.e. I need to schedule some events. The actual statement of this atom is `assign` , i.e. I need to schedule an event which will be triggered when there is a change in the variables $*$ and must update the value of v upon that change. Hence I schedule $(\sim (*), v = exp(*))$ into $E_i^{A\prime}$.

Similarly, $bool_i'$ gets $\sim (*)$ because the corresponding `assign` must be activated when there is a change in some of the variables $*$.

The other event guards are *false* which are never triggered and is reserved for the empty statement, $T = \|T + exp\|$ which stands for the time delay and $T = \|T\|$ when I want to schedule an event for the current time.

5.5 Healthiness Conditions

To increase the confidence in the operational semantics I will define and prove some healthiness conditions on it. Obviously, these healthiness conditions reflect my understanding of how a "correct" Verilog simulator should behave.

Let me give some definitions and simple properties about the operational semantics.

Definition 3 *A run of a Verilog program P is a sequence of configurations $\{c_j\}_{j=0}^{\infty}$ where c_0 is a start configuration derived from P and $c_{j+1} = Step(c_j)$ for $j \geq 0$.*

Lemma 1 *$head$ and $tail$ are total functions.*

Proof (Lemma 1) Both the functions are defined over $\mathcal{A}tom$ and have definitions for each sub-category of the domain. Some of the definitions are terminal like

$$head(\texttt{assign}\ \textbf{body}) = \texttt{assign}$$

and for those I know they are defined. I only need to check the totality of the recursive definitions. I will consider $head$ only though the same arguments apply to $tail$ as well.

The cases I need to look at are as follow:

$$head(\texttt{always}\ \ statement) = head(statement)$$

$$head(\texttt{initial}\ \ statement) = head(statement)$$

$$head(statement\ ;\mathcal{A}tom) = head(statement)$$

$$head(\texttt{if}\ \ (\texttt{bool})\ \ statement_1\ \texttt{else}\ \ statement_2) =$$

$$= \begin{cases} head(statement_1), & \text{iff } \|\texttt{bool}\ \| = \textit{true} \\ head(statement_2), & \text{otherwise} \end{cases}$$

$$head(\texttt{while}\ \ (\texttt{bool})\ \ statement) =$$

$$= \begin{cases} head(statement), & \text{iff } \|\texttt{bool}\ \| = \textit{true} \\ \varepsilon, & \text{otherwise} \end{cases}$$

$$head(\texttt{begin}\ \ statement_1\ ;\ \{statement\}\ \texttt{end}\ ) = head(statement_1)$$

The proof follows an induction by the complexity of the argument of $head$ with complexity here means number of characters in that argument, i.e. length of the argument. The base of this induction are the terminal cases of the definition where I already know that $head$ is defined. The induction step consists of trivial check that in all induction cases shown above, the argument of $head$ reduces its length. ■

Lemma 2 $\tau_0 \vee \tau_1 \vee \tau_2 \vee \tau_3 \equiv$ *true and* $\tau_i \wedge \tau_j \equiv$ *false for* $i \neq j$.

This lemma shows that my four step tests are orthogonal, i.e. at most one can be true at any time, and they capture all possible cases, i.e. at least one is true at any time.

Proof (Lemma 2) Let me have a configuration for which an variable interpretation $\|.\|$ and all τ tests are defined. I will show first that $\tau_0 \vee \tau_1 \vee \tau_2 \vee \tau_3 \equiv$ *true*.

$$\tau_0 \vee \tau_1 \vee \tau_2 \vee \tau_3 \equiv (\uparrow (\bigcup_{i,j} E_i^j) = \emptyset \wedge \bigvee_i \|bool_i\|) \vee$$

$$(\uparrow (\bigcup_{i,j} E_i^j) = \emptyset \wedge \bigwedge_i \neg\|bool_i\|) \vee$$

$$(\uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB}) \neq \emptyset) \vee (\uparrow (\bigcup_i E_i^A) \neq \emptyset) \equiv$$

$$\equiv [\uparrow (\bigcup_{i,j} E_i^j) = \emptyset \wedge (\bigvee_i \|bool_i\| \vee \bigwedge_i \neg\|bool_i\|)] \vee$$

$$(\uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB}) \neq \emptyset) \vee (\uparrow (\bigcup_i E_i^A) \neq \emptyset) \equiv$$

$$\equiv \uparrow (\bigcup_{i,j} E_i^j) = \emptyset \vee (\uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB}) \neq \emptyset) \vee (\uparrow (\bigcup_i E_i^A) \neq \emptyset)$$

Now it is vital to notice that $\uparrow (\bigcup_{i,j} E_i^j) = \emptyset \equiv \uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB})$ and then it is easier to extend the transformation as follows.

$$\tau_0 \vee \tau_1 \vee \tau_2 \vee \tau_3 \equiv (\uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB}) = \emptyset) \vee$$

$$(\uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB}) \neq \emptyset) \vee (\uparrow (\bigcup_i E_i^A) \neq \emptyset) \equiv$$

$$\equiv (\uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB}) = \emptyset) \vee$$

$$\uparrow (\bigcup_i E_i^{NB}) \neq \emptyset \vee \uparrow (\bigcup_i E_i^A) \neq \emptyset \equiv$$

$$\equiv (\uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB}) = \emptyset) \vee$$

$$\neg(\uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB}) = \emptyset) \equiv \textit{true}$$

Analogously I can show that all τ_j are mutually inconsistent. I will work out only $\tau_1 \wedge \tau_2$. All other conjunctions are treated in a very similar fashion.

$$\tau_1 \wedge \tau_2 \equiv (\uparrow (\bigcup_{i,j} E_i^j) = \emptyset \wedge \bigwedge_i \neg \|bool_i\|) \wedge$$

$$(\uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB}) \neq \emptyset) \equiv$$

$$\equiv \uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB}) = \emptyset \wedge \bigwedge_i \neg \|bool_i\| \wedge$$

$$\uparrow (\bigcup_i E_i^A) = \emptyset \wedge \uparrow (\bigcup_i E_i^{NB}) \neq \emptyset \equiv \textit{false}$$

and this completes the proof of lemma 2. ■

Lemma 3 *The function $Step(.)$ is total.*

The totality of $Step$ guarantees that a run is always achievable.

Proof (Lemma 3) The proof for the totality of $Step$ is based on the basic syntax of Verilog given in tables 4.1, and 4.2. As seen in the definitions of $\mathcal{A}tom_i'$, $bool_i'$, $(variable, value', new')$, $E_i^{A'}$ and $E_i^{NB'}$, the new configuration $Step(c)$ is well defined and depends only on the totality of the functions $head$ and $tail$ which we have already established. ■

Admittedly, there is no established formal semantics of Verilog HDL, i.e. the formal proof of correctness for a simulator and/or semantics for the language cannot be derived because I do not have a standard to compare our approach with. My only hope is to show that the formal model behind the semantics is a true reflection of my intuition for the behaviour which a Verilog program would generate.

Healthiness Condition 1 *If $\{c_j\}_{j=0}^{\infty}$ is a run generated by program P, then $T_{c_0} = 0$ and $T_{c_{j+1}} \geq T_{c_j}$ for all $j \geq 0$.*

The intuition behind this healthiness condition is that time in our operational semantics is monotonically increasing.

Proof (Healthiness Condition 1) I should note first that any run begins with a start configuration where $T = 0$. From there after, in all cases of the definition of $Step$ I either keep T or increment its value with one. ■

Healthiness Condition 2 *If $c' = Step(c)$ for a configuration c, then all enabled active events from c are executed during the transition step.*

This healthiness condition guarantees the true parallelism of the proposed operational semantics.

Proof (Healthiness Condition 2) The proof follows the construction of V' and $E_i^{A\prime}$. In the first two step tests,i.e. τ_1 and τ_2, I have that all active events are disabled, so $V' = V$ and $E_i^{A\prime} = E_i^A$. In the case of τ_3 though, I have that the formation of $E_i^{A\prime}$ consists of the union of all disabled active events from c, i.e. $\downarrow (E_i^A)$, and the new scheduled events from the atoms. In the same time, the new values of the variables in V' are calculated according to the enabled active events. Thus all active enabled events from c are executed in the transition from c to c'. ■

Definition 4 *If $\uparrow (\bigcup_{i,j} E_i^j) = \emptyset$ for a configuration, then it is a stable configuration. A start configuration is not stable.*

Healthiness Condition 3 *Time advances at a stable configuration only.*

This health condition guarantees that unless a stable configuration is reached, the time does not advance. This property ensures that the stimulus and the response are synchronised.

Before looking at condition 3, I will prove a small lemma.

Lemma 4 *Let* $\{c_j\}_{j=0}^{\infty}$ *be an arbitrary run and*

$$c = (T, ((\mathcal{A}tom_1, bool_1), \ldots, (\mathcal{A}tom_n, bool_n)), V, (E_1, \ldots, E_n))$$

be any of c_j *for* $j > 0$*. Then for all* i*, there exists* $e \in E_i^A$ *where* $e = (bool_i, statement)$*.*

Proof (Lemma 4) The proof follows induction for j and the base for the induction is $j = 1$. Obviously τ_0 is true for c_0, i.e. for the start configuration, $bool_i$ is true and $\downarrow (E_i^A) = \emptyset$ for all i. Following the definition of $Step$ I can see what $bool_i'$ and $E_i^{A'}$ will be, i.e. I can check if the claim from the lemma holds for c_1 which is the base for the induction.

Let me fix i, i.e. I fix the atom in the configuration. For all possible cases I see that $(bool_i', \alpha) \in E_i^{A'}$ for some α which shows that the base for the induction holds for c_1.

Let me now assume that the claim holds for c_n. I would need to show that it holds for c_{n+1}. Here I need to consider two cases.

- If $\|bool_i\| =$ *true*, then I simply see that the cases for $bool_i'$ and $E_i^{A'}$ correspond and for each of them I have $(bool_i', \alpha) \in E_i^{A'}$ for some α.

- I know that the claim from the lemma holds for c_n, i.e. if $\|bool_i\| =$ *false*, then the corresponding event $(bool_i, \alpha) \in E_i^A$ would not be enabled, i.e. $(bool_i, \alpha) \in \downarrow (E_i^A)$. Obviously, $\downarrow (E_i^A) \subseteq E_i^{A'}$, i.e. $(bool_i, \alpha) \in E_i^{A'}$. On the other hand, $\|bool_i\| =$ *false* implies $bool_i' = bool_i$.

This shows that the claim from the lemma holds for c_{n+1}. ■

Lemma 4 implies that, except for the start configuration, $\uparrow (\bigcup_{i,j} E_i^j) = \emptyset \Rightarrow \bigwedge_i \neg \| bool_i \|$, $\tau_1 = \uparrow (\bigcup_{i,j} E_i^j) = \emptyset$ and τ_0 is true in the start configuration only. Now I can prove healthiness condition 3.

Proof (Healthiness Condition 3) It is clear from the definition of $Step$, that time T advances only when $\tau_1 =$ *true*. From lemma 4 I derived that $\tau_1 \Leftrightarrow \uparrow (\bigcup_{i,j} E_i^j) = \emptyset$ which by definition is only true in the stable states. ■

I can interpret healthiness condition 3 in the following way. If the system is in a stable configuration and there is an event going off in its successor, i.e. there is a stimulus from the environment of the system, then it would freeze the time until a new stable state is reached. Obviously, a new stable state reached means that the system has produced a response to the stimulus and thus has stabilised itself. The fact that the response and that stimulus occur simultaneously simply says that Verilog is a *synchronous* language [73, 28, 54].

5.6 Example of a Simulation

As a simple example I will give the sequence of configurations, i.e. a run, for the following Verilog program.

```
module test ;

    reg a0, a1, b0, b1 ;

    wire cout, r0, r1, r2 ;
```

```
    assign  #1  {cout, r0} = a0 + b0 ;
    assign  #1  {r2, r1} = a1 + b1 + cout ;
    initial  begin
        a0 = 0 ;  a1 = 0 ;
        b0 = 0 ;  b1 = 0 ;
        #10  a0 = 1 ;  b0 = 1 ;
    end
endmodule
```

The example is purely illustrative and could be viewed as simulation of the program. The obvious atoms are

- $\texttt{assign}_1$ where

  $\texttt{assign}_1$ `is assign` $\#1$ $\{\texttt{cout}, \texttt{r0}\} = \texttt{a0} + \texttt{b0}$;

- $\texttt{assign}_2$ where

  $\texttt{assign}_2$ `is assign` $\#1$ $\{\texttt{r2}, \texttt{r1}\} = \texttt{a1} + \texttt{b1} + \texttt{cout}$;

- `initial` where

  ```
  initial  is  initial  begin
          a0 = 0 ;  a1 = 0 ;
          b0 = 0 ;  b1 = 0 ;
          #10  a0 = 1 ;  b0 = 1 ;
  ```

```
end
```

therefore the start configuration will be

$$c_0 = (0, ((\texttt{assign}_1\ , \textit{true}), (\texttt{assign}_2\ , \textit{true}), (\texttt{initial}\ , \textit{true})), \{\ \bot\ \}, (\emptyset^2, \emptyset^2, \emptyset^2)).$$

At this point I evaluate the τ step tests. Obviously only $\tau_0 = \textit{true}$, so the next step will be

$$\begin{array}{l} c_1 = (0, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)), \\ \qquad ((a1 = 0; b0 = 0; b1 = 0; \#10\ \ a0 = 1; b0 = 1), T = 0), \\ \quad \{\ \bot\ \}, \\ \quad (\ \ (\{(\sim (a0, b0), t_1 = T + 1), (T = t_1, \{cout, r0\} = a0 + b0)\}, \emptyset)\ , \\ \qquad (\{(\sim (a1, b1, cout), t_2 = T{+}1), (T = t_2, \{r2, r1\} = a1{+}b1{+}cout)\}, \emptyset)\ , \\ \qquad (\{(T = 0, a0 = 0)\}, \emptyset) \\ \quad) \\ ) \end{array}$$

Now I need to evaluate the τ step tests again which gives me $\tau_3 = \textit{true}$ on the basis that the event $(T = 0, a0 = 0)$ from the event list associated with the `initial` atom is enabled. Thus my next configuration would be

$$\begin{array}{l} c_2 = (0, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)), \\ \qquad ((b0 = 0; b1 = 0; \#10\ \ a0 = 1; b0 = 1), T = 0)), \\ \quad \{(a0, 0, \textit{true}),\ \bot\ \}, \end{array}$$

$$
\begin{array}{l}
\quad (\ (\{(\sim (a0, b0), t_1 = T + 1), (T = t_1, \{cout, r0\} = a0 + b0)\}, \emptyset)\ , \\
\qquad (\{(\sim (a1, b1, cout), t_2 = T{+}1), (T = t_2, \{r2, r1\} = a1{+}b1{+}cout)\}, \emptyset)\ , \\
\qquad (\{(T = 0, a1 = 0)\}, \emptyset) \\
\quad) \\
)
\end{array}
$$

Again I have the case when $\tau_3 =$ *true*. Hence

$$
\begin{array}{l}
c_3 = (0, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)), \\
\qquad\qquad ((b1 = 0; \#10\ \ a0 = 1; b0 = 1), T = 0)), \\
\qquad \{(a0, 0, \textit{false}), (a1, 0, \textit{true}), (t_1, 1, \textit{true}),\ \bot\ \}, \\
\quad (\ (\{(\sim (a0, b0), t_1 = T + 1), (T = t_1, \{cout, r0\} = a0 + b0)\}, \emptyset)\ , \\
\qquad (\{(\sim (a1, b1, cout), t_2 = T{+}1), (T = t_2, \{r2, r1\} = a1{+}b1{+}cout)\}, \emptyset)\ , \\
\qquad (\{(T = 0, b0 = 0)\}, \emptyset) \\
\quad) \\
)
\end{array}
$$

and similarly I can generate the sequence

$$
\begin{array}{l}
c_4 = (0, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)), \\
\qquad\qquad ((\#10\ \ a0 = 1; b0 = 1), T = 0)), \\
\qquad \{(a0, 0, \textit{false}), (a1, 0, \textit{false}), (b0, 0, \textit{true}), (t_1, 1, \textit{false}), (t_2, 1, \textit{true}),\ \bot\ \}, \\
\quad (\ (\{(\sim (a0, b0), t_1 = T + 1), (T = t_1, \{cout, r0\} = a0 + b0)\}, \emptyset)\ , \\
\qquad (\{(\sim (a1, b1, cout), t_2 = T{+}1), (T = t_2, \{r2, r1\} = a1{+}b1{+}cout)\}, \emptyset)\ ,
\end{array}
$$

$$(\{(T = 0, b1 = 0)\}, \emptyset)$$

)

)

$$c_5 = (0, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$$

$$((a0 = 1; b0 = 1), T = 10)),$$

$$\{(a0, 0, \textit{false}), (a1, 0, \textit{false}), (b0, 0, \textit{false}), (b1, 0, \textit{true}), (t_1, 1, \textit{false}),$$

$$(t_2, 1, \textit{false}),\ \perp\ \},$$

$$(\ (\{(\sim (a0, b0), t_1 = T + 1), (T = t_1, \{cout, r0\} = a0 + b0)\}, \emptyset)\ ,$$

$$(\{(\sim (a1, b1, cout), t_2 = T+1), (T = t_2, \{r2, r1\} = a1+b1+cout)\}, \emptyset)\ ,$$

$$(\{(T = 10, \varepsilon)\}, \emptyset)$$

)

)

$$c_6 = (0, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$$

$$((a0 = 1; b0 = 1), T = 10)),$$

$$\{(a0, 0, \textit{false}), (a1, 0, \textit{false}), (b0, 0, \textit{false}), (b1, 0, \textit{false}), (t_1, 1, \textit{false}),$$

$$(t_2, 1, \textit{false}),\ \perp\ \},$$

$$(\ (\{(\sim (a0, b0), t_1 = T + 1), (T = t_1, \{cout, r0\} = a0 + b0)\}, \emptyset)\ ,$$

$$(\{(\sim (a1, b1, cout), t_2 = T+1), (T = t_2, \{r2, r1\} = a1+b1+cout)\}, \emptyset)\ ,$$

$$(\{(T = 10, \varepsilon)\}, \emptyset)$$

)

)

Here I have all the events are not enabled, i.e. $\tau_1 = \textit{true}$ and I take a time advancing step.

$$c_7 = (1, ((\texttt{assign}_1 , \sim (a0, b0)), (\texttt{assign}_2 , \sim (a1, b1, cout)),$$
$$((a0 = 1; b0 = 1), T = 10)),$$
$$\{(a0, 0, \textit{false}), (a1, 0, \textit{false}), (b0, 0, \textit{false}), (b1, 0, \textit{false}), (t_1, 1, \textit{false}),$$
$$(t_2, 1, \textit{false}), \perp \},$$
$$(\ (\{(\sim (a0, b0), t_1 = T + 1), (T = t_1, \{cout, r0\} = a0 + b0)\}, \emptyset),$$
$$(\{(\sim (a1, b1, cout), t_2 = T+1), (T = t_2, \{r2, r1\} = a1+b1+cout)\}, \emptyset),$$
$$(\{(T = 10, \varepsilon)\}, \emptyset)$$
$$)$$
$$)$$

This leads me to a configuration where $\tau_3 = \textit{true}$.

$$c_8 = (1, ((\texttt{assign}_1 , \sim (a0, b0)), (\texttt{assign}_2 , \sim (a1, b1, cout)),$$
$$((a0 = 1; b0 = 1), T = 10)),$$
$$\{(a0, 0, \textit{false}), (a1, 0, \textit{false}), (b0, 0, \textit{false}), (b1, 0, \textit{false}), (t_1, 1, \textit{false}),$$
$$(t_2, 1, \textit{false}), (r0, 0, \textit{true}), (cout, 0, \textit{true}), \perp \},$$
$$(\ (\{(\sim (a0, b0), t_1 = T + 1)\}, \emptyset),$$
$$(\{(\sim (a1, b1, cout), t_2 = T + 1)\}, \emptyset),$$
$$(\{(T = 10, \varepsilon)\}, \emptyset)$$
$$)$$

