Vérification de descriptions VHDL par interprétation abstraite.

Charles Hymans

Thèse

présentée à

l'École Polytechnique

Pour l'obtention du titre de

Docteur En Sciences de l'École Polytechnique

Discipline
Mathématiques et Informatique

par

Charles Hymans

le 9 septembre 2004

Vérification de composants VHDL par interprétation abstraite

Verification of VHDL descriptions by abstract interpretation

Président	Reinhard Wilhelm *Professeur, Universität des Saarlandes, Allemagne*
Rapporteurs	Kenneth L. McMillan *Senior Research Scientist, Cadence Berkeley Labs, États Unis*
	Kwangkeun Yi *Professeur, Seoul National University, Corée du Sud*
Examinateurs	Gilles Dowek *Professeur, École Polytechnique*
	Manuel Hermenegildo *Professeur, Technical University of Madrid, Espagne,* *& University of New Mexico, États Unis*
Directeur de thèse	Radhia Cousot *Directeur de recherche, CNRS*

Hâtez-vous lentement ; et, sans perdre courage,
Vingt fois sur le métier remettez votre ouvrage :
Polissez-le sans cesse et le repolissez ;
Ajoutez quelquefois, et souvent effacez.

Boileau, *L'Art poétique*

Remerciements

Je suis profondément reconnaissant à Kwangkeun Yi et Kenneth L. McMillan d'avoir accepté d'être rapporteurs de ma thèse. Je remercie Reinhard Wilhelm, Gilles Dowek et Manuel Hermenegildo de m'avoir fait l'honneur de participer à mon jury de thèse.

J'exprime ma sincère gratitude envers Radhia Cousot pour son patient encadrement depuis les premiers stages puis tout au long de cette thèse. Elle m'a donné la chance de travailler dans les meilleures conditions possibles. Je la remercie particulièrement pour son exigence de qualité scientifique, son ouverture sur le monde et sa compréhension.

Arnaud Venet a guidé mes premiers pas dans le monde de la recherche. Il m'a aidé dans mes choix les plus cruciaux. Je lui suis éternellement reconnaissant. Je remercie Patrick Cousot de m'avoir initié à la théorie de l'interprétation abstraite. Je remercie Christophe Blandin et Renaud Pacalet de m'avoir dévoilé les enjeux de la vérification de matériel et fourni des exemples à étudier.

Merci à mes formidables collègues de bureau Francesco Logozzo, Damien Massé et Élodie-Jane Sims. J'ai apprécié les moments d'intense réflexion que nous avons passés ensembles. Thank you to David Schmidt, Axel Simon and Eben Upton for brightening up the office we shared. Je remercie aussi les membres de la *magic team* à l'École Normale Supérieure et de l'équipe de ROPAS à la Seoul National University.

Merci à tous mes amis, en particulier à Denis Conduché et Xavier Rival pour leur amitié revigorante. Enfin, merci à ma famille et à Shinsuk pour leur soutien constant sans lequel cette thèse ne serait probablement pas.

Contents

Chapitre 1

Résumé de la thèse

1.1 La vérification de matériel

Chaque année, les progrès réalisés dans la fabrication de semi-conducteurs permettent l'intégration d'un nombre sans cesse croissant de transistors sur une puce. Cette évolution est malheureusement concomitante d'une augmentation dramatique des coûts non-récurrents de fabrication : un jeu de masques atteint 2 millions de dollars en technologie 65nm. Ainsi, les concepteurs doivent impérativement assurer la correction de systèmes aux complexités sans cesse accrus. À cette fin, de 60% à 70% du temps de développement est dédié à la simulation. Cette tendance ne peut que s'accentuer d'où la nécessité de développer des outils automatiques d'aide à la validation.

À la différence de la simulation, les méthodes formelles de vérification peuvent être exhaustives. Elles permettent de montrer la conformité d'une description de circuit à sa spécification. Cependant, les méthodes formelles doivent, elles aussi, affronter l'effroyable complexité des descriptions. Ainsi, elles souffrent souvent du problème *d'explosion des états*.