)

$$c_9 = (1, ((\texttt{assign}_1 , \sim (a0, b0)), (\texttt{assign}_2 , \sim (a1, b1, cout)),$$

$$((a0 = 1; b0 = 1), T = 10)),$$

$$\{(a0, 0, \textit{false}), (a1, 0, \textit{false}), (b0, 0, \textit{false}), (b1, 0, \textit{false}), (t_1, 1, \textit{false}),$$

$$(t_2, 2, \textit{true}), (r0, 0, \textit{false}), (cout, 0, \textit{false}), \perp \},$$

$$(\ (\{(\sim (a0, b0), t_1 = T + 1)\}, \emptyset),$$

$$(\{(\sim (a1, b1, cout), t_2 = T+1), (T = t_2, \{r2, r1\} = a1+b1+cout)\}, \emptyset),$$

$$(\{(T = 10, \varepsilon)\}, \emptyset)$$

)

)

Again I advance the time

$$c_{10} = (2, ((\texttt{assign}_1 , \sim (a0, b0)), (\texttt{assign}_2 , \sim (a1, b1, cout)),$$

$$((a0 = 1; b0 = 1), T = 10)),$$

$$\{(a0, 0, \textit{false}), (a1, 0, \textit{false}), (b0, 0, \textit{false}), (b1, 0, \textit{false}), (t_1, 1, \textit{false}),$$

$$(t_2, 2, \textit{false}), (r0, 0, \textit{false}), (cout, 0, \textit{false}), \perp \},$$

$$(\ (\{(\sim (a0, b0), t_1 = T + 1)\}, \emptyset),$$

$$(\{(\sim (a1, b1, cout), t_2 = T+1), (T = t_2, \{r2, r1\} = a1+b1+cout)\}, \emptyset),$$

$$(\{(T = 10, \varepsilon)\}, \emptyset)$$

)

)

$$c_{11} = (2, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$$
$$((a0 = 1; b0 = 1), T = 10)),$$
$$\{(a0, 0, \textit{false}), (a1, 0, \textit{false}), (b0, 0, \textit{false}), (b1, 0, \textit{false}), (t_1, 1, \textit{false}),$$
$$(t_2, 2, \textit{false}), (r0, 0, \textit{false}), (r1, 0, \textit{true}), (r2, 0, \textit{true}), (cout, 0, \textit{false})\},$$
$$(\ (\{(\sim (a0, b0), t_1 = T + 1)\}, \emptyset)\ ,$$
$$(\{(\sim (a1, b1, cout), t_2 = T + 1)\}, \emptyset)\ ,$$
$$(\{(T = 10, \varepsilon)\}, \emptyset)$$
$$)$$
$$)$$

and I have to advance the time again. At this point nothing interesting happens until I get $T = 10$, so I skip to c_{19} where

$$c_{19} = (10, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$$
$$((a0 = 1; b0 = 1), T = 10)),$$
$$\{(a0, 0, \textit{false}), (a1, 0, \textit{false}), (b0, 0, \textit{false}), (b1, 0, \textit{false}), (t_1, 1, \textit{false}),$$
$$(t_2, 2, \textit{false}), (r0, 0, \textit{false}), (r1, 0, \textit{false}), (r2, 0, \textit{false}), (cout, 0, \textit{false})\},$$
$$(\ (\{(\sim (a0, b0), t_1 = T + 1)\}, \emptyset)\ ,$$
$$(\{(\sim (a1, b1, cout), t_2 = T + 1)\}, \emptyset)\ ,$$
$$(\{(T = 10, \varepsilon)\}, \emptyset)$$
$$)$$
$$)$$

$$c_{20} = (10, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$$

$$((b0 = 1), T = 10)),$$

$$\{(a0, 0, \textit{false}), (a1, 0, \textit{false}), (b0, 0, \textit{false}), (b1, 0, \textit{false}), (t_1, 1, \textit{false}),$$

$$(t_2, 2, \textit{false}), (r0, 0, \textit{false}), (r1, 0, \textit{false}), (r2, 0, \textit{false}), (cout, 0, \textit{false})\},$$

$$(\ (\{(\sim (a0, b0), t_1 = T + 1)\}, \emptyset)\ ,$$

$$(\{(\sim (a1, b1, cout), t_2 = T + 1)\}, \emptyset)\ ,$$

$$(\{(T = 10, a0 = 1)\}, \emptyset)$$

$$)$$

$$)$$

$$c_{21} = (10, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$$

$$(\varepsilon, T = 10)),$$

$$\{(a0, 1, \textit{true}), (a1, 0, \textit{false}), (b0, 0, \textit{false}), (b1, 0, \textit{false}), (t_1, 1, \textit{false}),$$

$$(t_2, 2, \textit{false}), (r0, 0, \textit{false}), (r1, 0, \textit{false}), (r2, 0, \textit{false}), (cout, 0, \textit{false})\},$$

$$(\ (\{(\sim (a0, b0), t_1 = T + 1)\}, \emptyset)\ ,$$

$$(\{(\sim (a1, b1, cout), t_2 = T + 1)\}, \emptyset)\ ,$$

$$(\{(T = 10, b0 = 1)\}, \emptyset)$$

$$)$$

$$)$$

$$c_{22} = (10, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$$

$$(\varepsilon, \textit{false})),$$

$\{(a0, 1, \textit{false}), (a1, 0, \textit{false}), (b0, 1, \textit{true}), (b1, 0, \textit{false}), (t_1, 11, \textit{true}),$

$(t_2, 2, \textit{false}), (r0, 0, \textit{false}), (r1, 0, \textit{false}), (r2, 0, \textit{false}), (cout, 0, \textit{false})\},$

$($ $(\{(\sim (a0, b0), t_1 = T + 1), (T = t_1, \{cout, r0\} = a0 + b0)\}, \emptyset)$,

$(\{(\sim (a1, b1, cout), t_2 = T + 1)\}, \emptyset)$,

$(\{(\textit{false}, \varepsilon)\}, \emptyset)$

$)$

$)$

$c_{23} = (10, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$

$(\varepsilon, \textit{false})),$

$\{(a0, 1, \textit{false}), (a1, 0, \textit{false}), (b0, 1, \textit{false}), (b1, 0, \textit{false}), (t_1, 11, \textit{false}),$

$(t_2, 2, \textit{false}), (r0, 0, \textit{false}), (r1, 0, \textit{false}), (r2, 0, \textit{false}), (cout, 0, \textit{false})\},$

$($ $(\{(\sim (a0, b0), t_1 = T + 1), (T = t_1, \{cout, r0\} = a0 + b0)\}, \emptyset)$,

$(\{(\sim (a1, b1, cout), t_2 = T + 1)\}, \emptyset)$,

$(\{(\textit{false}, \varepsilon)\}, \emptyset)$

$)$

$)$

$c_{24} = (11, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$

$(\varepsilon, \textit{false})),$

$\{(a0, 1, \textit{false}), (a1, 0, \textit{false}), (b0, 1, \textit{false}), (b1, 0, \textit{false}), (t_1, 11, \textit{false}),$

$(t_2, 2, \textit{false}), (r0, 0, \textit{false}), (r1, 0, \textit{false}), (r2, 0, \textit{false}), (cout, 0, \textit{false})\},$

$$
\begin{array}{l}
\quad (\ (\{(\sim(a0,b0), t_1 = T+1), (T = t_1, \{cout, r0\} = a0 + b0)\}, \emptyset)\,, \\
\qquad (\{(\sim(a1,b1,cout), t_2 = T+1)\}, \emptyset)\,, \\
\qquad (\{(\mathit{false}, \varepsilon)\}, \emptyset) \\
\quad) \\
)
\end{array}
$$

$$
\begin{array}{l}
c_{25} = (11, ((\texttt{assign}_1\ , \sim(a0,b0)), (\texttt{assign}_2\ , \sim(a1,b1,cout)), \\
\qquad (\varepsilon, \mathit{false})), \\
\quad \{(a0,1,\mathit{false}), (a1,0,\mathit{false}), (b0,1,\mathit{false}), (b1,0,\mathit{false}), (t_1,11,\mathit{false}), \\
(t_2,2,\mathit{false}), (r0,0,\mathit{false}), (r1,0,\mathit{false}), (r2,0,\mathit{false}), (cout,1,\mathit{true})\}, \\
\quad (\ (\{(\sim(a0,b0), t_1 = T+1)\}, \emptyset)\,, \\
\qquad (\{(\sim(a1,b1,cout), t_2 = T+1)\}, \emptyset)\,, \\
\qquad (\{(\mathit{false}, \varepsilon)\}, \emptyset) \\
\quad) \\
)
\end{array}
$$

$$
\begin{array}{l}
c_{26} = (11, ((\texttt{assign}_1\ , \sim(a0,b0)), (\texttt{assign}_2\ , \sim(a1,b1,cout)), \\
\qquad (\varepsilon, \mathit{false})), \\
\quad \{(a0,1,\mathit{false}), (a1,0,\mathit{false}), (b0,1,\mathit{false}), (b1,0,\mathit{false}), (t_1,11,\mathit{false}), \\
(t_2,12,\mathit{true}), (r0,0,\mathit{false}), (r1,0,\mathit{false}), (r2,0,\mathit{false}), (cout,1,\mathit{false})\}, \\
\quad (\ (\{(\sim(a0,b0), t_1 = T+1)\}, \emptyset)\,, \\
\qquad (\{(\sim(a1,b1,cout), t_2 = T+1), (T = t_2, \{r2, r1\} = a1+b1+cout)\}, \emptyset)\,,
\end{array}
$$

$(\{(\mathit{false}, \varepsilon)\}, \emptyset)$

)

)

$c_{27} = (12, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$

$(\varepsilon, \mathit{false})),$

$\{(a0, 1, \mathit{false}), (a1, 0, \mathit{false}), (b0, 1, \mathit{false}), (b1, 0, \mathit{false}), (t_1, 11, \mathit{false}),$

$(t_2, 12, \mathit{false}), (r0, 0, \mathit{false}), (r1, 0, \mathit{false}), (r2, 0, \mathit{false}), (cout, 1, \mathit{false})\},$

($(\{(\sim (a0, b0), t_1 = T + 1)\}, \emptyset)$,

$(\{(\sim (a1, b1, cout), t_2 = T+1), (T = t_2, \{r2, r1\} = a1+b1+cout)\}, \emptyset)$,

$(\{(\mathit{false}, \varepsilon)\}, \emptyset)$

)

)

$c_{28} = (12, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$

$(\varepsilon, \mathit{false})),$

$\{(a0, 1, \mathit{false}), (a1, 0, \mathit{false}), (b0, 1, \mathit{false}), (b1, 0, \mathit{false}), (t_1, 11, \mathit{false}),$

$(t_2, 12, \mathit{false}), (r0, 0, \mathit{false}), (r1, 1, \mathit{true}), (r2, 0, \mathit{false}), (cout, 1, \mathit{false})\},$

($(\{(\sim (a0, b0), t_1 = T + 1)\}, \emptyset)$,

$(\{(\sim (a1, b1, cout), t_2 = T + 1)\}, \emptyset)$,

$(\{(\mathit{false}, \varepsilon)\}, \emptyset)$

)

)

$$c_{29} = (13, ((\texttt{assign}_1\ , \sim (a0, b0)), (\texttt{assign}_2\ , \sim (a1, b1, cout)),$$

$$(\varepsilon, \textit{false})),$$

$$\{(a0, 1, \textit{false}), (a1, 0, \textit{false}), (b0, 1, \textit{false}), (b1, 0, \textit{false}), (t_1, 11, \textit{false}),$$

$$(t_2, 12, \textit{false}), (r0, 0, \textit{false}), (r1, 1, \textit{false}), (r2, 0, \textit{false}), (cout, 1, \textit{false})\},$$

$$(\ (\{(\sim (a0, b0), t_1 = T + 1)\}, \emptyset)\ ,$$

$$(\{(\sim (a1, b1, cout), t_2 = T + 1)\}, \emptyset)\ ,$$

$$(\{(\textit{false}, \varepsilon)\}, \emptyset)$$

)

)

From here after, only τ_0 is true and the configuration keeps stuttering. I can easily track the values of r_2, r_1 and r_0 where the result is stored.

5.7 Summary

An operational semantics for Verilog is presented here. It is fully parallel and it treats the language with its Behavioural and RTL constructs as defined in tables 4.1 and 4.2.

Several healthiness conditions support the trustworthiness of my semantics. I resort to them because Verilog lacks an established formal semantics to which I can relate mine. However, I believe that the healthiness conditions truly reflect my intuition about how a simulator for Verilog should behave.

At the end I present a small example of a run. This shows the applicability of my work in a possible real simulator.

Chapter 6

Equivalence of Denotational and Operational Semantics

The proof of equivalence between the operational and the denotational semantics of Verilog is presented. This guarantees my ability to calculate properties in an effective way by constructing a machine.

6.1 Introduction

In the previous two chapters 4 and 5 I presented two different formalisms that define the semantics of Verilog in denotational and operational terms correspondingly. However, the possibility of differences in these formalisms still remains and hence the need for a formal proof of their equivalence. Here I present the details of the proof which also has the added benefit of strengthening my belief that the semantics I present is "correct". Admittedly,

Verilog does not have an established formal semantics and I cannot check if my work is correct with respect to a "standard" semantics. Therefore, in chapter 5 I gave several healthiness conditions which I believe should serve as necessary tests for every formalism that claims to give semantics for Verilog, being a simulator or otherwise.

In this chapter I go a step further. Since I now have two formalisms I can attempt to show that they are equivalent in the sense that the behaviour described by the denotational semantics is precisely the one generated by the operational semantics. Definition 6 and Theorem 4 formalise this notion. Once I have this proof, I can claim a higher confidence in the truthfulness and correctness of my work.

6.2 Outline of the Proof

The general idea of the proof is to construct a step for the denotational semantics and, at the same time, to construct a state and an ITL formula from an operational configuration. Having the latter I can define when an ITL state and a formula are equivalent to a configuration and having the former I can start with equivalent states for the operational and the denotational semantics and take a step after which I can again compare the states.

For the execution of this plan I will need several lemmas, theorems and a definition. Lemma 5 allows me to combine the steps of all parallel atoms into a single step for the whole system. Lemma 6 allows me to find the step of the clock in the denotational semantics. Theorem 2 constructs a step for the denotational semantics, while definition 5

in conjunction with theorem 3 tells me that the set of $\mathcal{A}tom$s from a syntactically correct Verilog program is a bit smaller that the set given by the general definition of $\mathcal{A}tom$ in table 4.2. The result of Theorem 3 is ultimately used in defining the equivalence between an ITL formula and state and an operational configuration (definition 6) which trivially leads to the final result of Theorem 4.

6.3 Detailed Proof

Lemma 5

$$
\begin{aligned}
&(f_j \ \supset \mathsf{skip}) \quad \supset \\
&\qquad \exists x \bullet (f_1(x)\,;F_1(x)) \wedge \ldots \wedge \exists x \bullet (f_n(x)\,;F_n(x)) \equiv \\
&\qquad \exists x_1, \ldots, x_n \bullet \{ \\
&\qquad\qquad (f_1[x/x_1] \wedge \ldots \wedge f_n[x/x_n]); \\
&\qquad\qquad (F_1[x/x_1] \wedge \ldots \wedge F_n[x/x_n]) \\
&\qquad \}
\end{aligned}
$$

Lemma 6

$$
\mathit{inf} \quad \supset \quad ((\mathsf{while}\ b\ \mathsf{do}\ f_1)\,;f_2)^* \equiv (\mathsf{if}\ b\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2)^*
$$

Proof (Lemma 6)

$$\begin{aligned} lhs &\equiv ((\textsf{while } b \textsf{ do } f_1)\,;f_2)^* \\ &\equiv (((b \wedge f_1)^* \wedge \textit{fin}\ \neg b)\,;f_2)^* \\ &\equiv ((b \wedge f_1)^*\,;(\neg b \wedge f_2))^* \\ rhs &\equiv (\textsf{if } b \textsf{ then } f_1 \textsf{ else } f_2)^* \\ &\equiv ((b \wedge f_1) \vee (\neg b \wedge f_2))^* \end{aligned}$$

I will consider several cases. Let $f_1 \supset \textsf{empty}$. Then $(b \wedge f_1)^* \equiv \textsf{empty}$, therefore $lhs \equiv (\neg b \wedge f_2)^*$. At the same time $rhs \equiv (\neg b \wedge f_2)^*$ and therefore $lhs \equiv rhs$.

If $f_2 \supset \textsf{empty}$, then $lhs \equiv ((b \wedge f_1)^*\,;(\neg b \wedge \textsf{empty}))^*$, i.e. $\Diamond\neg b \supset (lhs \supset \textit{finite})$ and $\textit{inf} \wedge lhs \equiv \textit{false}$. Similarly, $\Diamond\neg b \supset (rhs \supset \textit{finite})$, i.e. $\textit{inf} \wedge rhs \equiv \textit{false}$, therefore $\Diamond\neg b \wedge \textit{inf} \supset (lhs \equiv rhs)$. However, $\Box b \supset lhs \equiv (b \wedge f_1)^* \wedge \textit{inf} \equiv rhs \wedge \textit{inf}$, thus $(f_2 \supset \textsf{empty}) \supset \inf \supset (lhs \equiv rhs)$.

Now, let me consider the case when $f_1 \wedge f_2 \wedge \textit{more}$. In this case

$$\begin{aligned} b \wedge \textit{inf} &\supset ((\textsf{while } b \textsf{ do } f_1)\,;f_2)^* \\ &\equiv ((\textsf{while } b \textsf{ do } f_1)\,;f_2)\,;((\textsf{while } b \textsf{ do } f_1)\,;f_2)^* \\ &\equiv (\textsf{if } b \textsf{ then } (f_1\,;(\textsf{while } b \textsf{ do } f_1)) \textsf{ else empty})\,;f_2)\,;((\textsf{while } b \textsf{ do } f_1)\,;f_2)^* \\ &\equiv f_1\,;(\textsf{while } b \textsf{ do } f_1)\,;f_2\,;((\textsf{while } b \textsf{ do } f_1)\,;f_2)^* \end{aligned}$$

$$\equiv \quad f_1 \,;\, ((\textsf{while } b \textsf{ do } f_1) \,;\, f_2)^*$$

and

$$\begin{aligned} b \wedge \mathit{inf} \quad &\supset \quad (\textsf{if } b \textsf{ then } f_1 \textsf{ else } f_2)^* \\ &\equiv \quad (\textsf{if } b \textsf{ then } f_1 \textsf{ else } f_2) \,;\, (\textsf{if } b \textsf{ then } f_1 \textsf{ else } f_2)^* \\ &\equiv \quad f_1 \,;\, (\textsf{if } b \textsf{ then } f_1 \textsf{ else } f_2)^* \end{aligned}$$

and

$$\begin{aligned} \neg b \wedge \mathit{inf} \quad &\supset \quad ((\textsf{while } b \textsf{ do } f_1) \,;\, f_2)^* \\ &\equiv \quad ((\textsf{while } b \textsf{ do } f_1) \,;\, f_2) \,;\, ((\textsf{while } b \textsf{ do } f_1) \,;\, f_2)^* \\ &\equiv \quad (\textsf{if } b \textsf{ then } (f_1 \,;\, (\textsf{while } b \textsf{ do } f_1)) \textsf{ else empty}) \,;\, f_2) \,;\, ((\textsf{while } b \textsf{ do } f_1) \,;\, f_2)^* \\ &\equiv \quad f_2 \,;\, ((\textsf{while } b \textsf{ do } f_1) \,;\, f_2)^* \end{aligned}$$

and

$$\begin{aligned} \neg b \wedge \mathit{inf} \quad &\supset \quad (\textsf{if } b \textsf{ then } f_1 \textsf{ else } f_2)^* \\ &\equiv \quad (\textsf{if } b \textsf{ then } f_1 \textsf{ else } f_2) \,;\, (\textsf{if } b \textsf{ then } f_1 \textsf{ else } f_2)^* \\ &\equiv \quad f_2 \,;\, (\textsf{if } b \textsf{ then } f_1 \textsf{ else } f_2)^* \end{aligned}$$

This tells me that

$$lhs \wedge \mathit{inf} \equiv ((b \wedge f_1) \vee (\neg b \wedge f_2)) \,;\, lhs \wedge \mathit{inf}$$

and

$$rhs \wedge \mathit{inf} \equiv ((b \wedge f_1) \vee (\neg b \wedge f_2)) \,;\, rhs \wedge \mathit{inf}.$$

Therefore $lhs \wedge \mathit{inf} \equiv rhs \wedge \mathit{inf}$, i.e.

$$\mathit{inf} \quad \supset \quad lhs \equiv rhs$$

■

Theorem 2 *Let* `P` *be a Verilog program.* `P` *'s denotational semantics* $\|$`P`$\|$ *has a step.*

Proof (Theorem 2) Let me say that `P` is a program with n atoms and

`P` ::= `module` P $(*)$;

$global\ variables$;

$atom_1$;

$atom_2$;

...

$atom_n$;

`endmodule`

I will assume that the global variables that do not appear on the left-hand side of a non-

blocking assignment are ranging among $\mathtt{v_i}$ while those that appear on the lefthand side of non-blocking assignments are ranging in $\mathtt{x_i}$. Let the following be the initial values of the variables in P :

- $\mathcal{T}ime = \hat{t}$, for some integer value $\hat{t}$, i.e. I consider the case when time starts from $\hat{t}$.
- For any global variable v I will assume an initial value $\hat{v}$ and associated event trigger new_v. For that its corresponding memory variable $\mathcal{V}$, I will have that $\mathcal{V} = \hat{v}$ and $new_v = new_{\mathcal{V}}$,
- For some "non-blocking" variables $\mathtt{x_k}$ I will assume that some events have been scheduled for them. Let me say that $t_k \geq \hat{t}$ is the time for the non-blocking assignment and $\hat{x}_k$ is the scheduled value. Then I will have $[t_k, \hat{x}_k]$ in the list $\mathcal{L}^{\mathcal{X}_k}_{atom_j}$, if a non-blocking assignment for $\mathtt{x_k}$ occurs in the code for $atom_j$.

I also know that the denotational semantics for P is as follows.

$$
\begin{array}{l}
\| \mathtt{P} \| ::= \\
\quad \exists Atom_1.active, \ldots, Atom_n.active, \\
\quad\quad \mathcal{V}_i, \mathcal{X}_k, Disable, \mathcal{T}ime, \mathcal{L}^{\mathcal{X}_k}_{atom_j} \bullet \\
\quad\quad\quad \{ \\
\quad\quad\quad\quad \mathcal{V}_i = \hat{v}_i \wedge new_{\mathcal{V}_i} = new_{v_i} \wedge \mathcal{X}_j = \hat{x}_j \wedge new_{\mathcal{X}_j} = new_{x_j} \wedge
\end{array}
$$

$$\mathcal{L}^{\mathcal{X}_k}_{atom_j} = [\ldots,[t_k,\hat{x}_k],\ldots] \wedge \mathcal{T}ime = \hat{t} \wedge$$

$$\Box(Disable = (Atom_1.active \vee \ldots \vee Atom_n.active)) \wedge$$

$$\mathit{inf} \wedge clock(Disable) \wedge \|atom_1\| \wedge \|atom_2\| \wedge \ldots \wedge \|atom_n\| \wedge$$

$$NB^{x_1}_{atom_j} \wedge \ldots \wedge NB^{x_m}_{atom_k}$$

$$\}$$

where $\mathcal{V}_i$ are the global variables, $\mathcal{X}_k$ are the variables appearing on the left-hand side of a non-blocking assignment and $\mathcal{L}^{\mathcal{X}_k}_{atom_j}$ are the list variables for non-blocking assignments for each atom and each non-blocking variable.

$\|\mathtt{P}\|$ is transformed into a form where I can apply lemma 5, i.e.

$$\|\mathtt{P}\| \quad \equiv$$

$$\exists Atom_1.active,\ldots,Atom_n.active,$$

$$\mathcal{V}_i, \mathcal{X}_k, Disable, \mathcal{T}ime, \mathcal{L}^{\mathcal{X}_k}_{atom_j} \bullet$$

$$\{$$

$$\mathcal{V}_i = \hat{v}_i \wedge new_{\mathcal{V}_i} = new_v \wedge \mathcal{X}_j = \hat{x}_j \wedge new_{\mathcal{X}_j} = new_{x_j} \wedge$$

$$\mathcal{L}^{\mathcal{X}_k}_{atom_j} = [\ldots,[t_k,\hat{x}_k],\ldots] \wedge \mathcal{T}ime = \hat{t} \wedge$$

$$\Box(Disable = (Atom_1.active \vee \ldots \vee Atom_n.active)) \wedge$$

$$\mathit{inf} \wedge H_{clock(Disable)} ; T_{clock(Disable)} \wedge \bigwedge_{i=1}^{n} H_{\|atom_i\|} ; T_{\|atom_i\|} \wedge$$

$$\bigwedge_{i=1,j=1}^{m,k} (H_{NB^{x_i}_{atom_j}} ; NB^{x_i}_{atom_j}) \wedge$$

$$\}$$

Using the definition of a clock and lemma 6, I can define

$$H_{clock(Disable)} \equiv$$

$$\text{if } Disable \text{ then skip}$$

$$\text{else } \mathcal{T}ime := \mathcal{T}ime + 1$$

$$T_{clock(Disable)} \equiv clock(Disable)$$

and of course $H_{NB^{x_i}_{atom_j}}$ will be given as

$$H_{NB^{x_i}_{atom_j}} \widehat{=} \{$$

$$\text{if } Disable \text{ then skip}$$

$$\text{else } ($$

$$\mathcal{L}^v_{Atom} := filter(\mathcal{L}^v_{Atom}, \mathcal{T}ime) \wedge$$

$$\forall i < |\mathcal{L}^v_{Atom}| \bullet \text{if } (\mathcal{L}^v_{Atom}[i][0] = \mathcal{T}ime) \text{ then } \mathcal{V} := \mathcal{L}^v_{Atom}[i][1]$$

$$)$$

$$\}$$

and having in mind that $H_{NB^{x_i}_{atom_j}}$; $NB^{x_i}_{atom_j} \equiv NB^{x_i}_{atom_j}$.

Similarly I can define the $H_{\|atom_i\|}$ and $T_{\|atom_i\|}$. For that I will need to consider some cases.

Case 1 If $atom_i$ is an non-delayed continuous assignment statement, i.e. `assign` $v = exp(v_1, \ldots, v_n)$, then $H_{\|atom_i\|}$ will be

$$
\begin{aligned}
H_{\|atom_i\|} \quad \equiv \\
&\{(Atom_1.active =\sim (\mathcal{V}_1, \ldots, \mathcal{V}_n)) \wedge \\
&\quad \{\text{if } \neg Atom_1.active \text{ then skip} \\
&\quad\ \text{else } \mathcal{V} := exp(\mathcal{V}_1, \ldots, \mathcal{V}_n) \\
&\quad \} \\
&\}
\end{aligned}
$$

and $T_{\|atom_i\|} = \|atom_i\|$. This is because $\|atom_i\| \equiv \{H_{\|atom_i\|}\}^*$ and $H_{\|atom_i\|} \supset$ skip.

Case 2 If $atom_i$ is a delayed continuous assignment statement, i.e. `assign` $\#exp_1\ v = exp(v_1, \ldots, v_n)$, then I can similarly give definitions for $H_{\|atom_i\|}$ and $T_{\|atom_i\|}$. However, before doing this I will need to look into the definition of $\|atom_i\|$

$$\|\texttt{assign}\ \#\texttt{exp}_1\ \ v = exp_2(*)\| \mathrel{\widehat{=}}$$

$$
\begin{array}{l}
\exists t, Atom.act_1, Atom.act_2 \bullet \\
\quad \{\Box(Atom.active = Atom.act_1 \vee Atom.act_2) \wedge \\
\quad\ \{\textsf{if } \sim (*) \textsf{ then } (Atom.act_1 = \textit{true} \wedge t := \mathcal{T}ime + exp_1) \\
\quad\quad \textsf{else } (Atom.act_1 = false \wedge \\
\quad\quad\ \textsf{if } (t \neq \mathcal{T}ime) \textsf{ then } t := t \\
\quad\quad\ \textsf{else } t := t - 1 \\
\quad\quad) \wedge \\
\quad\quad \textsf{if } t = \mathcal{T}ime \textsf{ then } (Atom.act_2 = \textit{true} \wedge \mathcal{V} := exp_2(*)) \\
\quad\quad \textsf{else } (Atom.act_2 = false \wedge \textsf{skip}) \\
\quad\ \}^* \\
\quad \}
\end{array}
$$

Without loss of generality I can rename the local variables $t, Atom.act_1, Atom.act_2$ and pull them in front of the whole formula $\|\mathtt{P}\|$, i.e. I can make them global, hence I can apply lemma 5. Therefore, I can safely assume that

$$
\begin{array}{l}
H_{\|atom_i\|} \mathrel{\widehat{=}} \\
\qquad \{(Atom.active = Atom.act_1 \vee Atom.act_2) \wedge
\end{array}
$$

$$
\begin{array}{l}
\{\textsf{if } \sim(*) \textsf{ then } (Atom.act_1 = \textit{true} \wedge t := \mathcal{T}ime + exp_1) \\
\quad \textsf{else } (Atom.act_1 = false \wedge \\
\quad\quad \textsf{if } (t \neq \mathcal{T}ime) \textsf{ then } t := t \\
\quad\quad \textsf{else } t := t - 1 \\
\quad) \wedge \\
\quad \textsf{if } t = \mathcal{T}ime \textsf{ then } (Atom.act_2 = \textit{true} \wedge \mathcal{V} := exp_2(*)) \\
\quad \textsf{else } (Atom.act_2 = false \wedge \textsf{skip}) \\
\}
\end{array}
$$

and hence $T_{\|atom_i\|} = \|atom_i\|$ again because $\|atom_i\| \equiv \{H_{\|atom_i\|}\}^*$ and $H_{\|atom_i\|} \supset$ skip.

Case 3 If $atom_i$ is an always statement, i.e. `always` $statement$, then $\|atom_i\| \equiv \|statement\|$; $\|atom_i\| \equiv H_{\|statement\|}$; $T_{\|statement\|}$; $\|atom_i\|$, i.e.

$$H_{\|\texttt{always}\ \ statement\|} \mathrel{\widehat{=}} H_{\|statement\|} \text{ and}$$

$$T_{\|\texttt{always}\ \ statement\|} \mathrel{\widehat{=}} T_{\|statement\|} \text{ ; } \|atom_i\|.$$

Thus, I have to investigate the semantics of a statement which I will do later.

Case 4 If $atom_i$ is an initial statement, i.e. `initial` $statement$, then $\|atom_i\| \equiv \|statement\| \,;\, \Box(Atom.active = \textit{false}) \equiv H_{\|statement\|} \,;\, T_{\|statement\|} \,;\, \Box(Atom.active = \textit{false})$, i.e.

$$H_{\|\texttt{initial}\ \ statement\|} \mathrel{\widehat{=}} H_{\|statement\|} \text{ and}$$

$$T_{\|\texttt{initial}\ \ statement\|} \mathrel{\widehat{=}} T_{\|statement\|} \,;\, \Box(Atom.active = \textit{false})$$

and I have to look again at the semantics of statement.

Case 5 Now I will have to show that I can transform $\|statement\| \equiv H_{\|statement\|} \,;\, T_{\|statement\|}$.

Case 5.1 If statement is ε, then $\|statement\| \equiv Atom.active = \textit{false} \wedge \mathsf{skip}$, i.e. $H_{\|statement\|} \equiv (Atom.active = \textit{false} \wedge \mathsf{skip})$ and $T_{\|statement\|} \equiv \mathsf{empty}$.

Case 5.2 If statement is event control, i.e. $statement = @\ (e_1\texttt{or}\ , \ldots, \texttt{or}\ e_n)$, then

$$\|@\ (e_1\texttt{or}\ , \ldots, \texttt{or}\ e_n)\| \equiv$$

$$\mathsf{while}\ \neg \sim (\mathcal{E}_1, \ldots, \mathcal{E}_n)\ \mathsf{do}\ (\|\varepsilon\|);$$

$$(Atom.active = \textit{true} \wedge \mathsf{skip})$$

Obviously, if $\neg \sim (\mathcal{E}_1, \ldots, \mathcal{E}_n)$, then $H_{\|statement\|} \equiv H_{\|\varepsilon\|}$ and $T_{\|statement\|} \equiv T_{\|\varepsilon\|}$; $\|statement\|$. However, if $\sim (\mathcal{E}_1, \ldots, \mathcal{E}_n)$, then $H_{\|statement\|} \equiv (Atom.active = \textit{true} \wedge$ skip) and $T_{\|statement\|} =$ empty.

Case 5.3 Let me now consider the case of time delay when $statement = \# \, exp$, i.e.

$$
\begin{aligned}
&\|statement\| \equiv \\
&\quad \exists t \bullet \{t = \mathcal{T}ime + exp \wedge \\
&\qquad (\textsf{while } \mathcal{T}ime < t \textsf{ do } (\|\varepsilon\|) \, ; (Atom.active = \textit{true} \wedge \textsf{skip})) \\
&\quad \} \equiv \\
&\quad \exists t \bullet \{t = \mathcal{T}ime + exp \wedge (H_{\|statement\|} \, ; T_{\|statement\|})\}
\end{aligned}
$$

In this case

$$
H_{\|statement\|} \equiv \begin{cases} \|\varepsilon\|, & \text{if } \mathcal{T}ime < t \\ Atom.active = \textit{true} \wedge \textsf{skip}, & \text{otherwise} \end{cases}
$$

and

$$
T_{\|statement\|} \equiv \begin{cases} \textsf{while } \mathcal{T}ime < t \textsf{ do } (\|\varepsilon\|); \\ (Atom.active = \textit{true} \wedge \textsf{skip})), & \text{if } \mathcal{T}ime < t \\ \textsf{empty}, & \text{otherwise} \end{cases}
$$

Case 5.4 The rest of the statements have trivial definitions for H and T.

Having defined H, I can see that $H \supset$ skip, i.e. I can rewrite $\|\mathtt{P}\|$ using lemma 5 into the following form.

$$
\begin{aligned}
&\|\mathtt{P}\| \quad \equiv \\
&\quad \exists Atom_1.active, \ldots, Atom_n.active, \\
&\quad \mathcal{V}_i, \mathcal{X}_k, Disable, \mathcal{T}ime, \mathcal{L}^{\mathcal{X}_k}_{atom_j} \bullet \{ \\
&\qquad \mathcal{V}_i = \hat{v}_i \wedge new_{\mathcal{V}_i} = new_v \wedge \mathcal{X}_j = \hat{x}_j \wedge new_{\mathcal{X}_j} = new_{x_j} \wedge \\
&\qquad \mathcal{L}^{\mathcal{X}_k}_{atom_j} = [\ldots, [t_k, \hat{x}_k], \ldots] \wedge \mathcal{T}ime = \hat{t} \wedge \\
&\qquad \Box(Disable = (Atom_1.active \vee \ldots \vee Atom_n.active)) \wedge \\
&\qquad \mathit{inf} \wedge \\
&\qquad \{ \\
&\qquad\quad [H_{clock(Disable)} \wedge \bigwedge_{i=1}^{n} (H_{\|atom_i\|} \wedge H_{NB^{x_i}_{atom_j}})]; \\
&\qquad\quad [T_{clock(Disable)} \wedge \bigwedge_{i=1,j=1}^{m,k} (T_{\|atom_i\|} \wedge NB^{x_i}_{atom_j})] \\
&\qquad \} \\
&\quad \}
\end{aligned}
$$

and I know that

$$H_{clock(Disable)} \wedge \bigwedge_{i=1}^{n} (H_{\|atom_i\|} \wedge H_{NB^{x_i}_{atom_j}}) \quad \supset \quad \text{skip}$$

and this will be the step of $\|$`P` $\|$. Now I will have to show that the state after the step taken by both the Operational and the Denotational semantics of `P` are the same. ■

Definition 5 *$V\mathcal{A}tom$ is defined as follows.*

$v\mathcal{A}tom ::= statement \mid$ `always` $\mid statement\,;v\mathcal{A}tom$
$V\mathcal{A}tom ::=$ `initial` $\mid$ `assign` $\mid v\mathcal{A}tom$

Table 6.1: Syntax of $V\mathcal{A}tom$

Obviously, $V\mathcal{A}tom \subset \mathcal{A}tom$ and this is because `always` $\subset \mathcal{A}tom$, so $v\mathcal{A}tom \subset \mathcal{A}tom$. Analogously, `initial` and `assign` are both subsets of $\mathcal{A}tom$. It is also obvious that $V\mathcal{A}tom$ is what I can expect to see in a configuration which is part of a run of a Verilog program. This result is a part of the following theorem.

Theorem 3 *If $c = (T, ((\mathcal{A}tom_i, bool_i)), V, (E_j))$ is a non-start configuration taken from a run of a syntactically correct Verilog program, then $\mathcal{A}tom_i$ is a $V\mathcal{A}tom$ and there are several possible cases for E_j.*

1. $E_j^A = \{(\sim (v_1, \ldots, v_n), v = exp(v_1, \ldots, v_n))\}$ *and* $E_j^{NB} = \emptyset$ *for some* v, v_j

2. $E_j^A = \{(\sim (v_1, \ldots, v_n), t_j = \hat{t})\}$ *and* $E_j^{NB} = \emptyset$ *for some* $v_j, t_j, \hat{t}$

3. $E_j^A = \{(\sim (v_1, \ldots, v_n), t_j = \hat{t}), (T = t_j, v = exp(v_1, \ldots, v_n))\}$ *and* $E_j^{NB} = \emptyset$ *for some* $v, v_j, t_j, \hat{t}$

4. $E_j^A = \{(when, what)\}$ *and* $E_j^{NB} = \{(when_1, what_1), \ldots, (when_k, what_k)\}$ *where*

$$(when, what) = \begin{cases} (\textit{false}, \varepsilon) \\ (\textit{true}, \varepsilon) \\ (\textit{true}, v = \hat{v}) \\ (T = \hat{t}, \varepsilon) \\ (T = \hat{t}, v = \hat{v}) \\ (\sim (v_1, \ldots, v_n), \varepsilon) \\ (\sim (v_1, \ldots, v_n), v = \hat{v}) \end{cases}$$

and

$$(when_i, what_i) = \begin{cases} (T = \hat{t}, v = \hat{v}) \\ (\sim (v_1, \ldots, v_n), v = \hat{v}) \end{cases}$$

and $\hat{t}, \hat{v}$ *are constants.*

5. $E_j^A = S_1 \cup S_2$ *where*

$$S_1 \subset \{(when, what)\}$$

and

$$S_2 \subset \{(when_1, what_1), \ldots, (when_k, what_k)\}$$

Proof (Theorem 3) The proof is an induction on a run, i.e. I check that the theorem holds for c_1 after I apply the step on a start configuration. Then, I will see that if it holds for

c, it will hold for $Step(c)$ also. This is because I always include an event into E_j^A set when $bool_j$ is true, i.e. I schedule new events if I have that one of the events are enabled, therefore I delete one and add the next.