1.2 Objectif

Nous étudions la faisabilité d'un outil de vérification automatique qui réunit trois caractéristiques : assurance de couverture, efficacité de calcul, et intégration dans le flot de conception. Afin d'établir des propriétés de la description, et non pas seulement de trouver des erreurs, l'outil doit être exhaustif. Il doit couvrir l'ensemble des exécutions possibles. L'efficacité est un point crucial. C'est cela qui, véritablement, déterminera l'acceptation

de la méthode par les ingénieurs de conception. Nous nous fixons l'objectif d'être au moins compétitif avec la simulation traditionnelle. Enfin, l'outil doit travailler à partir de la même description que celle qui est déjà utilisée pour la simulation et la synthèse. De cette manière, l'intégration dans le flot de conception classique est assurée.

La théorie de l'interprétation abstraite permet de concilier ces objectifs apparemment contradictoires. L'interprétation abstraite formalise la notion d'abstraction entre sémantiques de programme. Son application originelle est la conception d'analyses statiques. L'idée consiste à calculer une version approchée de la sémantique. La régularité de la sémantique approchée est exploitée en utilisant des représentations en machine concises et des algorithmes puissants. Il est ainsi possible d'obtenir efficacité de calcul sans pour autant sacrifier l'exhaustivité. Cependant, abstraire a un coût : une abstraction trop rude ne sera pas en mesure de démontrer la correction du programme. Tout l'art de l'interprétation abstraite consiste donc à trouver le compromis idéal entre efficacité et précision pour un domaine d'application donné.

1.3 Résultats

Notre étude a pour objet le très répandu langage de description de matériel VHDL. Nous formalisons l'algorithme de simulation défini par le standard IEEE pour un noyau réaliste du langage. Chaque étape de calcul d'un simulateur événementiel est définie par une sémantique opérationnelle à la Plotkin. Essentiellement, une description VHDL se composée d'un nombre fini de processus exécutés en parallèle. La communication entre les processus s'effectue à travers une mémoire partagée, des signaux, et uniquement aux points de synchronisation. La synchronisation a lieu dès lors que tous les processus sont suspendus. À ce moment, la mémoire partagée est mise à jour, et certains processus sont réveillés par des événements de natures diverses : modification de la valeur d'un signal ou bien avancement du temps. Nous montrons que le parallélisme à la VHDL est bénin. Il est donc inutile d'explorer l'ensemble des entrelacements d'exécution de processus. On peut simplement se contenter de fixer leur ordre une fois pour toutes. Le sous-ensemble de VHDL traité manipule des entiers, de la logique standard (valeurs 0, 1 et U) et des tableaux statiquement alloués. Il contient du non-déterminisme par l'ajout de générateurs aléatoires et des fonctions de conversion d'un type de donnée à l'autre.

À partir de ce modèle mathématique, nous avons dérivé, systématiquement

par abstraction, une analyse statique pour VHDL. Cette analyse calcule une approximation supérieure de l'ensemble des états accessibles. Elle est paramétrée par le choix de l'abstraction de la mémoire. Ainsi, n'importe quel domaine numérique peut-être adopté et donnera lieu à un compromis original entre la précision des résultats et le coût du calcul. Nous avons évidemment réalisé la preuve de correction de l'analyse.

Enfin, nous présentons une instance possible de notre analyse statique. Elle est développée dans l'intention de valider des codes correcteurs d'erreurs linéaires. Le domaine numérique choisi est celui des égalités linéaires entre variables booléennes, doublé du domaine des constantes. Nous avons réalisé l'implémentation de l'analyse en OCaml. Il a aussi fallu écrire un préprocesseur qui traduit les descriptions VHDL dans le noyau. Ces prototypes nous ont donné l'occasion d'évaluer l'approche sur un cas concret. Nous avons appliqué l'analyse sur une description matérielle de code correcteur d'erreur de type Reed Solomon qui nous a été fourni par des partenaires industriels. Pour établir la correction du composant, il faut montrer que la composition encodage/corruption/décodage résulte bien en l'identité. Pour cela, nous comparons, automatiquement à l'aide de notre prototype, chaque composant avec une spécification de haut-niveau. Nous expliquons comment structurer cette spécification afin d'éviter l'explosion du nombre d'états générés par la présence de pipeline dans les composants. Les résultats sont remarquables : non seulement la vérification est un succès, mais de plus elle s'avère efficace. Ainsi, les performances de l'outil sont bien meilleures que le model checker VIS qui échoue par manque de mémoire. Notre prototype est aussi compétitif en terme de temps de calcul avec la simulation, tout en apportant évidemment une assurance beaucoup plus forte de correction.

Nous avons conçu et réalisé un outil de vérification automatique efficace, exhaustif et paramété. De plus, l'effort à consentir pour adjoindre à l'outil une abstraction différente est relativement réduit. Il suffit de modifier uniquement le domaine numérique en paramètre et il est inutile de se soucier de la sémantique de VHDL. Ce travail constitue une étape dans l'intégration d'outils formels aux flots de conception existants.

Chapter 2

Introduction

2.1 Semiconductors

Digital cameras, voice over IP phones, harvester guidance systems, factory monitors, smart rifles, brake controllers and pacemakers, all these devices incorporate integrated circuits. According to [MED04], the market of the semiconductor industry totaled $141 billion in 2003, while the electronic industry addressed a $800 billion market. Are there more tangible proofs of the pervasive influence of semiconductors on our society? Innovation in semiconductors contributes to the evolution of many industrial sectors. For instance electronics equipment amounts for as much as 20% of the production cost of a car.

The formidable proliferation of hardware in the world began not even a century ago. Bell Labs researchers William Shockley, John Bardeen and Walter Brattain created the first transistor in 1947. Schematically, see Fig. 2.1, a transistor operates like a switch: when voltage is applied to the gate (G) then the current between the source (S) and the drain (D) flows, otherwise source and drain are disconnected. Transistors constitute the basic blocks of electronic circuits. The ability to integrate a whole circuit on a single silicon wafer was invented a decade later. In 1958, Jack Kilby, while working at Texas Instrument, built the first integrated circuit. In 1959, Jean Hoerni and Robert Noyce at Fairchild developed the fabrication process still in use today: planar technology, see Fig. 2.1. A set of masks, which can be thought of the negatives for the various layers of the circuit, is manufactured. Then, in a way similar to photography, light is used to transfer the shape of the masks onto the silicon substrate, see Fig. 2.2.

Since then, the semiconductor industry has been driven by an exponen-

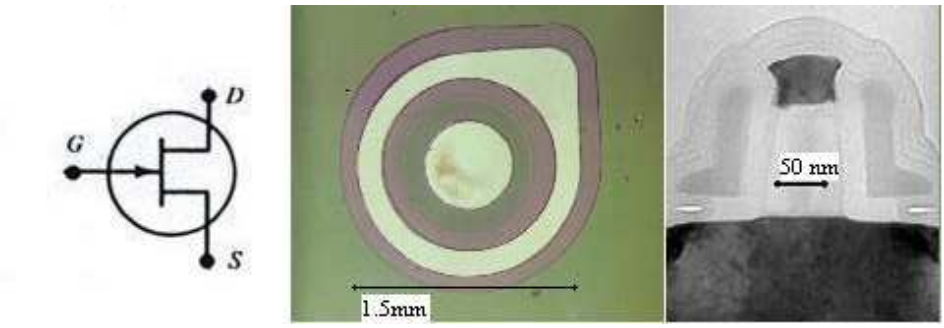


Figure 2.1: Transistor schematics, first planar transistor (source: smithsonianchips.si.edu) and 90nm transistor (source: intel)

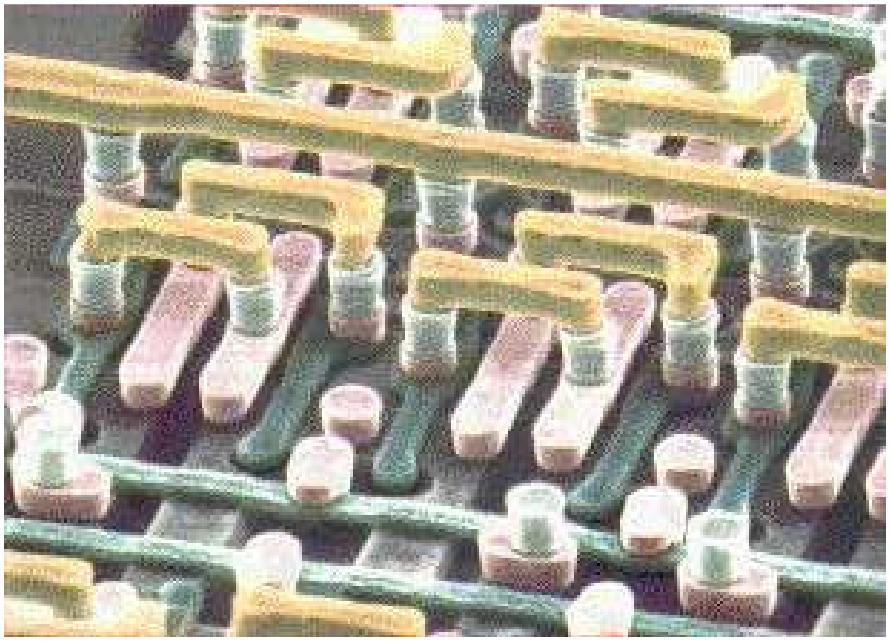


Figure 2.2: Interconnection structure of an SRAM (source: IBM Journal of R&D, Vol. 39, No. 4)