Case 1 occurs when the $\mathcal{A}tom_j$ is non-delayed continuous assignment, i.e. `assign` $v = exp(v_1, \ldots, v_n)$.

Case 2 occurs when the $\mathcal{A}tom_j$ is delayed continuous assignment, i.e. `assign` $v = \#exp_1\ exp(v_1, \ldots, v_n)$, $\hat{t}$ is a constant and $t_j \geq \|T\|$.

Case 3 occurs when the $\mathcal{A}tom_j$ is delayed continuous assignment, i.e. `assign` $v = \#exp_1\ exp(v_1, \ldots, v_n)$, $\hat{t}$ is a constant and $t_j < \|T\|$.

Case 4 occurs when the $\mathcal{A}tom_j$ is a behavioural atom and the cases that follow are

$$(when, what) = \begin{cases} (\textit{false}, \varepsilon) & \text{, if head of } \mathcal{A}tom_j \text{ is } \varepsilon \\ (\textit{true}, \varepsilon) & \text{, if head of } \mathcal{A}tom_j \text{ is } v <= exp \\ (\textit{true}, \varepsilon) & \text{, if head of } \mathcal{A}tom_j \text{ is } v <= \#exp_1\ exp_2 \\ (\textit{true}, v = \hat{v}) & \text{, if head of } \mathcal{A}tom_j \text{ is } v = exp \\ (T = \hat{t}, \varepsilon) & \text{, if head of } \mathcal{A}tom_j \text{ is } \#exp \\ (T = \hat{t}, v = \hat{v}) & \text{, if head of } \mathcal{A}tom_j \text{ is } v = \#exp_1\ exp_2 \\ (\sim (v_1, \ldots, v_n), \varepsilon) & \text{, if head of } \mathcal{A}tom_j \text{ is } @(v_1, \ldots, v_n) \\ (\sim (v_1, ., v_n), v = \hat{v}) & \text{, if head of } \mathcal{A}tom_j \text{ is } v = @(v_1, \ldots, v_n)\ exp \end{cases}$$

and

$$(when_i, what_i) = \begin{cases} (\textit{true}, v = \hat{v}) & \text{, if head of } \mathcal{A}tom_j \text{ is } v <= exp \\ (T = \hat{t}, v = \hat{v}) & \text{, if head of } \mathcal{A}tom_j \text{ is } v <= \#exp_1\ exp_2 \end{cases}$$

Case 5 occurs at non-blocking assign event activation, i.e. when τ_2 is true. This is the only case when there might be more that 2 events in E_j^A. However, right on the next step they all will be executed, because if I look at the definition of $Step$ when τ_2, I will see that I add only those non-blocking assign events which the function $\uparrow$ selects, i.e. they are activated according to the interpretation $\|.\|$. I only have to note now that the variable set V and the time T are left unchanged in a τ_2 step, i.e. the interpretation will validate the same events as active and will execute them at the very next step. ■

The denotational semantics of $V\mathcal{A}tom$ is easily derived. I only have to define $\|statement\ ; v\mathcal{A}tom\| \ \widehat{=}\ \|statement\|\ ;\ \|v\mathcal{A}tom\|$. Now I can define properly what I mean by deriving denotational semantics out of the operational configuration.

6.3.1 From a configuration to an ITL state

How can I construct a state out of a configuration? Let me have a configuration

$$c = (T, ((\mathcal{A}tom_1, bool_1), \ldots, (\mathcal{A}tom_n, bool_n)), V, ((E_1^A, E_1^{NB}), \ldots, (E_n^A, E_n^{NB}))).$$

from c I can construct an ITL state by

- $\mathcal{T}ime = T$

- The values of all $\mathcal{G}lobal\ \mathcal{V}ariable$s can be taken from their counterparts in the variable set V. Let $(x, value, new_x) \in V$. Then the corresponding global variable must be $\mathcal{X} = value$. However, I know that for the ITL state I actually have two variables, namely $X = value$ and $new_{\mathcal{X}} = new_x$.

- If the event $(t, v = value) \in E_i^{NB}$, then I will put $[t, value]$ in the list $\mathcal{L}^v_{Atom_i}$.

6.3.2 Constructing an ITL formula

Now I have to determine the corresponding ITL formula. I know the general form of $\|\mathtt{P}\ \|$, I only have to see what $H_{\|atom_i\|}$ and $T_{\|atom_i\|}$ are for each atom. It is quite trivial to note that $T_{\|atom_i\|} = \|\mathcal{A}tom_i\|$. However, for $H_{\|atom_i\|}$ I will need to consider some cases on the event lists (E_j) for each $\mathcal{A}tom_j$. Here I will apply the result of theorem 3.

1. If $E_j^A = \{(\sim (v_1, \ldots, v_n), v = exp(v_1, \ldots, v_n))\}$ and $E_j^{NB} = \emptyset$ for some v, v_j, then I am having a non-delayed continuous assignment atom and I know what the H of such an atom is.

2. If $E_j^A = \{(\sim (v_1, \ldots, v_n), t_j = \|T + exp_1\|)\}$ and $E_j^{NB} = \emptyset$ for some v_j, t_j, then I have a delayed continuous assignment with the local variable $t < \mathcal{T}ime$, for which I know the definition.

3. If $E_j^A = \{(\sim (v_1, \ldots, v_n), t_j = \|T + exp_1\|), (T = t_j, v = exp(v_1, \ldots, v_n))\}$ and

$E_j^{NB} = \emptyset$ for some v, v_j, t_j, this is again a delayed continuous assignment with $t \geq \mathcal{T}$ for which I know the definitions of H and T.

4. If $E_j^A = \{(when, what)\}$ and $E_j^{NB} = \{(when_1, what_1), \ldots, (when_k, what_k)\}$, then I have a behavioural atom, i.e. an initial or an always atom. For those, I need to look into the definition of $statement$ and $v\mathcal{A}tom$.

   - If $(when, what) = (\mathit{false}, \varepsilon)$, i.e. I have an ε as a statement. In this case I know what the definitions are.
   - If $(when, what) = (\mathit{true}, \varepsilon)$, i.e. I have a non-blocking assignment (delayed or not) and this is treated just below.
   - $(when, what) = (\mathit{true}, v = \hat{v})$, then I have a simple blocking assignment, i.e. $H_{\|\mathtt{v} = exp\|} \mathrel{\widehat{=}} \{(Atom.active = \mathit{true}) \wedge \mathcal{V} := exp\}$.
   - If $(when, what) = (T = \hat{t}, \varepsilon)$, then I have just time delay and in this case $H_{\|\# \ exp\|} \mathrel{\widehat{=}} \exists t \bullet \{t = \mathcal{T}ime + exp \wedge \mathsf{while}\ \mathcal{T}ime < t\ \mathsf{do}\ (\|\varepsilon\|)\}$
   - If $(when, what) = (T = \hat{t}, v = \hat{v})$, then I have a delayed blocking assignment $H_{\|\mathtt{v} = \#exp_1 \ exp\|} \mathrel{\widehat{=}} \exists x \bullet \{x = exp \wedge (\|\eta\| \ ; ((Atom.active = \mathit{true}) \wedge \mathcal{V} := x))\}$
   - If $(when, what) = (\sim (v_1, \ldots, v_n), \varepsilon)$, then $H_{\|@ \ (e_1 \mathtt{or} \ , \ldots, \mathtt{or} \ e_n)\|} \mathrel{\widehat{=}} \mathsf{while}\ \neg \sim (\mathcal{E}_1, \ldots, \mathcal{E}_n)\ \mathsf{do}\ (\|\varepsilon\|)$
   - If $(when, what) = (\sim (v_1, \ldots, v_n), v = \hat{v})$, then I have event control de-

layed blocking assignment which has $H_{\|\mathbf{v} =\eta\ exp\|} \widehat{=} \exists x \bullet \{x = exp \wedge (\|\eta\| ; ((Atom.active = \mathit{true}) \wedge \mathcal{V} := x))\}$. In this case η is strictly $@(v_1, \ldots, v_n)$.

I have to consider only the case of the non-blocking assignment. If there is $(when_j, what_j) \in E_j^{NB}$ and

- $(when_j, what_j) = (T = \hat{t}, v = \hat{v})$ and the constant $\hat{t} > \|T\|$, then I have time delayed non-blocking assignment and $H_{\|\mathbf{v} <=\#\ exp_1\ exp_2\|} \widehat{=}$ $\{(Atom.active = \mathit{true}) \wedge \mathcal{L}^v_{Atom} := \mathcal{L}^v_{Atom} + [[\mathcal{T}ime + exp_1, exp_2]]\}$, where $exp_1 = \|T\| - \hat{t}$.
- $(when_j, what_j) = (\mathit{true}, v = \hat{v})$, then I have a non-delayed non-blocking assignment and $H_{\|\mathbf{v} <=exp\|} \widehat{=} \{(Atom.active = \mathit{true}) \wedge \mathcal{L}^v_{Atom} := \mathcal{L}^v_{Atom} + [[\mathcal{T}ime, exp]]\}$,

6.3.3 The final result

Definition 6 *I will say that an ITL state and an ITL formula are equivalent to a configuration if the state can be derived from the configuration using the procedure given in section 6.3.1 and the formula can be derived from the configuration using the procedure given in section 6.3.2.*

Now I can formulate and prove the following theorem.

Theorem 4 *If the initial states are equivalent, then both semantics take equivalent steps.*

Proof (Theorem 4) It should be obvious that for any program `P` , the first configuration c_1 of a run and $\|$`P` $\|$ are equivalent (correspondent) according to the definition 6 given above.

Next, I take a configuration, from which I construct an ITL state plus a formula corresponding to the configuration (the procedures are given in sections 6.3.1 and 6.3.2). Then I take independent steps for the operational semantics and the denotational semantics and I check that the definition for corresponding configuration state and formula after the step holds as well. Therefore I conclude that the operational and denotational semantics are equivalent. ■

6.4 Summary

This is the most technical part of the thesis. The proof of the equivalence between my denotational and operational formalisms for Verilog allows me to claim a higher level of confidence in my results. I can now reason both operationally and denotationally about a Verilog program, therefore I can reason both about properties and machines that implement them.

Chapter 7

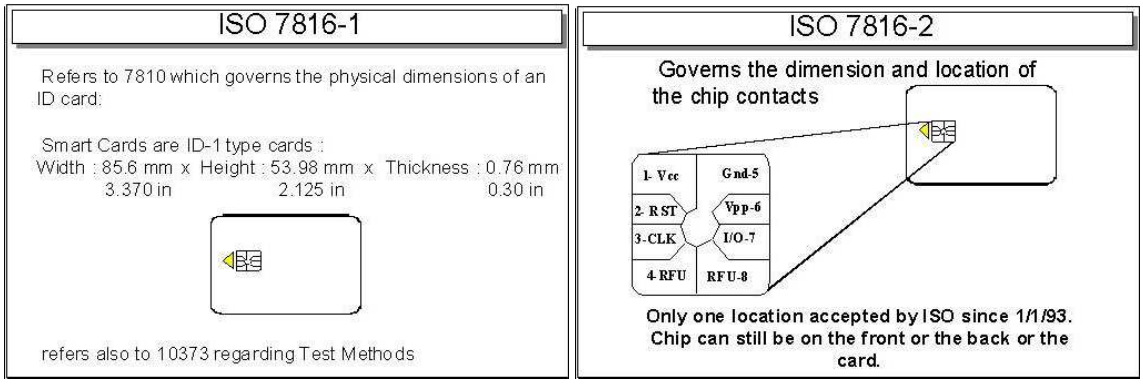
A Case Study — Smart Card Application

A partial refinement of a mixed hardware/software application, together with all supporting proofs, is shown in this chapter. All major steps through the development are given. This application is a "proof-of-concept" only. However, I believe the technique is practical.

7.1 Introduction

A smart card application is required to perform Rivest Shamir Adleman (RSA) [92] encryption and decryption with *a private key* on the smart card's chip itself [75]. The application should consist of a smart card and a reader. Both the reader and the smart card should comply with International Standardisation Organisation (ISO) 7816 set of stan-

dards [91] for size and pin configuration.



(a) (b)

Figure 7.1: ISO 7816 Standards

7.1.1 Electrical Signals Description

The following are the pin assignments as defined by [91]. See figure (7.1(b)) for reference.

1. V_{CC} — Power supply input. This contact is used to supply the power voltage by the reader.

2. RST — Reset signal supplied from the reader.

3. CLK — Clocking or timing signal supplied from the reader.

4. RFU — Reserved for Further Use.

5. GND — Ground reference voltage supplied from the reader.

6. V_{pp} — Programming voltage input. This contact is to supply the voltage required to program or to erase the internal non-volatile memory and it is supplied from the reader.

7. I/O — Input or Output for serial data to the integrated circuit inside the card. This contact is used as input (reception mode) or output (transmission mode) for data exchange.

8. RFU — Reserved for Further Use.

7.1.2 Operating Procedure for Integrated Circuit(s) Cards

This operating procedure applies to every smart card with contacts.

The dialogue between the reader and the the card shall be conducted through the consecutive operations:

- connection and activation of the contacts by the reader, i.e. power-up.

- reset of the card, i.e. `reset` command.

- answer to reset by the card.

- subsequent information exchange between the card and the reader.

- deactivation of the contacts by the reader.

For a detailed description of each of these operations refer to [91].

7.2 Requirements for a Reader

The reader should supply the smart card with appropriate power and frequency sources and be an interface device between the smart card and a host computer. The host and the reader should be linked with a serial cable with RS 232 protocol.

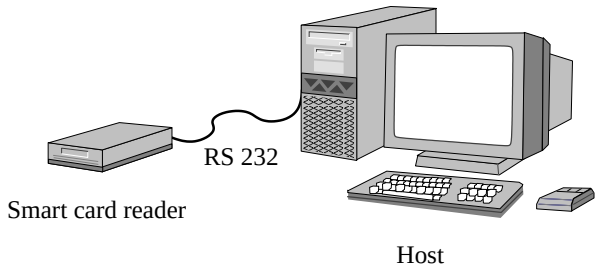


Figure 7.2: A Reader and a Host

7.3 Requirements for a Smart Card

The smart card should perform the following, all-in-all, on its chip. It should, in no circumstances, disclose the private key stored inside.

1. After the initial `reset`, it must respond with results from its self-test procedure as an answer to reply.

2. Authenticate the reader via PIN, password or biometric identification. For the whole lifetime of the card, there must be at least one successful authentication within every period that covers three unsuccessful ones. This means that the life of the card must

end and the data in the card destroyed, if there are no successful authentications within a period of three unsuccessful ones.

3. Implement the following commands issued by the reader.

   - `make_keys`: Generate a pair of public and private keys and store them in its Electrically Erasable Programmable Read Only Memory (EEPROM) for further use.

   - `give_public`: Give the public key to the reader.

   - `encrypt` and `decrypt`: Encrypt/Decrypt a data block given by the reader using the private key. Data block is a sequence of bytes with fixed length. When the card receives an `encrypt` or `decrypt` command it must respond with the encrypted/decrypted result.

I will adopt the convention that `reset` is a **privileged** command and `make_keys`, `give_public`, `encrypt` and `decrypt` are **unprivileged** commands.

The reader can asynchronously reset the card at any time. This should not damage any data stored in the card or the card itself. Here we will specify that a reset signal on the RST pin of the card, see pin 2-RST in figure (7.1(b)), is equivalent to a privileged `reset` command. Once reset, the card must wait for user authentication before any other command.

Only after a successful authentication the card is required to respond to the reader's unprivileged commands, i.e. `make_keys`, `give_public`, `encrypt` and `decrypt` in any order. The `give_public`, `encrypt` and `decrypt` commands can be issued by the reader only if there is a pair of public and private keys stored in the card. A unprivileged command cannot interrupt the card, if the later is responding to a command.

7.4 Top Level Specification

I will start with the assumption that the card and the reader are a single system. This, at a later stage, will allow me to formally specify each individual component in terms of its environment, i.e. in terms of its counterpart. Going back to page 27, I can see that it represents Step 0 from Figure 3.1 on that page. In the following I use the formalisations:

- `activate` activates the contacts, i.e. the card is in the reader.
- `reset` resets the card successfully.
- `not_reset` stays for unsuccessful reset.
- `authenticate` represents successful authentication.
- `not_authent` represents unsuccessful authentication.
- `command` executes unprivileged command given to the card by the reader.
- `deactivate` deactivates the contacts, i.e. card not in the reader.

I will also accept, that these actions are mutually exclusive to one another, i.e.

$$Req_{(7.1)} \equiv \mathtt{any}_1 \wedge \mathtt{any}_2 \equiv \mathit{false} \tag{7.1}$$

for any two *different* actions of the possible given just above. This assumption would be quite natural when one considers that the card and the reader could perform only one action at a time.

The top level specification must satisfy several properties. In the following I will assume F, F_1, F_2 and F_3 to be all finite formulas, i.e. $F \supset \mathit{finite}$ and $F_j \supset \mathit{finite}$ for $j = 1, 2, 3$.

1. A successful `reset` must always be preceded by `activate`, i.e. one can only reset the card if it has been activated.

$$\begin{aligned} Req_{(7.2)} \mathrel{\widehat{=}} \Box(& \\ & F \,;\, \mathtt{reset} \supset \\ & (F \supset \Diamond(\mathtt{activate} \,;\, \neg \circledDiamond \mathtt{deactivate})) \\ ) & \end{aligned} \tag{7.2}$$

2. Every `authenticate` must be preceded by a `reset`, i.e. one can authenticate the card only if the last attempt to reset it has been successful and it has not been

deactivated since then.

$$
\begin{aligned}
Req_{(7.3)} \mathrel{\widehat{=}} \Box(& \\
& F \,;\, \texttt{authenticate} \quad \supset \\
& (F \ \supset \ \Diamond(\texttt{reset} \,;\, \neg \, \Diamond\!\!\!\!\cdot(\texttt{not\_reset} \vee \texttt{deactivate}))) \\
) &
\end{aligned}
\tag{7.3}
$$

3. All unprivileged commands must always be preceded by a successful authentication.

$$
\begin{aligned}
Req_{(7.4)} \mathrel{\widehat{=}} \Box(& \\
& ((F \,;\, \texttt{command}) \wedge \neg(F \ \supset \ \Diamond\texttt{command})) \quad \supset \\
& (F \ \supset \ \Diamond\texttt{authenticate}) \\
) &
\end{aligned}
\tag{7.4}
$$

4. every period that covers three unsuccessful authentications and not a successful one

must be followed by a period where no unprivileged command is executed.

$$
\begin{aligned}
Req_{(7.5)} \mathrel{\widehat{=}} \Box(& \\
& ((\texttt{not\_authent} \,;\, F_1 \,;\, \texttt{not\_authent} \,;\, F_2; \\
& \qquad \texttt{not\_authent} \,;\, F_3) \wedge \\
& \neg(F_1 \supset \diamondsuit\texttt{authenticate}) \wedge \neg(F_2 \supset \diamondsuit\texttt{authenticate}) \\
&) \quad \supset \quad (F_3 \supset (\neg\Diamond\texttt{command})) \\
&)
\end{aligned}
\tag{7.5}
$$

The automaton in figure (7.3) represents the possible transitions of a system composed of a reader and a smart card working together. The automaton has a start state but it lacks a final state. Instead it has a “sink” (q_{10}) interpreted as a “dead” end. Notably, each label is a name of a procedure.

I can specify the behaviour displayed by the automaton in figure 7.3 by a set of compositional properties that encode the automaton. The top-level formula Φ given by (7.6) can be composed into groups of the sub-formulas, thus

$$\Phi \mathrel{\widehat{=}} \Box \bigwedge_{i=0}^{10} \phi_i \tag{7.6}$$

I shall recall that a state is a boolean expression, thus it is a state formula in ITL. At this moment I will only specify that all states q_j are different, i.e. mutually exclusive. Later I

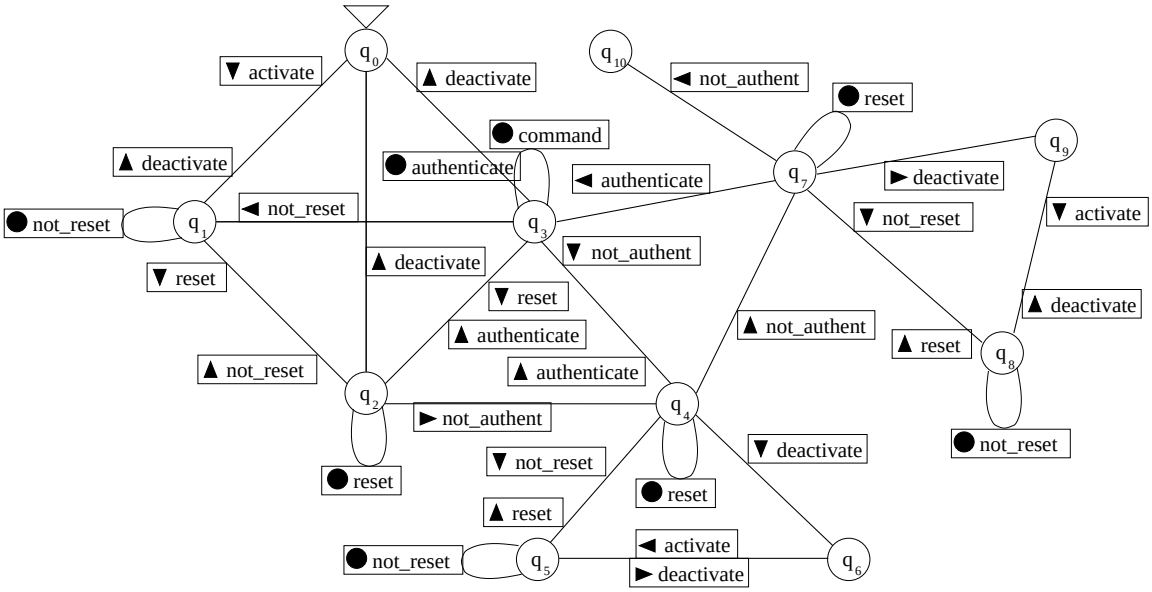


Figure 7.3: Smart Card as a Finite State Automaton

will give precise definitions for each state.

(7.7) $$\phi_0 \mathrel{\widehat{=}} q_0 \ \supset\ \texttt{activate} \wedge \mathit{fin}\ q_1$$

$$\phi_1 \mathrel{\widehat{=}} f_{q_{1a}} \vee f_{q_{1b}} \vee f_{q_{1c}}$$

(7.8a) $$f_{q_{1a}} \mathrel{\widehat{=}} q_1 \ \supset\ \texttt{not\_reset} \wedge \mathit{fin}\ q_1$$

(7.8b) $$f_{q_{1b}} \mathrel{\widehat{=}} q_1 \ \supset\ \texttt{deactivate} \wedge \mathit{fin}\ q_0$$

(7.8c) $$f_{q_{1c}} \mathrel{\widehat{=}} q_1 \ \supset\ \texttt{reset} \wedge \mathit{fin}\ q_2$$

$$\phi_2 \mathrel{\widehat{=}} f_{q_{2a}} \vee f_{q_{2b}} \vee f_{q_{2c}} \vee f_{q_{2d}} \vee f_{q_{2e}}$$

(7.9a) $$f_{q_{2a}} \mathrel{\widehat{=}} q_2 \supset \texttt{reset} \wedge \mathit{fin}\ q_2$$

(7.9b) $$f_{q_{2b}} \mathrel{\widehat{=}} q_2 \supset \texttt{not\_reset} \wedge \mathit{fin}\ q_1$$

(7.9c) $$f_{q_{2c}} \mathrel{\widehat{=}} q_2 \supset \texttt{deactivate} \wedge \mathit{fin}\ q_0$$

(7.9d) $$f_{q_{2d}} \mathrel{\widehat{=}} q_2 \supset \texttt{authenticate} \wedge \mathit{fin}\ q_3$$

(7.9e) $$f_{q_{2e}} \mathrel{\widehat{=}} q_2 \supset \texttt{not\_authent} \wedge \mathit{fin}\ q_4$$

$$\phi_3 \mathrel{\widehat{=}} f_{q_{3a}} \vee f_{q_{3b}} \vee f_{q_{3c}} \vee f_{q_{3d}} \vee f_{q_{3e}} \vee f_{q_{3f}}$$

(7.10a) $$f_{q_{3a}} \mathrel{\widehat{=}} q_3 \supset \texttt{deactivate} \wedge \mathit{fin}\ q_0$$

(7.10b) $$f_{q_{3b}} \mathrel{\widehat{=}} q_3 \supset \texttt{not\_reset} \wedge \mathit{fin}\ q_1$$

(7.10c) $$f_{q_{3c}} \mathrel{\widehat{=}} q_3 \supset \texttt{reset} \wedge \mathit{fin}\ q_2$$

(7.10d) $$f_{q_{3d}} \mathrel{\widehat{=}} q_3 \supset \texttt{command} \wedge \mathit{fin}\ q_3$$

(7.10e) $$f_{q_{3e}} \mathrel{\widehat{=}} q_3 \supset \texttt{authenticate} \wedge \mathit{fin}\ q_3$$

(7.10f) $$f_{q_{3f}} \mathrel{\widehat{=}} q_3 \supset \texttt{not\_authent} \wedge \mathit{fin}\ q_4$$

$$\phi_4 \mathrel{\widehat{=}} f_{q_{4a}} \vee f_{q_{4b}} \vee f_{q_{4c}} \vee f_{q_{4d}} \vee f_{q_{4e}}$$

(7.11a) $$f_{q_{4a}} \mathrel{\widehat{=}} q_4 \supset \texttt{authenticate} \wedge \mathit{fin}\ q_3$$

(7.11b) $$f_{q_{4b}} \mathrel{\widehat{=}} q_4 \supset \texttt{reset} \wedge \mathit{fin}\ q_4$$

(7.11c) $$f_{q_{4c}} \mathrel{\widehat{=}} q_4 \supset \texttt{not\_reset} \wedge \mathit{fin}\ q_5$$

(7.11d) $$f_{q_{4d}} \mathrel{\widehat{=}} q_4 \supset \texttt{deactivate} \wedge \mathit{fin}\ q_6$$

(7.11e) $$f_{q_{4e}} \mathrel{\widehat{=}} q_4 \supset \texttt{not\_authent} \wedge \mathit{fin}\ q_7$$

$$\phi_5 \mathrel{\widehat{=}} f_{q_{5a}} \vee f_{q_{5b}} \vee f_{q_{5c}}$$

(7.12a) $$f_{q_{5a}} \mathrel{\widehat{=}} q_5 \supset \texttt{reset} \wedge \mathit{fin}\ q_4$$

(7.12b) $$f_{q_{5b}} \mathrel{\widehat{=}} q_5 \supset \texttt{not\_reset} \wedge \mathit{fin}\ q_5$$

(7.12c) $$f_{q_{5c}} \mathrel{\widehat{=}} q_5 \supset \texttt{deactivate} \wedge \mathit{fin}\ q_6$$

(7.13) $$\phi_6 \mathrel{\widehat{=}} q_6 \supset \texttt{activate} \wedge \mathit{fin}\ q_5$$

$$\phi_7 \mathrel{\widehat{=}} f_{q_{7a}} \vee f_{q_{7b}} \vee f_{q_{7c}} \vee f_{q_{7d}} \vee f_{q_{7e}}$$

(7.14a) $$f_{q_{7a}} \mathrel{\widehat{=}} q_7 \supset \texttt{authenticate} \wedge \mathit{fin}\ q_3$$

(7.14b) $$f_{q_{7b}} \mathrel{\widehat{=}} q_7 \supset \texttt{reset} \wedge \mathit{fin}\ q_7$$

(7.14c) $$f_{q_{7c}} \mathrel{\widehat{=}} q_7 \supset \texttt{not\_reset} \wedge \mathit{fin}\ q_8$$

(7.14d) $$f_{q_{7d}} \mathrel{\widehat{=}} q_7 \supset \texttt{deactivate} \wedge \mathit{fin}\ q_9$$

(7.14e) $$f_{q_{7e}} \mathrel{\widehat{=}} q_7 \supset \texttt{not\_authent} \wedge \mathit{fin}\ q_{10}$$

$$\phi_8 \mathrel{\widehat{=}} f_{q_{8a}} \vee f_{q_{8b}} \vee f_{q_{8c}}$$

(7.15a) $$f_{q_{8a}} \mathrel{\widehat{=}} q_8 \supset \texttt{reset} \wedge \mathit{fin}\ q_7$$

(7.15b) $$f_{q_{8b}} \mathrel{\widehat{=}} q_8 \supset \texttt{not\_reset} \wedge \mathit{fin}\ q_8$$

(7.15c) $$f_{q_{8c}} \mathrel{\widehat{=}} q_8 \supset \texttt{deactivate} \wedge \mathit{fin}\ q_9$$

(7.16) $$\phi_9 \mathrel{\widehat{=}} q_9 \supset \texttt{activate} \wedge \mathit{fin}\ q_8$$

(7.17) $$\phi_{10} \mathrel{\widehat{=}} q_{10} \supset \mathsf{empty} \wedge \mathit{fin}\ q_{10}$$

I can easily show that, in conjunction to the natural condition

$$q_0 \wedge \mathtt{any}^*,$$

which says that "I start from q_0 and I only use the commands from $\mathtt{any}$", the formulas representing the automaton imply the top-level requirements $Req_{(7.2)}$, $Req_{(7.3)}$, $Req_{(7.4)}$ and $Req_{(7.5)}$ via sequential composition, i.e. the following theorem holds.

Theorem 5 *The formula defined by* (7.6) *is a refinement for the conjunction of the formulas defined by* (7.2), (7.3), (7.4) *and* (7.5). *In other words,*

$$Req_{(7.2)} \wedge Req_{(7.3)} \wedge Req_{(7.4)} \wedge Req_{(7.5)} \sqsubseteq q_0 \wedge \mathtt{any}^* \wedge \Phi$$

Proof (Theorem 5) I will show that the following two refinements hold.

$$Req_{(7.2)} \sqsubseteq q_0 \wedge \mathtt{any}^* \wedge \Phi$$

$$Req_{(7.5)} \sqsubseteq q_0 \wedge \mathtt{any}^* \wedge \Phi$$

The rest can be proven with similar techniques so I will not give the full proof here.

Lemma 7

$$q_0 \wedge \mathtt{any}^* \wedge \Phi \quad \supset \quad Req_{(7.2)}$$

Proof (Lemma 7) I will recall that, following

$$\vdash \quad P \quad \Rightarrow \quad \vdash \quad \Box P,$$

in ITL I can infer that

$$\vdash \quad \Box P \supset Q \quad \Rightarrow \quad \vdash \quad \Box(\Box P \supset Q) \quad \Rightarrow$$

$$\Rightarrow \quad \vdash \quad \Box\Box P \supset \Box Q \quad \Rightarrow \quad \vdash \quad \Box P \supset \Box Q$$

and having in mind that both Φ and $Req_{(7.2)}$ have the form $\Box P$, it will be sufficient for the lemma to show

(7.18) $$q_0 \wedge \texttt{any}^* \wedge \Box \bigwedge_{i=0}^{10} \phi_i \wedge F \,;\texttt{reset} \quad \supset \quad F \supset \Diamond(\texttt{activate} \,;\neg \Diamond_{\texttt{a}} \texttt{deactivate}),$$

where $F \supset$ *finite*. I will first show that

(7.19) $$\texttt{any}^* \wedge \Box \bigwedge_{i=0}^{10} \phi_i \quad \supset \quad \Box_{\texttt{a}}((\texttt{deactivate} \,;\texttt{any}) \supset (\texttt{any} \equiv \texttt{activate})).$$

I will recall (7.8b), (7.9c), (7.10a), (7.11d), (7.12c), (7.14d), (7.15c) which give me

(7.20) $$\texttt{any}^* \wedge \Box \bigwedge_{i=0}^{10} \phi_i \quad \supset \quad \Box_{\texttt{a}}(\texttt{deactivate} \supset (\mathit{fin}\ q_0 \vee \mathit{fin}\ q_6 \vee \mathit{fin}\ q_9)).$$

Now I will combine (7.20) with (7.7), (7.13) and (7.16) by using the compositional proof rule (3.3) in section 3.5 and this will give me (7.19). It should be obvious by now that I can prove (7.18) by contradiction. Let me assume that for some F_1, F_2 and F_3

$$q_0 \wedge \texttt{any}^* \wedge \Box \bigwedge_{i=0}^{10} \phi_i \wedge F \,;\texttt{reset} \wedge (F \equiv F_1 \,;\texttt{activate} \,; F_2 \,;\texttt{deactivate} \,; F_3)$$

Having $\texttt{any}^*$ in mind I can guarantee that as with F_1 and F_2, $F_3 \supset \texttt{any}^*$. Of course, $F_3 \supset$ *more* because otherwise I would have had $\texttt{deactivate} \,;\texttt{reset}$ which contradicts with (7.19) and also $F_3 \supset$ *finite*. Again (7.19) gives me that $F_3 \supset \texttt{activate} \,; F'$ and for F' I can apply the same reasoning. However, the length of F' is smaller than the length of F_3, i.e. I can conclude (7.18) and this proves lemma 7. ■

Lemma 8

$$q_0 \wedge \texttt{any}^* \wedge \Phi \quad \supset \quad Req_{(7.5)}$$

Proof (Lemma 8) Following the same argument as with lemma 7 I can say that it will be sufficient for the lemma to show

(7.21)

$$q_0 \wedge \texttt{any}^* \wedge \Box \bigwedge_{i=0}^{10} \phi_i \wedge (\texttt{not\_authent} \,; F_1 \,; \texttt{not\_authent} \,; F_2 \,; \texttt{not\_authent} \,; F_3) \wedge$$
$$\neg(F_1 \supset \diamondsuit\texttt{authenticate}) \wedge \neg(F_2 \supset \diamondsuit\texttt{authenticate})$$
$$\supset \quad (F_3 \supset (\neg\diamondsuit\texttt{command}))$$

At this point I only need to note that after three `not_authent` commands, I must end-up in q_{10} and one look at (7.17) on page 148 is sufficient for the lemma. ■

This proves the theorem. ■

7.5 "Smart Card — Reader" Split

I have shown the correct behaviour of the system as a whole and I now need to consider the components involved, namely the smart card and the reader, with the communication between them. For that purpose I will need to define the interface between them. I will refer here to Figure 3.1 on page 27 where I will see that this forms Step 1.

An inspection of figure 7.1 on page 137 gives me a good idea of what the interface should look like. I have 8 pins between the card and the reader and 6 of them are used. Therefore I will use these pins as shared variables between the two components of my system.

The description of the standard [91] tells me that all variables $\mathtt{V_{cc}}, \mathtt{RST}, \mathtt{CLK}, \mathtt{Gnd}, \mathtt{V_{pp}}$ and $\mathtt{I/O}$ are boolean and even gives me the actual electrical characteristics, which can also be added as requirements at this point. However I will not go in that details and will simply use the notation $\mathtt{V_{cc}}$ for high and $\neg\mathtt{V_{cc}}$ for low level on the $\mathtt{V_{cc}}$ pin.

However, for the *different* states q_0, q_6 and q_9 to be distinguishable, the smart card should always know how many unsuccessful authentications there have been since the time of the last successful one. This leads me to introduce an additional state variable `BLK`

which will keep the number of these unsuccessful authentications and the definition of it shall be $q_0(\texttt{BLK}) = 0, q_6(\texttt{BLK}) = 1$ and $q_9(\texttt{BLK}) = 2$.

I am now ready to formalise the different procedures involved in the operation of the smart card.

- `activate` is described in the standard [91] as follows:

  - `RST` is in low state; Here I will change this to high and will bring the `RST` to low at the beginning of the `reset` procedure.
  - $\texttt{V}_{\texttt{cc}}$ shall be powered and shall stay powered till `deactivate`;
  - `I/O` in the interface device shall be put in reception mode, i.e. high state;
  - $\texttt{V}_{\texttt{pp}}$ shall be raised to low state;
  - `CLK` shall be provided with a suitable and stable clock until `deactivate`.

  I will need to keep the definitions regarding the $\texttt{V}_{\texttt{cc}}$ and `CLK` true over every interval bounded by `activate` and `deactivate`. Therefore, to reduce the specification, we will assume

(7.22) $$\Box(\texttt{activate} \,;\, F \,;\, \texttt{deactivate} \quad \supset \quad (F \;\supset\; \boxdot\, \texttt{V}_{\texttt{cc}} \wedge clock(\texttt{CLK}))$$

where the definition of $clock$ is as follows

$$clock(\mathtt{CLK}) \mathrel{\widehat{=}} (\mathtt{CLK}\ \textit{gets}\ \neg\mathtt{CLK})$$

and now I can define the process `activate` in ITL as the following interval and finite formula.

$$\mathtt{activate} \mathrel{\widehat{=}} \textit{finite} \wedge \boxdot(\mathtt{RST}) \wedge \Diamond(\boxdot\ \mathtt{I/O}) \wedge \boxdot(\neg \mathtt{V_{pp}}) \wedge \mathsf{stable}\,(\mathtt{BLK})$$

where I can split `activate` into two subparts $\mathtt{activate}_r$ and $\mathtt{activate}_c$ for the reader and the card correspondingly.

$$\mathtt{activate}_r \mathrel{\widehat{=}} \textit{finite} \wedge \boxdot(\mathtt{RST}) \wedge \Diamond(\boxdot\ \mathtt{I/O}) \wedge \boxdot(\neg \mathtt{V_{pp}})$$

$$\mathtt{activate}_c \mathrel{\widehat{=}} \mathsf{stable}\,(\mathtt{BLK})$$

and where I can compositionally show

$$q_0\ \supset\ \mathtt{activate} \wedge \textit{fin}\ q_1 \quad \sqsubseteq \quad q_0\ \supset\ \mathtt{activate}_r \wedge \mathtt{activate}_c \wedge \textit{fin}\ q_1$$

which is further refined by the conjunction of

$$q_0 \wedge Ass_r \wedge \texttt{activate}_r^i \supset \texttt{activate}_r \wedge \textit{fin}\ q_1$$

and

$$q_0 \wedge Ass_c \wedge \texttt{activate}_c^i \supset \texttt{activate}_c \wedge \textit{fin}\ q_1$$

where $Ass_r \mathrel{\widehat{=}} \texttt{activate}_c$ is an assumption for the reader and $Ass_c \mathrel{\widehat{=}} \texttt{activate}_r$ is an assumption for the card and in the mean time $\texttt{activate}_r^i$ is an implementation for the reader and $\texttt{activate}_c^i$ implements the smart card. As it may be now noticeable that the card and the reader assume the requirements for their counterparts for their correct work. It may be obvious now that the implementation of the reader $\texttt{activate}_r^i$ is best suited for software and the implementation for the smart card $\texttt{activate}_c^i$ must be hardware.