tial trend commonly known as Moore's law: the number of transistors per integrated circuits doubles every 18 months. In parallel to this evolution, the size and price of transistors have also decreased exponentially. Today, processors manufactured with the 90nm process, see Fig. 2.1, contain a few hundred million transistors. Over the years, integrated circuits with ever increasing functionality have created new applications and conquered new markets. These trends have allowed the semiconductor industry revenue to grow at an average of 14% per year [Jon03].

2.2 Verification crisis

Unfortunately the integration of numerous transistors on a chip has a high cost. The investment necessary to build a semiconductor manufacturing plant is evaluated in billion dollars. More importantly, the cost of mask sets have reached prohibitive amounts: it exceeds the million dollars for the 90nm technology and almost tops 2 million dollars in 65nm. To offset such high initial investments chips manufactured with the latest technology must be sold in very high volumes [MP03].

Even worse, designs should be error free before they are sent to fabrication: few companies can afford a million dollar re-spin. To ensure the correctness of their designs, engineers use simulation. Simulation accounts for 60% to 70% of the whole development time [MED02]. The number of engineers allocated to simulation outstrips the number of designers. Apparently, this proves not enough since almost as much as 50% of all designs go through more than one spin. The situation is bound to deteriorate as designs grow in size and complexity. Not only must the simulator handle bigger designs, it has to run more test-vectors so as to exercise a substantial fragment of all the behaviors, see Fig. 2.3. Largely because of the limitations of verification, the gap between what can be manufactured and what can be correctly designed quickly expands, see Fig. 2.4. According to the International Roadmap for Semiconductors [Int03], *the cost of design is the greatest threat to continuation of the semiconductor roadmap.* The situation is quickly becoming a crisis. There is a desperate need for new automatic tools that alleviate the task of verification.

2.3 Motivation

In contrast to simulation, formal verification methods can be exhaustive. They provide the unique capability of proving the correctness of a design

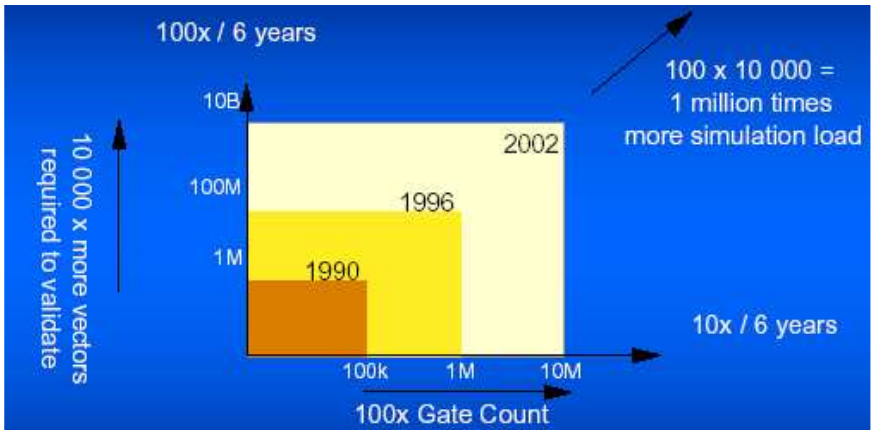


Figure 2.3: Increasing load on simulation (source: ST)