- Again following the standard, `deactivate` is given as
  - State low on `RST`;
  - State low on `CLK`;
  - $\texttt{V}_{\texttt{pp}}$ low;
  - State low on `I/O`;

– $\mathtt{V_{cc}}$ low;

which can easily be defined as the possibly infinite formula

$$\mathtt{deactivate} \mathrel{\widehat{=}} \boxdot_{\mathrm{m}} (\neg \mathtt{RST} \wedge \neg \mathtt{CLK} \wedge \neg \mathtt{V_{pp}} \wedge \neg \mathtt{I/O} \wedge \neg \mathtt{V_{cc}}) \wedge \mathsf{stable}\,(\mathtt{BLK})$$

As with `activate` I can split and formulate the components compositionally into

$$\mathtt{deactivate}_r \mathrel{\widehat{=}} \boxdot_{\mathrm{m}} (\neg \mathtt{RST} \wedge \neg \mathtt{CLK} \wedge \neg \mathtt{V_{pp}} \wedge \neg \mathtt{I/O} \wedge \neg \mathtt{V_{cc}})$$

$$\mathtt{deactivate}_c \mathrel{\widehat{=}} \mathsf{stable}\,(\mathtt{BLK})$$

and I can again compositionally show

$$q_1 \supset \mathtt{deactivate} \wedge \mathit{fin}\ q_0 \quad \sqsubseteq \quad q_1 \supset \mathtt{deactivate}_r \wedge \mathtt{deactivate}_c \wedge \mathit{fin}\ q_0$$

which is further refined by the conjunction of

$$q_1 \wedge Ass_r \wedge \mathtt{deactivate}_r^i \supset \mathtt{deactivate}_r \wedge \mathit{fin}\ q_0$$

and

$$q_1 \wedge Ass_c \wedge \mathtt{deactivate}_c^i \supset \mathtt{deactivate}_c \wedge \mathit{fin}\ q_0$$

where $Ass_r \mathrel{\widehat{=}} \texttt{deactivate}_c$ and $Ass_c \mathrel{\widehat{=}} \texttt{deactivate}_r$ are assumptions for the reader and the card correspondingly and in the mean time $\texttt{deactivate}_r^i$ is an implementation for the reader and $\texttt{deactivate}_c^i$ implements the smart card with the implementation of the reader $\texttt{deactivate}_r^i$ as software and the implementation for the smart card $\texttt{deactivate}_c^i$ as hardware.

It might be a good place to add here that both $\texttt{activate}_c^i$ and $\texttt{deactivate}_c^i$ implement one and the same ITL and Tempura construct stable which is easily implemented by non-volatile memory cell since the value there must survive $\neg \texttt{V}_{\texttt{cc}}$, i.e. power down.

- Turning my attention towards `reset`, I see the following in the standard

  - `RST` in low for at least 400 clock cycles;
  - State high on `I/O`, i.e. reception mode;
  - $\texttt{V}_{\texttt{pp}}$ low;
  - `BLK` is stable;

  In addition, the standard says that the `reset` procedure must complete in 40000 clock cycles, i.e. *len* (80000) for the ITL/Tempura specification since a clock cycle takes an interval with length 2 — one for `CLK` and one for $\neg$`CLK`.

This requirement is formalised by the following finite formula.

$$\texttt{reset} \mathrel{\widehat{=}} \boxdot(\neg \texttt{V}_{\texttt{pp}}) \wedge \textsf{stable}\,(\texttt{BLK}) \wedge (Request_{\texttt{reset}}; Response_{\texttt{reset}})$$

where $Request_{\texttt{reset}}$ and $Response_{\texttt{reset}}$ are defined by

$$\begin{aligned} Request_{\texttt{reset}} \mathrel{\widehat{=}} & \exists t \bullet (t \geq 40000 \wedge \textit{len}\,(2t) \wedge \boxdot(\neg \texttt{RST} \wedge \texttt{I/O})); \\ & \exists t \bullet (400 \leq t < 40000 \wedge \textit{len}\,(2t) \wedge \boxdot(\texttt{RST} \wedge \texttt{I/O})) \\ Response_{\texttt{reset}} \mathrel{\widehat{=}} & \boxdot\,\texttt{RST} \wedge (\texttt{L}\,;\texttt{H}\,;\texttt{H}\,;\texttt{L}\,;\textit{len}\,(3etu)\,;\texttt{L}\,;\texttt{L}\,;\texttt{H}) \end{aligned}$$

and where $\texttt{L} \mathrel{\widehat{=}} \textit{len}\,(etu) \wedge \boxdot\,\neg\texttt{I/O}$, $\texttt{H} \mathrel{\widehat{=}} \textit{len}\,(etu) \wedge \boxdot\,\texttt{I/O}$ and $etu \mathrel{\widehat{=}} 744$ stands for Elementary Unit Time as defined by [91]. If I denote $req_{\texttt{reset}} \mathrel{\widehat{=}} (\textit{len}\,(2c') \wedge \boxdot(\neg\texttt{RST} \wedge \texttt{I/O}))\,;(\textit{len}\,(2c'') \wedge \boxdot(\texttt{RST} \wedge \texttt{I/O}))$ where $c' = 40000$ and $c'' = 400$, then I will obviously have $Request_{\texttt{reset}} \sqsubseteq req_{\texttt{reset}}$. I will now need to split the specification for `reset` between the card and the reader and therefore I will need to define the $\texttt{reset}_r \mathrel{\widehat{=}} req_{\texttt{reset}}\,;\textit{len}\,(10etu)$ and $\texttt{reset}_c \mathrel{\widehat{=}} \textit{len}\,(2(c'+c''))\,;Response_{\texttt{reset}}$ parts for the reader and the card correspondingly. Having split the parts between the card and the reader, I will need to compositionally refine `reset` which is obviously refined by the conjunction of the following two assumption-commitment pairs

$$q_1 \supset \texttt{reset} \wedge \textit{fin}\ q_2 \quad \sqsubseteq \quad q_1 \supset \texttt{reset}_r \wedge \texttt{reset}_c \wedge \textit{fin}\ q_2$$

which is further refined by the conjunction of

$$q_1 \wedge Ass_r \wedge \texttt{reset}_r^i \supset \texttt{reset}_r \wedge \mathit{fin}\ q_2$$

and

$$q_1 \wedge Ass_c \wedge \texttt{reset}_c^i \supset \texttt{reset}_c \wedge \mathit{fin}\ q_2$$

where $Ass_r \mathrel{\widehat{=}} \texttt{reset}_c$ is an assumption for the reader and $Ass_c \mathrel{\widehat{=}} \texttt{reset}_r$ is an assumption for the card and in the mean time $\texttt{reset}_r^i$ is an implementation for the reader and $\texttt{reset}_c^i$ implements the smart card with the implementation of the reader $\texttt{reset}_r^i$ as software and the implementation for the smart card $\texttt{reset}_c^i$ as hardware.

- With `not_reset` I have very little difficulties because of the ability of ITL to negate behaviour.

$$\texttt{not\_reset} \mathrel{\widehat{=}} \boxdot_{\texttt{m}} (\neg \texttt{V}_{\texttt{pp}}) \wedge \mathsf{stable}\,(\texttt{BLK}) \wedge (Request_{\texttt{reset}}; FailedResponse_{\texttt{reset}})$$

and where

$$FailedResponse_{\texttt{reset}} \mathrel{\widehat{=}} \boxdot_{\texttt{m}} \texttt{RST} \wedge \neg(\texttt{L}\,;\texttt{H}\,;\texttt{H}\,;\texttt{L}\,;\mathit{len}\,(3etu)\,;\texttt{L}\,;\texttt{L}\,;\texttt{H})$$

and as with reset I can give the definitions for $\texttt{not\_reset}_r \equiv \texttt{reset}_r$, i.e. the reader behaves as for a normal reset, and $\texttt{not\_reset}_c \equiv \mathit{len}\,(2(c' + c''))$;

$FailedResponse_{\texttt{reset}}$, i.e. the card fails to reset and the compositional refinement for the parts of the specification for the card and the reader.

$$q_2 \supset \texttt{not\_reset} \wedge \mathit{fin}\ q_1 \quad \sqsubseteq \quad q_2 \supset \texttt{not\_reset}_r \wedge \texttt{not\_reset}_c \wedge \mathit{fin}\ q_1$$

which is further refined by the conjunction of

$$q_2 \wedge Ass_r \wedge \texttt{not\_reset}_r^i \supset \texttt{not\_reset}_r \wedge \mathit{fin}\ q_1 \tag{7.23}$$

and

$$q_2 \wedge Ass_c \wedge \texttt{not\_reset}_c^i \supset \texttt{not\_reset}_c \wedge \mathit{fin}\ q_1 \tag{7.24}$$

where $Ass_r \mathrel{\widehat{=}} \texttt{not\_reset}_c$ is an assumption for the reader and $Ass_c \mathrel{\widehat{=}} \texttt{not\_reset}_r$ is an assumption for the card and in the mean time $\texttt{not\_reset}_r^i$ is an implementation for the reader and $\texttt{not\_reset}_c^i$ implements the smart card with the implementation of the reader $\texttt{not\_reset}_r^i$ as software and the implementation for the smart card $\texttt{not\_reset}_c^i$ as hardware.

In very similar fashion, I can go through all the commands and states shown on the au-

tomaton in figure 7.3. This will give me a set of equations similar to (7.23) and (7.24). This set will serve as a requirement for the actual implementation, i.e. all formulas of the form $\texttt{reset}_c^i$ and $\texttt{reset}_r^i$ can be developed into real hardware and software to satisfy the requirements of this set.

7.6 The Refinement into Tempura

I will go further and develop only one of the commands that the card should be performing. I will choose the command `encrypt` as a computationally demanding one. This corresponds to the hardware branch of Step 2 from Figure 3.1 on page 27.

Let me see how the specification for that will look like. I know the general form of the specification for a `command` and therefore

$$q_3 \ \supset \ \texttt{encrypt} \wedge \mathit{fin}\ q_3$$

where

$$\texttt{encrypt} \ \widehat{=} \ \boxdot(\neg \texttt{V}_{\texttt{pp}}) \wedge \mathsf{stable}\,(\texttt{BLK}) \wedge$$
$$(ComCode(\texttt{encrypt})\ ;\ ACK\ ;\ (PlainData\ ;\ ACK)^{\text{for 12 bytes}}\ ;\ \texttt{L}^*\ ;\ \texttt{H};$$
$$(EncrData; ACK)^{\text{for 12 bytes}}).$$

Here I assume that $ComCode$ returns the command code for the given instruction, ACK is Acknowledge (in some form or another), $PlainData$ is the data to be encrypted, $EncrData$ is the result of the encryption and it is acknowledged every byte for 12 octets (bytes).

I now need to split this specification between the reader and the card, i.e. I will have to specify $\texttt{encrypt}_c$ and $\texttt{encrypt}_r$ s.t. $\texttt{encrypt}_c \wedge \texttt{encrypt}_r \supset \texttt{encrypt}$. For that reason I will need to specify the mechanism for communication between the two modules. I will build this mechanism from 2 primitives, namely $SndBit$ and $RcvBit$. Single bit can be transmitted via the $\texttt{I/O}$ channel by

$$SndBit(b) \mathrel{\widehat{=}} \textsf{if}\ (b = 0)\ \textsf{then}\ \texttt{L}\ \textsf{else}\ \texttt{H}$$

$$RcvBit(b) \mathrel{\widehat{=}} \textit{len}\ (\frac{etu}{2} - 1)\ ;\ b := \texttt{I/O}\ ;\ \textit{len}\ (\frac{etu}{2})$$

I only have to remember that $RcvBit$ takes a memory variable as an argument. These definitions guarantee

$$SndBit(a) \wedge RcvBit(b) \supset \textit{fin}\ (a = b)$$

and can easily be generalised for a whole byte for example. So, with no loss of generality I will write Snd and Rcv whose arguments will be bytes. Now I can specify the parts for

the card and the reader.

$$\begin{aligned}\texttt{encrypt}_r \mathrel{\widehat{=}} \boxdot(\neg \texttt{V}_{\texttt{pp}}) \wedge (&Snd(\texttt{encrypt})\,;\, Rcv(ACK);\\ &(Snd(PlainData)\,;\, Rcv(ACK))^{\text{for 12 bytes}}\,;\,(\textsf{while } \texttt{L} \textsf{ do } \textit{true});\\ &\textit{len}\,(\texttt{H})\,;\,(Rcv(EncrData); Snd(ACK))^{\text{for 12 bytes}})\end{aligned}$$

and

$$\begin{aligned}\texttt{encrypt}_c \mathrel{\widehat{=}} \textsf{stable}\,(\texttt{BLK}) \wedge (&Rcv(\texttt{encrypt})\,;\, Snd(ACK);\\ &(Rcv(PlainData)\,;\, Snd(ACK))^{\text{for 12 bytes}};\\ &(\texttt{L}^* \wedge \textit{fin}\,(EncrData = PlainData^e \bmod M))\,;\texttt{H};\\ &(Snd(EncrData); Rcv(ACK))^{\text{for 12 bytes}})\end{aligned}$$

where of course I have to make sure that $EncrData$ is the encryption of the $PlainData$, i.e. $EncrData = PlainData^e \bmod M$ is the RSA transformation. Now I can combine the so defined $\texttt{encrypt}_r$ and $\texttt{encrypt}_c$ into the compositional specifications

$$q_3 \wedge Ass_r \wedge \texttt{encrypt}_r^i \supset \texttt{encrypt}_r \wedge \textit{fin}\ q_3 \tag{7.25}$$

and

$$q_3 \wedge Ass_c \wedge \texttt{encrypt}_c^i \supset \texttt{encrypt}_c \wedge \mathit{fin}\ q_3 \tag{7.26}$$

where $Ass_r \mathrel{\widehat{=}} \texttt{encrypt}_c$ is an assumption for the reader and $Ass_c \mathrel{\widehat{=}} \texttt{encrypt}_r$ is an assumption for the card and in the mean time $\texttt{encrypt}_r^i$ is an implementation for the reader and $\texttt{encrypt}_c^i$ implements the smart card.

Going further, I can develop the $\texttt{encrypt}_c^i$ implementation so that it fits in (7.26), i.e. I need to show how exactly I will implement $\texttt{encrypt}_c$ having $Ass_c \mathrel{\widehat{=}} \texttt{encrypt}_r$ in mind as well. It is fairly straightforward to refine stable $(\texttt{BLK})$. All I need to do is declare a memory variable $\mathcal{V}_{\texttt{BLK}}$ with

$$\exists \mathcal{V}_{\texttt{BLK}} \bullet \mathit{true}. \tag{7.27}$$

Next I must refine $Rcv(\texttt{encrypt})$ and $Snd(ACK)$. All I need to do is watch out for the leading LHHL sequence and this is done by

$$\texttt{H}^* \,;\, \texttt{L} \,;\, \texttt{H} \,;\, \texttt{H} \,;\, \texttt{L} \,;\, (Code_{\texttt{encrypt}} \wedge \mathit{len}\,(3etu)) \,;\, \texttt{L} \,;\, \texttt{L} \,;\, \texttt{H} \tag{7.28}$$

and the idea is that if the card does not put the same sequence on the I/O line, then this will result in *false*.

Acknowledgement is easy to put on a line. It is the same as (7.28) but with change in the code. I have

$$\mathtt{H}^* \,;\, \mathtt{L} \,;\, \mathtt{H} \,;\, \mathtt{H} \,;\, \mathtt{L} \,;\, (Code_{ACK} \wedge \mathit{len}\,(3etu)) \,;\, \mathtt{L} \,;\, \mathtt{L} \,;\, \mathtt{H} \tag{7.29}$$

For $Rcv(PlainData)$ I will need to declare a memory variable and I will name it $\mathcal{T}$ for plain text. In addition, $\mathcal{T}$ is needed later in the specification, so it must be a global one. The receive process can be specified by receiving each bit individually and acknowledging the bytes in between. So I have $\mathcal{T}_j$ are the bits and

$$\begin{aligned} &Rcv(\mathcal{T}_0) \,;\, Rcv(\mathcal{T}_1) \,;\, Rcv(\mathcal{T}_2) \,;\, Rcv(\mathcal{T}_3) \,;\, Rcv(\mathcal{T}_4); \\ &Rcv(\mathcal{T}_5) \,;\, Rcv(\mathcal{T}_6) \,;\, Rcv(\mathcal{T}_7) \,;\, Snd(ACK) \end{aligned} \tag{7.30}$$

and I can repeat the same sequence for the rest of the 12 bytes.

I will have to implement the actual encryption now. Here I will reference to the encrypted data as $\mathcal{X}$.

$$\begin{aligned} &\mathcal{X} := 1; \\ &(\mathtt{I/O} = 0 \wedge \mathcal{X} := \mathcal{X} * \mathcal{T} \; mod \; M)^e \end{aligned} \tag{7.31}$$

and all that is left is to send the encrypted data and check for an acknowledgement on the

way, which I already know how to do. Combining all specifications (7.27), (7.28), (7.29), (7.30) and (7.31) into one gives me

(7.32)
$$\begin{aligned}
&\texttt{encrypt}_r^i \mathrel{\widehat{=}} \exists \mathcal{V}_{\texttt{BLK}} \bullet \mathit{true} \wedge \exists \mathcal{T}, \mathcal{X} \bullet \\
&(\texttt{H}^* \,;\, \texttt{L} \,;\, \texttt{H} \,;\, \texttt{H} \,;\, \texttt{L} \,;\, (Code_{\texttt{encrypt}} \wedge \mathit{len}\,(3etu)) \,;\, \texttt{L} \,;\, \texttt{L} \,;\, \texttt{H}; \\
&\texttt{H}^* \,;\, \texttt{L} \,;\, \texttt{H} \,;\, \texttt{H} \,;\, \texttt{L} \,;\, (Code_{ACK} \wedge \mathit{len}\,(3etu)) \,;\, \texttt{L} \,;\, \texttt{L} \,;\, \texttt{H}; \\
&Rcv(\mathcal{T}_0) \,;\, Rcv(\mathcal{T}_1) \,;\, Rcv(\mathcal{T}_2) \,;\, Rcv(\mathcal{T}_3) \,;\, Rcv(\mathcal{T}_4); \\
&Rcv(\mathcal{T}_5) \,;\, Rcv(\mathcal{T}_6) \,;\, Rcv(\mathcal{T}_7) \,;\, Snd(ACK); \\
&\ldots \\
&\mathcal{X} := 1; \\
&(\texttt{I/O} = 0 \wedge \mathcal{X} := \mathcal{X} * \mathcal{T} \; mod \; M)^e \,;\, \texttt{H}; \\
&Snd(\mathcal{X}_0) \,;\, Snd(\mathcal{X}_1) \,;\, Snd(\mathcal{X}_2) \,;\, Snd(\mathcal{X}_3) \,;\, Snd(\mathcal{X}_4); \\
&Snd(\mathcal{X}_5) \,;\, Snd(\mathcal{X}_6) \,;\, Snd(\mathcal{X}_7) \,;\, Rcv(ACK) \,;\, \ldots)
\end{aligned}$$

It is now quite obvious that the specification for $\texttt{encrypt}_r^i$ is deterministic and concrete, i.e. this is a Tempura specification which I can implement in Verilog. The ... in the specification are for the repeated Rcv and Snd statements which are similar to what I have in (7.32).

7.7 The Refinement into Verilog

Finally, I can perform Step 3, or rather the hardware part of it, from Figure 3.1 on page 27, where I give the refinement into Verilog.