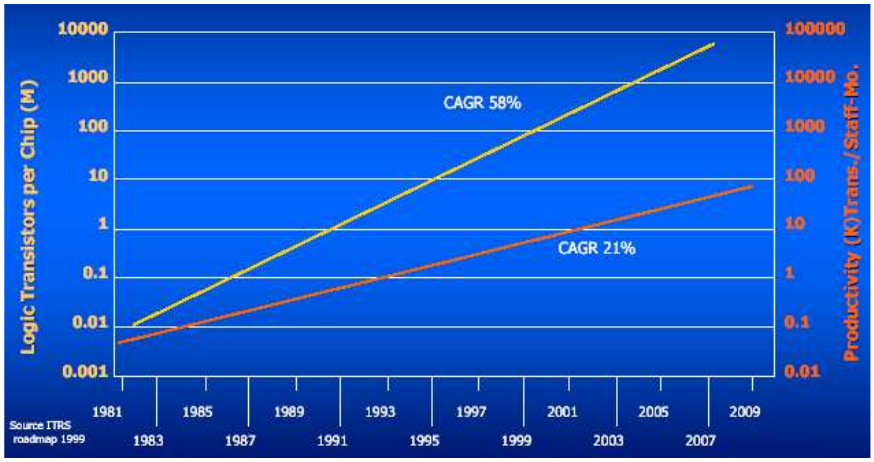


Figure 2.4: Design productivity gap (source: [Int03])

with respect to its specifications. However, formal methods too must cope with the sheer complexity of designs and often suffer the well-known state-explosion problem.

We study the feasibility of a formal verification tool that displays three apparently irreconcilable characteristics:

- full coverage,
- efficiency,
- seamless integration.

The tool should be exhaustive. It must cover all possible executions so that it is possible to prove properties of a design. Obviously, it must be sound so as to never report incoherent results. Efficiency is crucial as it mostly determines the acceptance of alternative methods to simulation. At least, the tool should compete advantageously with simulation. In particular, this means we are striving for automation. Tools that require too much manual intervention tend to be difficult to manage but by experts. On top of that, we don't want to impose a modification of existing design flows. Hence, the tool must input the same hardware description that is already used for simulation and synthesis. As an additional benefit, this avoids semantics mismatches that occur when distinct descriptions are used for verification and synthesis.

The framework of abstract interpretation [Cou78, CC77] provides the means to achieve all these goals. Abstract interpretation formalizes the notion of abstraction between program semantics. One of its primary application lies in the design of static analyses that compute approximate but sound information about all behaviors of programs. Such analyses need not retain all the details of the exact semantics. Instead, they can use concise representations and powerful algorithms that take advantage of the regularity enjoyed by the approximate semantics. Abstraction is the key concept to obtain efficiency. Soundness ensures that the results of the analysis are trustworthy and that erroneous conclusions about the design can not be drawn.

Naturally, abstraction has a cost. Sometimes, the results are too unprecise to attest the absence of errors in the design. Hence, the craft of abstract interpretation is to attain the optimal tradeoff between efficiency and precision.

2.4 Results

We design a static analysis for the popular hardware description language VHDL. It computes an approximation of the reachable states. We follow the methodology of abstraction interpretation.

First, we formalize the simulation algorithm for a realistic kernel of VHDL. We use the formalism of structural operational semantics. Very roughly, the semantics of an event-based simulator for VHDL works as follows. A VHDL description consists of a finite number of processes that are run concurrently. Communication between processes takes place through a shared memory, the signals, and at synchronization points only. Synchronization occurs after wait statements have suspended all processes. At this point, the simulator updates the shared memory and resumes the activity of some processes according to the events that just happened: a process was sensitive on a signal that was modified or time was advanced. We proved that the parallelism in VHDL is very weak. Indeed, it is useless to explore all possible interleaving of processes. It is sufficient to fix the order of execution of processes once and for all. The subset of VHDL that is considered manipulates standard multi-valued logic, integers and statically allocated arrays. Among others, it contains non-determinism thanks to random generators and functions that convert from one datatype to another.

The static analysis is systematically derived by abstraction from this mathematical model of the VHDL semantics. The abstract representation of sets of memories is left as a parameter of the analysis. Hence, any abstract numerical domain can be chosen and will lead to a distinct trade-off between the precision of the results and the cost of the analysis. We prove the soundness of the static analysis with respect to VHDL simulation semantics.

Then, we describe a possible instantiation of the analysis, which is tailored at the verification of linear error correcting codes. To represent sets of memories, we choose to combine the numerical domain of linear equalities with the domain of constants. We implemented a prototype of the resulting analysis in OCaml. We successfully applied the prototype to the verification of a Reed-Solomon error correcting code. The VHDL components were provided by industrial partners and were not originally written for the purpose of formal verification. The tool allows to automatically compare the description with high-level specifications. Because of the presence of a pipeline in the component, the tool could incur a state explosion problem. However, we explain how to structure the specifications so as to avoid this pitfall. The results of the experimentation are excellent: verification is a success and the tool proves to be efficient. It outperforms the BDD model checker VIS. It

is also competitive with industrial verification techniques since it completes in less time than what was allocated by the engineers for simulation.

This thesis is not specifically about a new semantics for VHDL; a new general framework for static analysis or new algorithms for manipulating linear constraints. It is rather about the holistic integration of these disparate techniques. The work presented in this thesis appeared as [Hym02a, Hym03, Hym04].

Organization of the thesis

Chapter 3 defines the simulation semantics of VHDL. Chapter 4 describes the design by abstract interpretation of a generic static analysis for VHDL. We show a possible instantiation of the analysis and explain the implementation choices of the tool in chapter 5. Eventually, chapter 6 presents the verification of a Reed Solomon encoder and decoder with our prototype.

Chapter 3

Semantics

3.1 Introduction

Very High Scale/Speed Integrated Circuits Hardware Description Language, aka VHDL, is one of the most widely used hardware description languages. In the early 1980s, the US Department of Defense was facing a crisis: its suppliers all had different ways of documenting hardware. Since the documentations were bound to specific manufacturing technology, they were reaching obsolescence very quickly. Needless to say that interoperability was an inextricable nightmare. Hence, VHDL was initially created as a language to describe the behavior of hardware. The ability to simulate descriptions soon became apparent. Finally, logic synthesis tools made the language even more attractive and gave rise to the modern hardware design methodology. In 1987, VHDL was standardized as IEEE Std 1076 [IEE87]. The up-to-date revision is [IEE02]. IEEE standard 1164 [IEE93] adds an important complement to the language: it defines a data-type to allow the manipulation of multi-valued logic signals.

We favored VHDL for its wide acceptance in the industry. We could have equally opted for Verilog [IEE01]. We strongly believe that a similar methodology can be adopted to develop equally efficient verification tools for Verilog. However, the case study, that professional hardware engineers made available to us, was written in VHDL.

Today, engineers spend most of their time working with VHDL (or Verilog). Lower level descriptions, i.e. netlists in edif [EIA93] or blif [Uni92] formats, are manipulated mostly by automatic tools. Higher level descriptions are used to evaluate the impact of hardware (and software) architectural decision early in the design cycle [DBB+02]. Performance/cost estimation

[CMP+01], early power analysis [BJ03] and sometimes high-level synthesis [SM01] can all be performed at this level. System level languages and tools have not reached maturity yet and still constitute an active field of research. Among others, system level design languages include SystemC used as in the Metropolis framework [BWH+03], Esterel and Time Modeling Language with the Archan approach [MC03], and Kahn Process Networks and C for the SPADE methodology [vdWLGH99].

In order to reason about VHDL descriptions, their semantics must be properly defined. This is the goal of the chapter.

3.2 Related work

Definitions of semantics for VHDL abound. This emphasizes the complexity of the IEEE standard [IEE02] which is subject to multiple interpretations. Semantics tend to vary according to their purpose, their presentation and the aspect of the language they concentrate on.

Goossens defines an operational semantics for a realistic yet manageable subset of VHDL in [Goo95]. The semantics is a formalization of the simulation algorithm as described in informal prose by the 1987 version of the VHDL IEEE standard. [TE01] provides an extension of this work to VHDL'93.

An interpreter for synchronous VHDL written in the pure functional language GOFER is presented in [BFK94]. A more complete but fairly complex implementation using evolving algebra pseudocode is given in [MBG94, BGM95]. The main purpose of interpreters is to palliate the imprecision of the IEEE standard by providing a reference implementation for simulators.

Van Tassel [Tas93] embeds an operational semantics of VHDL in HOL. Russinoff [Rus95a, Rus95b] presents the semantics of a relatively simple hardware description language close to VHDL and encodes it in the Boyer More logic. Formalization in the ACL2 logic of the simulation algorithm for a large and realistic subset of VHDL is described in [BG00, GBO02, DB97]. These embeddings in proof assistants allow the further undertaking of formal proofs of correctness of VHDL descriptions. Furthermore, the formalization in ACL2 [GBO02] immediately provides a symbolic simulator.