There are several important stages in (7.32). The first one is the beginning $\mathtt{H}^*\,;\mathtt{L}$. I can easily show the refinement of $\mathtt{H}^*\,;f$ where $\neg(f \supset \diamondsuit\mathtt{H}^+)$, i.e. f does not start with H. Let P be the following program.

```
module H*;f(I/O,CLK);
  reg i;
 initial begin
  i = 0;
  while (I/O = 1) begin
    i = etu;
    while (I/O = 1 and i > 0) begin
      @(CLK); i = i − 1;
    end
  end
  if (i <= 0) f ;endmodule
```

I will show that if $\mathtt{f}$ is refinement for f in the sense that $f \sqsubseteq \|\mathtt{f}\|$, then I have $\mathtt{H}^* ; f \sqsubseteq \|\mathtt{P}\|$, i.e. $\mathtt{P}$ is refinement of $\mathtt{H}^* ; f$. The translation of $\mathtt{P}$ into $\mathcal{L}_T$ + will be the following.

$$
\begin{array}{l}
\exists Initial.active, \mathcal{T}ime, \mathcal{I}, Disable \bullet \\
\quad clock(Disable) \wedge \Box(Disable = Initial.active) \wedge \mathcal{I} = \bot \\
\quad \{ \\
\quad\quad (\mathcal{I} := 0 \wedge Initial.active = \textit{true}); \\
\quad\quad \textsf{while}\ (\mathtt{I/O} = 1)\ \textsf{do}\ (\\
\quad\quad\quad (\mathcal{I} := etu \wedge Initial.active = \textit{true}); \\
\quad\quad\quad \textsf{while}\ (\mathtt{I/O} = 1 \wedge \mathcal{I} > 0)\ \textsf{do}\ (\\
\quad\quad\quad\quad \textsf{while}\ (\neg \sim (\mathtt{CLK}))\ \textsf{do}\ (Initial.active = \textit{false} \wedge \textsf{skip}); \\
\quad\quad\quad\quad (\mathcal{I} := \mathcal{I} - 1 \wedge Initial.active = \textit{true}); \\
\quad\quad\quad) \\
\quad\quad); \\
\quad\quad \textsf{if}\ (\mathcal{I} <= 0)\ \textsf{then}\ \|\mathtt{f}\|; \\
\quad \}
\end{array}
$$

I can prove $\mathtt{H}^* ; f \sqsubseteq \|\mathtt{P}\|$ by considering two cases. First, I will start with the innermost

while which, assuming $\Box\Diamond \sim (\texttt{CLK})$ — a global assumption guaranteed by (7.22) — gives me

$$\textsf{while}\,(\neg \sim (\texttt{CLK}))\ \textsf{do}\ (Initial.active = \textit{false} \wedge \textsf{skip})\ \supset\ \textsf{stable}\,(\texttt{CLK}).$$

Turning my attention to the middle while , it is quite obvious that

$$\begin{array}{l}
(\mathcal{I} := etu \wedge Initial.active = \textit{true});\\
\textsf{while}\,(\texttt{I/O} = 1 \wedge \mathcal{I} > 0)\ \textsf{do}\ (\\
\quad \textsf{stable}\,(\texttt{CLK});\\
\quad (\mathcal{I} := \mathcal{I} - 1 \wedge Initial.active = \textit{true});\\
)\\
\quad \supset\\
\textit{finite} \wedge (\texttt{I/O} = 1 \wedge \mathcal{I} > 0 \wedge \textsf{stable}\,(\texttt{CLK}))^n \wedge\\
\textit{fin}\,(\texttt{I/O} = 0 \vee \mathcal{I} <= 0) \wedge n \leq etu \wedge (n < etu \equiv \mathcal{I} > 0)\\
\quad \supset\\
n = etu\ \supset\ \texttt{H}
\end{array}$$

for some n. The last implication uses the assumption $Ass_c \mathrel{\widehat{=}} \texttt{encrypt}_r$ where we

have embedded the behaviour for an H. The important part of the last formula is that if I have less than etu number of repetitions for $(\mathtt{I/O} = 1 \wedge \mathcal{I} > 0 \wedge \mathsf{stable}\,(\mathtt{CLK}))$, then $\mathcal{I} > 0$. This obviously would have meant that there was a partial H on the line, i.e. the behaviour of the reader is unexpected.

The outer while is now fairly simple. I have there

$$\mathsf{while}\,(\mathtt{I/O} = 1)\ \mathsf{do}\ \mathtt{H}\ \supset\ \mathtt{H}^*$$

and with the last check for complete H all the way through, i.e. $\mathsf{if}\,(\mathcal{I} <= 0)\ \mathsf{then}\ \|\mathtt{f}\,\|$ I can get the desired $\mathtt{H}^*\,;\,f$.

On the way, I have shown the refinement of L and H which is all based on

$$\begin{array}{l}\mathtt{i} = \mathtt{etu}\,;\\ \mathtt{while}\ (\mathtt{I/O} = \mathtt{bit}\ \mathtt{and}\ \mathtt{i} > 0)\ \mathtt{begin}\\ \quad @(\mathtt{CLK})\,;\ \mathtt{i} = \mathtt{i} - 1\,;\\ \mathtt{end}\end{array}$$

where bit is 0 or 1 for the appropriate cases.

The refinement for Snd and Rcv is even simpler. I only need to remember that

$$SndBit(b) \mathrel{\widehat{=}} \mathsf{if}\,(b = 0)\ \mathsf{then}\ \mathtt{L}\ \mathsf{else}\ \mathtt{H}$$

and in this case the refinement of L and H will be as follows

$$\texttt{i} = \texttt{etu}\ ;\texttt{I/O} := \texttt{b}\ ;$$

$$\texttt{while}\ (\texttt{i} > 0)\ \texttt{begin}\ \ @(\texttt{CLK})\ ;\ \texttt{i} = \texttt{i} - 1\ ;\texttt{end}$$

where b will be 0 or 1 for the appropriate case. The difference in the refinement here is that in the previous case I had to check if the opposite side, in my case the reader, kept the I/O line in 0 or 1 for the designated time of etu. Here I implicitly assign this value to it.

The refinement of Rcv, where

$$RcvBit(b) \mathrel{\widehat{=}} \mathit{len}\,(\frac{etu}{2} - 1)\ ;b := \texttt{I/O}\ ;\mathit{len}\,(\frac{etu}{2})$$

is as follows

$$\texttt{i} = \texttt{etu/2} - 1\ ;$$

$$\texttt{while}\ (\texttt{i} > 0)\ \texttt{begin}\ \ @(\texttt{CLK})\ ;\ \texttt{i} = \texttt{i} - 1\ ;\ \texttt{end}$$

$$\texttt{b} := \texttt{I/O}\ ;\texttt{i} = \texttt{etu/2}\ ;$$

$$\texttt{while}\ (\texttt{i} > 0)\ \texttt{begin}\ \ @(\texttt{CLK})\ ;\ \texttt{i} = \texttt{i} - 1\ ;\ \texttt{end}$$

and the proof that this is the refinement can be easily derived from the proof for $\texttt{H}^*\ ;\ f$ above.

The only part of (7.32) that still needs implementation in Verilog is the part where the actual encryption is performed, i.e. (7.31). However, this is an obvious while construct since the f^e form and poses no difficulties in the refinement. Namely I have

$$\texttt{x} = \texttt{1} \ ; \texttt{i} = \texttt{e} \ ;$$

$$\texttt{while} \ (\texttt{i} > \texttt{0}) \ \texttt{begin} \ \ \texttt{I/O} = \texttt{0} \ ; \ \texttt{x} = \texttt{x} * \texttt{T} \ \texttt{mod} \ \texttt{M} \ ; \ \texttt{end}$$

for the refinement of the actual encryption.

7.8 Summary

I believe the case study in this chapter is industrially relevant. The development process goes through all major steps of my initial methodology for codesign as stated in chapter 3 and figure 3.1 and I prove properties of interest about the system at every point of the refinement. The sheer size of the case study prevented me from pursuing the final result in its completeness. However, I hope the reader will be convinced that the practicality of my approach has been successfully demonstrated.

Chapter 8

Conclusion

8.1 Vision

I started this project with a vision for a *compositional* methodology which would allow me to blend software and hardware in a seamless way. I also aimed for a methodology with built-in rigorous reasoning about the design process and the properties of the required system.

The benefits of such an approach could be viewed in two ways. The compositional theory [88, 35, 19, 17, 65, 63] states that I would be able to reason about a system, or any of its subsystems, within its context. Thus I can guarantee:

1. a system that will co-operate with its intended environment and will not be a closed component but rather an open one,

2. a black-box abstraction where I view a system, or any of its subsystems, as a pair of an interface and a behaviour,

3. the ability to infer properties about the system from the properties of its subsystems and

4. the ability to derive requirements for the subsystems from the requirements towards the system as a whole.

The first two have a profound effect on the usability of the designed system. The view that a system should be nothing more than a pair of an interface and a behaviour allows me to say that any environment that falls within the assumptions of the interface and the behaviour will be a suitable match for the system; therefore I will simply "connect" it with the environment and "plug-and-play" with it. Also, the coupling of the environment and the system enforces a reactive computational model upon me, since the communication (albeit synchronous or asynchronous) must be based on messages and/or events. Therefore, the underlying computational model has to come as a reaction to the messages and/or events exchanged between the environment and the system.

The last two allow me to incorporate two types of design, namely top-down and bottom-up design, in a systematic way. Thus, if I can infer properties of the system as a whole from the properties of its components; then I can construct larger systems out of smaller parts and this facilitates reuse, bottom-up design and backwards engineering. On the other hand, if I can derive the requirements for the components from the requirements

towards the whole system, then I enable top-down design and forward engineering.

It is interesting to note here that the synthesis between the bottom-up and top-down designs gives me another possible alley for exploration — the abstraction of requirements from an existing system, combined with further re-development — and this results in re-engineering and migration.

The other important aspect of my vision was the rigorous reasoning about the design process and the properties of the designed system. It has to be said that our expectations towards the stability, reliability and correctness of the systems we use are constantly increasing. More and more activities now depend on the correct performance of a computer system of some sort and we seem to have little tolerance for crashes, incorrect or unexpected behaviour, down-time and unavailability. I can find this level of expectation towards the system not only in safety-critical applications, where lives could be at risk, but also in business-critical environments, where the survival of a whole organisation might depend on the correct and expected performance of the underlying computer infrastructure.

My second fundamental vision was for a methodology that would allow me to specify a system in a very abstract way regardless of its intended target implementation. I believe that starting from a highly abstract representation of the system allows me to blur the differences between hardware and software. At the beginning I am interested in the desired behaviour only and this makes no distinction between technologies, architectures, communication paradigms, etc. Although this may seem insensitive, it gives me a chance

to think about the system in terms of behaviour, interface, requirement, components, etc., rather than programming or hardware description language, source code or net-list size, involved software or hardware technologies, etc. Only at a later stage of the development I start taking into consideration other important issues such as the issue of underlying architecture.

8.2 Achievement

In chapter 3 I clearly state the overall methodology for co-design as depicted in figure 3.1 on page 27. I start from a high-level and abstract specification and then I progress with the development through *correctness preserving* refinement steps.

Throughout the whole project, my main high-level specification language and mechanism for reasoning has been the Interval Temporal Logic (ITL). This has been the bedrock of my methodology and I have tried to relate all my reasoning to that.

Once I capture the desired properties of the system with an ITL formula I can prove that they are not conflicting by using the Tempura tool. When I am satisfied with the level of correctness with which I have captured the requirements I can then start developing the system by using the set of compositional refinement laws presented in section 3.6 pages 48 onwards. The desired result is a deterministic representation of the system in Tempura, which is an executable subset of ITL.

At this point I reason about the architecture that is most suited for the application and

I obtain a set of Tempura modules. Again I can use the tools within the Tempura tool to simulate and verify properties about the obtained Tempura code. In section 3.6.2 I explain how I can simulate and analyse Tempura code within the Tempura tool. A screen dump of the tool is shown in figure 8.1 on page 185.

The next step of the methodology is to select which Tempura modules will be implemented in hardware and which in software. Only at this stage do I consider technology related issues and I can use a multitude of techniques for deciding the hardware-software split [66, 47, 48, 55, 96]. Here I can also discuss communication issues between the different modules and compositionally verify them all thanks to the *assumption-commitment* style refinement rules of ITL.

This is now the stage within my methodology when I have to take my level of abstraction to real hardware and software, since I have to refine the hardware and the software modules separately. As it has already been shown, the refinement to software [13, 88, 14, 16, 86] is achievable. Therefore I had to concentrate my efforts into finding a way to refine the Tempura specifications into my hardware description language (HDL) of choice. This is where I realise the need for ITL based semantics for Verilog.

Once I have the abstraction gap between Tempura and Verilog bridged, I can then use the existing Register Transfer Layer (RTL) to netlist synthesis tools and technologies via commercially available synthesisers to achieve real hardware as on FPGA or ASIC.

However, there was a theoretical barrier I had to overcome in the process. The problem was that the basic syntax and semantics for ITL and Tempura do not have *memory*

variables as explained in section 3.4.3 (pages 41 onwards). As I show there, I can conservatively introduce memory variables and I can even prove in theorem 1 on page 43 that they have the basic "memory" property of keeping their values until explicit assignment operators change them.

What followed as a result was an ITL based semantics for Verilog in chapter 4. There I consider both Behavioural and RTL statements, thus bridging yet another abstraction level gap within Verilog itself. Unlike most of the other approaches [90, 89, 67, 30], mine deals with a rich core of the language, whereas only some `case` like convenient but non-essential constructs are left out.

Again in contrast to other attempts for the semantics of Verilog [94], where different semantical models are constructed for the different abstraction levels within the language, I achieve a single ITL based formalism throughout my work and I believe this facilitates the refinement in a better way.

Admittedly, Verilog lacks a well accepted formal semantics and therefore I felt the need for a second perspective on the language. Two of the main styles of semantics are denotational and operational. Since ITL provided me with denotational semantics for Verilog, I decided to construct an operational semantics for Verilog as well. Unlike most of the commercially available simulators, my operational semantics is *fully parallel*.

In chapter 5 I also formulate and prove several healthiness conditions on the operational semantics through theorems 1, 2 and 3 which I believe should be necessary for any formalism defining the operational semantics for Verilog, being a simulator or not. At

the end of the chapter I show how the operational semantics can be used for simulating a program in Verilog, thus I show that the semantics is a blueprint for a simulator for the language.

It was logical, after defining two different formalisms about the semantics for Verilog, to prove that they are equivalent in the sense that the behaviour described by the denotational semantics is precisely the one generated by the operational semantics. This would guarantee the uniformity and boost the confidence in the trustworthiness of my work. The outline of the proof is given in section 6.2 and the full details are spelled out in section 6.3.

The final chapter of my exposé introduces an industrially relevant case study of a smart card application. It involves RSA asymmetrical encryption and decryption on the smart card chip itself. The rationale for this is that the smart card can protect the private key best and therefore is the perfect candidate for such an application. I show there how I can formalise and structurise the problem by using the refinement laws given in section 3.6. This allows me to achieve a high level of modularity.

It was obvious to me that the case study was a formidable project on its own and therefore I used it as a "proof-of-concept" only. I have given the refinement to only a small part of the whole specification. However, I believe that the principles throughout the case study are clear and applicable in industrially sized applications as well. I also show there how I can use my ITL based semantics to refine a Tempura specification into a program in Verilog.

8.3 Related Work

Although Hardware/Software Codesign is a young discipline, with the earliest reference dated 1992 at the First International Workshop on Hardware/Software Codesign [95], there is a growing body of work devoted to the topic.

Figure 2.1 gives a typical design flow widely adopted as an initial idea. It quickly became clear that this design flow has many practical problems inherited from the very early design decision taken in accordance to it. For example, communication paradigms, architecture and Hardware/Software split within the system are chosen early in the design process without any validation of their suitability. Almost all of these design decisions rely mostly on intuition and best practices rather than rigorous reasoning.

Another problem with the early approach could be described as "late integration" syndrome, i.e. the system integration occurs only after all sub-systems have been developed to a considerable extend. This is considered very late in the design cycle, because only at this late stage the designer can validate and justify the design decisions taken at the very beginning of the development. This problem is particularly important because it magnifies the difficulty and the complexity of a project with *changing* system requirements.

One major thread within the Hardware/Software Codesign research and development has been the idea of *co-simulation* [77, 2, 23]. This approach tries (systematically and/or heuristically) to break the problem into smaller parts (sub-problems, tasks, computation entities, basic scheduling blocks, etc.) and to allocate each part into software or hardware.

The core objective is to find the best configuration, with respect to speed of execution and communication, between the software code and the hardware ASIC/FPGA implementation. In effect, we can view the problem as granularity and optimisation [47, 48, 55, 96].

The architecture selection is an important issue within the Hardware/Software Codesign. Generally we can assume that it is a mapping process from system's functionality to a set of (predefined) components, i.e. targeted architecture. Using this technique, successful automation has been achieved in applications involving a memory hierarchy or an I/O subsystem design based on standard components. In addition to that, there are some alternative approaches on retargetable compilation [79], or on an abstract partitioning for co-design [42, 43, 69, 81].

Temporal and Spatial Partitioning as shown in figure 2.4 on page 22 is another important and interesting area of research. It achieves flexibility which, however, does not come for free. A scheduler is a major part and it decides "on the fly" how to partition the code into compiled program for a microprocessor and which will be used to reconfigure the FPGA. The particularly interesting area of research in reconfigurable processors [52] implements this idea.

Several different styles of semantics for Verilog [67, 30, 90, 89] have been proposed. However, the complexity of the language proves a difficult challenge for some and they consider subsets of the language, while others choose to use several different semantics for different levels of abstraction within the language.

My work contributes to the body of research in Hardware/Software Codesign in sev-

eral ways:

1. I develop a unifying and compositional framework for Codesign based on pairs of assumptions and commitments.

2. I propose stepwise formal refinement as a sound tool for the correct development of mixed hardware/software systems.

3. I use Interval Temporal Logic (ITL) as a bedrock for my reasoning.

4. I integrate simulation in my framework through the existing Tempura tool.

5. I construct a denotational semantics for Verilog which is then used to define the refinement relation between ITL, Behavioural Verilog and RTL Verilog in a formal manner.

6. I construct an operational semantics for Verilog, define and prove several healthiness conditions and show full parallelism for it. It can serve as a blueprint for a real simulator for Verilog whose full parallelism and formal underpinning would be unique.

7. I prove equivalence between our denotational and operational semantics for the language.

8. I test my theory on an industrially relevant case study. RSA asymmetrical encryption is performed on a smart card chip for improved security.

8.4 Future Work

I can see two major possible expansions of this work — a theoretical and a practical one.

The careful reader would notice that the operational semantics as presented here can capture a much bigger set of constructs. The obvious non-conservative extension could be sentences which combine Behavioural and RTL constructs sequentially, rather than just in parallel as it is within the language now. Currently Verilog does not have constructs like

$$\texttt{initial}\ statement\ ;\ \texttt{assign}$$

that one would like to interpret as an atom which sets some variables initially and then proceeds with the RTL behaviour of the `assign` statement.

A more interesting combination of this approach would have been

$$\texttt{assign}\ ;\ \texttt{initial}\ statement$$

where the possibly infinite behaviour of `assign` is followed by another statement. One can use this combination to express *fault tolerance and recovery*, i.e. if the `assign` fails, then the `initial` $statement$ will take care of the consequences.

Another important issue is a possible *algebraic* semantics for Verilog. We will be able

to define an algebraic equation of the following kind

$$\mathtt{V}_1 = \mathtt{V}_2 ,$$

where $\mathtt{V}_1$ and $\mathtt{V}_2$ are two Verilog constructs, by proving that their denotational semantical meanings are equivalent, i.e.

$$\|\mathtt{V}_1\| \equiv \|\mathtt{V}_2\|$$

or by proving that the operational runs generated by the two constructs are equivalent. One can view the latter as *bi-simulation* between the two sentences.

An algebraic semantics of this form could be used for optimisation, since the algebraic style semantics involves a system of equations, i.e. equivalences, therefore we can use such semantics for equivalently transforming a Verilog program into a "better" form. Here "better" might mean "faster", "smaller" or "cheaper" implementation in real hardware.