The compilation of VHDL descriptions into finite state machines for the purpose of symbolic model checking has been explored. Such compilation process is described in [LBPV94]. In [DB93, DB95, DSC98], designs are translated into finite state machines where one transition of the machine emulates one delta cycle. Encrenaz [Enc95] adopts a similar approach but

descr	→	*process* { | *process* }	(Parallel composition)
process	→	*command*; { *command*; }	(Sequence)
command	→	*lval* := *exp*	(Variable assignment)
	→	*lval* <= *exp*	(Signal assignment)
	→	wait on *sig_list*	
		until *exp* for *timeout*	(Suspension)
	→	while *exp* do *process*	(Iteration)
	→	$\Sigma_{i\in[1,n]}$ *exp*$_i$ -> *process*$_i$	(Alternative)
	→	display(*x*, ..., *x*)	(Display)
	→	lvec_to_int(*x*, *n*, *x*)	
	→	int_to_lvec(*x*, *n*, *x*)	(Conversion functions)
lval	→	*x* | *lval*[*exp*]	(Memory accesses)
exp	→	'0' | '1' | 'U'	
	→	*n* $\in \mathbb{Z}$ | true | false	(Constants)
	→	*lval* | *op exp* | *exp op exp*	
	→	lrnd() | rnd()	(Random generators)
	→	rising_edge(*x*)	(Edge detector)
op	→	not | or | and	
	→	lnot | lxor | land	
	→	- | + | < | =	

where *x* is a variable or a signal identifier, *sig_list* a possibly empty set of signal names and *timeout* a positive integer or the keyword ever (to denote the absence of timeout clause). The notation {...} reads "zero or more instances of the enclosed".

Figure 3.1: Syntax of mini-VHDL

uses petri nets as an intermediate representation.

Slightly more esoteric work include some denotational semantics [BFK95] and [FM95], a Hoare logic implemented in Prolog [BSK94], a compilatory semantics to the temporal logic of actions [Gol94].

Similar work has been done with the Verilog language [Gor95].

3.3 Syntax

As specified by the standard [IEE02], the elaboration phase translates descriptions into an executable model. We have implemented a front-end to automatically perform this preliminary step. It first parses the source files. It instantiates all modules and produces a flat description. Then it normal-

izes the various VHDL statements so that they all fit a more concise syntax. At the same time, it renames all identifiers in order to safely hoist all definitions at the global scope. It then checks and propagates type information. At last, it scalarizes array assignments. For instance the following code:

```
a(i) := a(i) xor a(i + 1);
-- where the type of variable a is defined as
-- array(15 downto 0) of std_logic_vector(3 downto 0)
```

becomes:

```
a[i][3] := a[i][3] xor a[i + 1][3];
a[i][2] := a[i][2] xor a[i + 1][2];
a[i][1] := a[i][1] xor a[i + 1][1];
a[i][0] := a[i][0] xor a[i + 1][0];
```

All these transformations are very conventional: see for instance [WM95, App98] for a description of basic type checking, the VHDL IEEE standard [IEE02] describes hierarchy flattening and [SV00] explains the scalarisation process for Verilog.

Elaborated descriptions belong to mini-VHDL, the language whose abstract syntax is depicted in Fig. 3.1. A description consists of the parallel composition of a fixed number of sequential processes. Each process is a sequence of statements. A statement may either be a variable or signal assignment, a wait statement, a while loop, an alternative construct or a print statement. The descriptions manipulate std_logics, booleans, integers and statically allocated multi-dimensional arrays. Of the std_logic literals only `'0'`, `'1'` and `'U'` are actually used. Operators that expect arguments of the type std_logic are prefixed by the letter `l` as in `lnot`, `lxor` and `land`. The `rising_edge` is necessary to detect rising edges of clocks. Functions `lvec_to_int` and `int_to_lvec` convert an array of std_logic to an integer and back. Even though these three functions are not basic VHDL but defined in external IEEE packages [IEE93], we consider them as primitive operators. We also incorporate random generators `rnd` and `lrnd` to be able to inject non-determinism in a design. We chose this subset of the language because it allows to handle exactly all the features of the industrial case study we present later in chapter 6. Notice we lack any dynamically allocated datatypes. We also deliberately ban the delayed signal assignments (signal assignments with after clause). As a result, the exact layout of the memory is known at compile time.