A practical set of refinement laws to guide our developer-centred methodology is also desirable. The denotational semantics for Verilog will serve as a definitive criteria for the soundness of each refinement law as stated by (4.1). However, we will need to develop many more case studies and try our theory on them before claiming "practicality".

There could be some practical work done to extend this project. As shown, our operational semantics is, in effect, a simulator for the language. Therefore, we can attempt to develop such a simulator. The *full parallelism* would have made it unique among all other

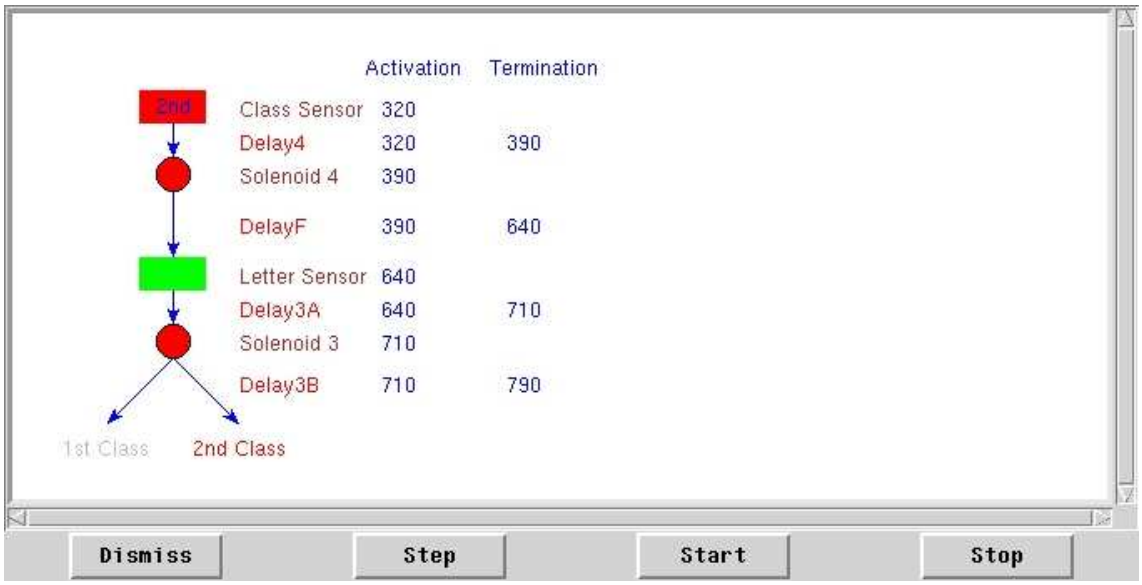


Figure 8.1: Tempura

Verilog simulators available. In addition to that, we could integrate the simulator with Tempura (see figures 8.1 above and 3.2 on page 52) which can then test how the simulator will compare with other known simulators.

References

[1] IEEE, *IEEE Standard for Verilog Hardware Description Language*, IEEE, 1364-2001, 2001.

[2] Allara A., Brandolese C., Fornaciari W., Salice F. and Sciuto D., "System-Level Performance Estimation Strategy for Sw and Hw", in *Proceedings of the International Conference on Computer Design 1998*, IEEE, 1998.

[3] Balarin F., "Hardware-Software Co-Design of Embedded Systems: The Polis Approach", Kluwer Academic Publishers, 1997.

[4] Beer I., "The Temporal Logic Sugar", in *Proceedings of CAV 2001*, LNCS, Springer-Verlag, 2001.

[5] Beneviste, A. and Berry G., "The Synchronous Approach to Reactive and Real-Rime Systems", *Proceedings of IEEE,* 79(9), pp. 1268-1336, 1991.

[6] Beth E., *The Foundation of Mathematics*, second edition, 1965.

[7] Börger E. and Del Castillo G., “A Formal Method for Provably Correct Composition of a Real-Life Processor Out of Basic Components”, in *Proceedings of the 1st ICECCS’95*, IEEE Computer Society Press, 1995.

[8] Borrione D., “A Functional Approach to Formal Hardware Verification”, in *Proc. of ICCD-88*, 1988.

[9] Brigham E., *The Fast Fourier Transform*, Prentice Hall, 1974.

[10] Burgelman R., Carter D. and Bamford R., “Intel Corporation: The Evolution of an Adaptive Organization”
`http://www.aom.pace.edu/meetings/1999/INTEL.htm`

[11] Castillo G. and Hardt W., “Towards a Unified Analysis Methodology of HW/SW Systems Based on Abstract State Machines: Modelling of Instruction Sets”, in *Proceedings of the GI/ITG/GMM Workshop*, Paderborn, 1998.

[12] Cau A., Hale R., Dimitrov J., Zedan H., Moszkowski B., Manjunathaiah M. and Spivey M., “A Compositional Framework for Hardware/Software Co-Design”, in Camposano R., Wolf W. (eds.) *Design Automation for Embedded Systems*, Kluwer, 2002.

[13] Cau A. and Zedan H., “Refining Interval Temporal Logic specifications”, in Eds. Bertran M., Rus T. *Transformation-Based Reactive Systems Development*, LNCS Vol. 1231, pp. 79–94, Springer 1997.

[14] Cau A. and Zedan H., "The Systematic Construction of Information Systems", in Henderson P. (ed.) *Systems Engineering for Business Process Change*, Springer Verlag, 2000.

[15] Chaochen Z., Hoare C.A.R. and Ravn A., " A Calculus of Durations", *Information Processing Letters*, Vol 40, No 5, pp. 269 – 276, December 1991.

[16] Chen Z., Zedan H., Cau A. and Yang H., "A Wide-Spectrum Language for Object-Based Development of Real-time Systems", *International Journal of Information Sciences*, Vol 118, pp. 15-35, 1999.

[17] Clarke E., "Compositional Model Checking", in *Proc. Workshop on Automatic Verification Methods for Finite State Systems* ed. Sifakis J., LNCS 407, Springer Verlag, 1989.

[18] Comer D., *Internetworking with TCP/IP*, Vol. 1 and 2, Englewood Cliffs, London, Prentice Hall, 1991.

[19] de Roever W., de Boer F., Hannemann U., Hooman J., Lakhnech Y., Poel M. and Zwiers J., *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, Cambridge University Press, 2001.

[20] Dimitrov J., "Compositional Reasoning about Events in Interval Temporal Logic", in *Proceedings of JCIS 2000*, pp. 675-678, Association for Intelligent Machinery, Atlantic City, Feb–Mar 2000.

[21] Dimitrov J., “Interval Temporal Logic (ITL) Semantics for Verilog”, *IEE event on Hardware-Software Co-Design*, IEE, London, 8th December 2000.

[22] Dimitrov J., “Operational Semantics for Verilog”, in *Proceedings of APSEC 2001*, pp. 161–168, IEEE, Macau, Dec 2001.

[23] Dreike P. and McCoy J., “Cosimulating Hardware and Software in Embedded Systems”, in *Proceedings Embedded Systems Programming Europe*, IEEE Computer Society Press, 1997.

[24] Eles P., “Codesign of Embedded Systems: Where are we now”, *IEE event on Hardware-Software Co-Design*, IEE, London, 8th December 2000.

[25] Edwards S., Lavagno L., Lee E. and Sangiovanni-Vincentelli A., “Design of Embedded Systems: Formal Models, Validation, and Synthesis”, *IEEE Proc*, Vol. 85, No. 3, March 1997, pp. 366–390.

[26] Ernst R., “Codesign of Embedded Systems: Status and Trends”, *IEEE Design & Test of Computers*, pp. 45–54, 1998.

[27] Gajski D., Zhu J. and Domer R., “Essential Issues in Codesign”, Technical Report ICS-TR-97-26, University of California, 1997.

[28] Giridhar P., Kumar V. and Mathai J., “The Mine Pump Control Program in Esterel”, Research Report CS-RR-332, Department of Computer Science, University

of Warwick, 1997. A full bibliography and copy of this paper is available through `http://www.dcs.warwick.ac.uk/pub/reports/rr/332.html`

[29] Golze U., *VLSI Chip Design with the Hardware Description Language Verilog*, Springer-Verlag, Berlin, 1996.

[30] Gordon M., "The Semantic Challenge of Verilog HDL" in *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pp. 136–145, San Diego, California, 26–29 June 1995.

[31] Gordon M. and Melham T., *Introduction to HOL: A theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.

[32] Hale R., *Programming in Temporal Logic*, PhD dissertation, Cambridge University 1988.

[33] Hansen M. and Chaochen Z., "Semantics and Completeness of DC" in de Bakker J., Huizing C., de Roever W., Rozenberg G. (editors) *Real-time: theory in practice*, Lecture notes in computer science Vol. 600, June 1991, pp. 209–225.

[34] Hoare C.A.R. and Jifeng H., *Unifying Theories of Programming*, Prentice Hall, 1998.

[35] Henzinger T., Qadeer S. and Rajamani S., "Decomposing Refinement Proofs Using Assume-Guarantee Reasoning", In Int. Conf. *Computer-aided Design*, pp. 245–252, IEEE Computer Society Press, 2000.

[36] Henzinger T., *The Temporal Specification and Verification of Real Time Systems*, PhD thesis, Stanford University, A full copy of this work can be found at `http://www-cad.eecs.berkeley.edu/~tah/Publications/`

[37] Hollander Y., Morley M. and Noy A., "The *e* Language: A Fresh Separation of Concerns", in *Proceedings of TOOLS Europe 2001*, IEEE Computer Society Press, 2001.

[38] Holzmann G., *Design and Validation of Computer Protocols*, Prentice-Hall, 1990.

[39] Hong I., Kirovski D., Qu G., Potkonjak M. and Srivastava B., "Power Optimization of Variable Voltage Core-Based Systems", in *Proceedings of DAC*, pp. 176–181, June 1998.

[40] Imperato M., *An Introduction to Z*, Chartwell-Bratt, 1991.

[41] Jifeng H., Page I. and Bowen J., "Towards a Provably Correct Hardware Implementation of Occam", in Milne G.J., Pierre L. (ed) *Correct Hardware Design and Verification Methods*, Proc. IFIP WG10.2 Advanced Research Working Conference, CHARME'93, LNCS 683, Springer-Verlag, 1993.

[42] Kumar S., Aylor J., Johnson B. and Wulf W., "A Framework for Hardware/Software Codesign", in *Proc. of the Int. Workshop on Hardware-Software Codesign*, Sept. 1992.

[43] Kumar S., Aylor J., Johnson B. and Wulf W., "Exploring Hardware/Software Abstractions and Alternatives for Codesign", in *Proc. of the Int. Workshop on Hardware-Software Codesign*, Oct. 1993.

[44] Hunt W., "FM8501: A Verified Microprocessor", in *Proc of IFIP WG 10.2 Workshop: From HDL To Guaranteed Correct Circuit Designs*, 1986.

[45] Karkowski I., "Computer Aided Embedded Systems Design", in *Proceedings of the Third Annual Conf. of ASCI*, Heijen, The Netherlands, 2–4 June 1997.

[46] Kleinjohann B., Tacken J. and Tahedl C., "Towards a Complete Design Method for Embedded Systems Using Predicate/Transition-Nets", in *Proceedings of the XIII IFIP WG 10.5 Conference on Computer Hardware Description Languages and Their Applictations (CHDL-97)*, pp. 4–23, Toledo, Spain, April 1997.

[47] Kundsen P. and Madsen J., "Communication Estimation for Hardware/Software Codesign", *6th International Workshop on Hardware/Software Codesign, Codes/CASHE'98.*

[48] Kundsen P. and Madsen J., "PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning", *4th International Workshop on Hardware/Software Codesign, Codes/CASHE'96.*

[49] Kurshan R., *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton University Press, 1994.

[50] Lavenier D., Quinton P. and Rajopadhye S, "Chapter 5, Digital Signal Processing for MultiMedia Systems", in Eds. Parhi, Hishitani *Advanced Systolic Design*, 1999

[51] Leupers R. and Marwedel P., "Retargetable Code Generation Based on Structural Processor Descriptions", *Design Automation for Embedded Systems*, vol. 3, no. 1, 1998.

[52] Li Y., "Hardware-Software Co-Design of Embedded Reconfigurable Architectures" *DAC 2000*, p. 507, Los Angeles, USA.

[53] Liu Z., Ravn A. and Li X., "Compositional Verification of Duration Properties of Real-Time Systems", Technical report No. 1997/30, Department of mathematics & computer science, University of Leicester.

[54] Lüettgen G. and Mendler M., "Towards a Model-Theory for Esterel", in *Proceedings of SLAP*, 2002.

[55] Madsen J., "LYCOS: the Lyngby Co-Synthesis System", in *Design Automation for Embedded Systems*, Volume 2, Issue 2, pp. 195–235, Kluwer Academic Publishers, March 1997.

[56] Manna Z. and Pnueli A., *The Temporal Logic of Reactive and Concurrent Systems: Specification*, 1991

[57] Mathai J., *Real-Time Systems: Specification, Verification and Analysis*, 1996.

[58] Megson G.M., *An Introduction to Systolic Algorithm Design*, Oxford University Press, 1992.

[59] Mosses P., "Foundation of Modular SOS", in Kutylowski, Pacholski, Wierzbicki ed. *Mathematical foundations of computer science: 24th international symposium*, LNCS 1672, 1999.

[60] Moszkowski B., "A Complete Axiomatization of Interval Temporal Logic with Infinite Time", in *proc. of the Fifteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2000)*, IEEE Computer Society Press, June 26-29, 2000, Santa Barbara, California, USA, pp. 242–251.

[61] Moszkowski B., "A Temporal Logic for Multilevel Reasoning about Hardware", *IEEE Computer*, Vol. 18, no. 2, 1985, pp. 10-19.

[62] Moszkowski B., "An Automata-Theoretic Completeness Proof for Interval Temporal Logic", eds Montanari U., Rolim J, Welzl E., in *Lecture Notes in Computer Science*, 1853, Springer-Verlag. Geneva, Switzerland, July 9–15, 2000, pp. 223–234.

[63] Moszkowski B., "Compositional Reasoning about Projected and Infinite time" in *Proc. First IEEE Int'l Conf. on Engineering of Complex Computer Systems (ICECCS'95)*, IEEE Computer Society Press, 1995, pp. 238–245.

[64] Moszkowski B., *Executing Temporal Logic Programs*, Cambridge University Press, 1986.

[65] Moszkowski B., “Some Very Compositional Temporal Properties” in Olderlog E. (ed.) *Programming concepts, Methods and Calculi*, IFIP Transactions, Vol. A-56, North-Holland 1994, pp. 307–326.

[66] Nielson F., Nielson H.R. and Hankin C., *Principles of Program Analysis*, Springer-Verlag, 1999.

[67] Pace G., “The Semantics of Verilog Using Transition System Combinators”, in Proc. *Formal Methods in Computer-Aided Design*, 2000.

[68] Plotkin G., *A Structural Approach to Operational Semantics* Lecture notes DAIMI FN-19, 1981

[69] Prakash S. and Parker A., “Synthesis of Application-Specific Multi-Processor Architectures”, in *Proc. of the Design Automation Conf*, June 1991.

[70] Ravn A., Rischel H. and Hansen K., “Specifying and Verifying Requirements for Real-Time Systems”, *IEEE Transactions on Software Engineering*, Vol 19, No 1, pp. 41 – 55, January 1993.

[71] Sagdeo V., *The Complete Verilog Book*, Kluwer Academic Publishers, 1998.

[72] Sampat N., Zedan H. and O’Callaghan A., “From Business Needs to Software Solutions: Comparing Use Case Driven Approaches for Component Based Development”, in *Proceedings of The Fifth International Conference on Computer Science and Informatics* (CS&I’2000), 2000.

[73] Sander I. and Jantsch A., “Formal System Design Based on the Synchrony Hypothesis, Functional Models, and Skeletons”, in *Proceedings of the IEEE International Conference on VLSI Design*, 1999.

[74] Schrott G. and Tempelmeier T., “Putting Hardware-Software Codesign into Practice”, in *Preprints of the 22th IFAC/IFIP Workshop on Real Time Programming*, Lyon, France, Sept 15–17, 1997.

[75] Shelfer K. and Procaccino J., “Smart Card Evolution”, *Communications of ACM*, Vol. 45, No. 7, July 2002.

[76] Spivey M. and Page I., *How to Program in Handel*, Technical report, see `http://www.comlab.ox.ac.uk/oucl/hwcomp.html`, Oxford University Computing Laboratory, 1993.

[77] Soininen J., Huttunen T., Tiensyrja K. and Heusala H., “Cosimulation of Real-Time Control Systems”, in *Proceedings of the European Design Automation Conference with EURO-VHDL ’95*, IEEE Computer Society Press, 1996.

[78] Sorensen E., Revn A. and Rischel H., *Control Program for a Gas Burner: Part 1: Informal Requirements, ProCoS Case Study 1*, ProCoS Report ID/DTHEVS2, 1990.

[79] Theissinger M., Stravers P., Veit H. “CASTLE: an Interactive Environment for Hardware-Software Co-design”, in *Proc. of the Int. Workshop on Hardware-Software Codesign*, 1994.

[80] Thomas D. and Moorby P., *The Verilog Hardware Description Language*, Kluwer Academic Publishers, ISBN 0792381661, May 1998.

[81] Vahid F. and Gajski D., "Specification Partitioning for System Design", in *Proc. of the Design Automation Conf*, June 1992.

[82] Verkest D., "Matisse: A System-on-Chip Design Methodology Emphasizing Dynamic Memory Management", *Journal of VLSI Signal Processing*, 21(3): 277–291, July 1999

[83] Wingard D., "MicroNetwork-Based Integration of SOCs", in *Proceedings of the 38th Design Automation Conference*, June 2001.

[84] Yang H., Liu X. and Zedan H., "Abstraction: A Key Notion for Reverse Engineering in A System Reengineering Approach", *Journal of Software Maintenance: Research and Practice*, 12(5):197-228, 2000.

[85] Zedan H. and Cau A., "A Logic-Based Approach for Hardware/Software Co-design", *IEE event on Hardware-Software Co-Design*, IEE, London, 8th December 2000.

[86] Zedan H., Cau A., Chen Z. and Yang H., "ATOM: An Object-based Formal Method for Real-time Systems" *Annals of Software Engineering*, Vol. 7, 1999.

[87] Zhang Y., *A Foundation for the Design and Analysis of Robotic Systems and Behaviours*, PhD thesis, University of British Columbia.

[88] Zhou S., Zedan H. and Cau A., "A Framework For Analysing The Effect of 'Change' In Legacy Code", in *IEEE Proc. of ICSM'99*, 1999.

[89] Zhu H., Bowen J. and Jifeng H., "From Operational Semantics to Denotational Semantics for Verilog" In *Proceedings, 11th CHARME*, 2001.

[90] Zhu H. and Jifeng H., *A DC-Based Semantics for Verilog* UNU/IIST Report 183.

[91] ISO/IEC 7816, This set of standards can be purchased from ISO at `http://www.iso.ch/` A short and partial description of these standards can be found at `http://www.scia.org/knowledgebase/aboutsmartcards/iso7816_wimages.htm`.

[92] RSA Laboratories, *PKCS #1: RSA Encryption Standard*, 1991—1993, `ftp://ftp.rsa.com/pub/pkcs/pkcs-1/pkcs-1v2-1d1.ps`

[93] `http://www.cse.dmu.ac.uk/~cau/itlhomepage/index.html`

[94] `http://www.cl.cam.ac.uk/users/djs1002/verilog.project/`

[95] `http://www.ece.uci.edu/~codes/`

[96] `http://www.it.dtu.dk/~lycos/`

[97] `http://www.uilondon.org/cherntim.html`