3.4 Intuition for the semantic model

The processes of a mini-VHDL description are run concurrently. Communication between processes happens through a shared memory (the signals) and only at some specific synchronization points. Once all processes are suspended, synchronization occurs: the memory is updated and the activity of some processes is resumed. We call a simulation cycle, the execution of the system between two consecutive synchronization points.

3.5 The kernel

We give an operational semantics in the style of Plotkin [Plo81] to mini-VHDL. We introduce our semantics gradually. Let us start with a very limited kernel language that nevertheless embodies the essential paradigm of the VHDL language. It is reduced to only four statements:

command	→	`x <= exp`	(Assignment)
	→	`wait on sig_list for timeout`	(Suspension)
	→	`forever do process`	(Iteration)
	→	`if exp then process`	(Selection)

Memory holding elements are restricted to signals of the type std_logic. We denote by Sig the set of all signals defined in the design.

Example 3.1. We will illustrate our definitions on the following piece of code:

```
  [1]forever do
    [2]clk <= lnot clk;
    [3]wait on {} for 1;
|
  [4]forever do
    [5]if clk = '1' then [6]o <= x land (lnot y);
    [7]wait on {clk} for ever;
|
  [8]x <= '0'; [9]y <= '1';
  [10]wait on clk for ever;
  [11]forever do
    [12]x <= lrnd();
    [13]wait on {clk} for ever;
```

$$\frac{}{\rho \vdash \texttt{'0'} \Longrightarrow \texttt{'0'}} \qquad \frac{}{\rho \vdash \texttt{'1'} \Longrightarrow \texttt{'1'}} \qquad \frac{}{\rho \vdash \texttt{'U'} \Longrightarrow \texttt{'U'}}$$

$$\frac{}{\rho \vdash \texttt{true} \Longrightarrow \texttt{true}} \qquad \frac{}{\rho \vdash \texttt{false} \Longrightarrow \texttt{false}}$$

$$\frac{v \in \{\texttt{true}, \texttt{false}\}}{\rho \vdash \texttt{rnd()} \Longrightarrow v} \qquad \frac{v \in \{\texttt{'0'}, \texttt{'1'}\}}{\rho \vdash \texttt{lrnd()} \Longrightarrow v} \qquad \frac{}{\rho \vdash \boldsymbol{x} \Longrightarrow \rho(\boldsymbol{x})}$$

$$\frac{\rho \vdash \boldsymbol{exp} \Longrightarrow v}{\rho \vdash \boldsymbol{op}\ \ \boldsymbol{exp} \Longrightarrow [\![op]\!](v)} \qquad \frac{\rho \vdash \boldsymbol{exp}_1 \Longrightarrow v_1 \qquad \rho \vdash \boldsymbol{exp}_2 \Longrightarrow v_2}{\rho \vdash \boldsymbol{exp}_1\ \ \boldsymbol{op}\ \ \boldsymbol{exp}_2 \Longrightarrow [\![op]\!](v_1, v_2)}$$

$$[\![\texttt{lnot}]\!](v) = \begin{cases} \texttt{'0'} & \text{if } v = \texttt{'1'} \\ \texttt{'1'} & \text{if } v = \texttt{'0'} \\ \texttt{'U'} & \text{otherwise} \end{cases}$$

$$[\![\texttt{land}]\!](v_1, v_2) = \begin{cases} \texttt{'1'} & \text{if } v_1 = \texttt{'1'} \text{ and } v_2 = \texttt{'1'} \\ \texttt{'0'} & \text{if } v_1 = \texttt{'0'} \text{ or } v_2 = \texttt{'0'} \\ \texttt{'U'} & \text{otherwise} \end{cases}$$

$$[\![\texttt{lxor}]\!](v_1, v_2) = \begin{cases} \texttt{'U'} & \text{if } v_1 = \texttt{'U'} \text{ or } v_2 = \texttt{'U'} \\ \texttt{'1'} & \text{else if } v_1 \neq v_2 \\ \texttt{'0'} & \text{otherwise} \end{cases}$$

$$[\![\texttt{not}]\!](v) = \neg v$$

$$[\![\texttt{and}]\!](v_1, v_2) = v_1 \wedge v_2$$

$$[\![\texttt{or}]\!](v_1, v_2) = v_1 \vee v_2$$

$$[\![\texttt{=}]\!](v_1, v_2) = v_1 = v_2$$

Figure 3.2: Semantics of expressions

This description manipulates signals from the set $Sig = \{\texttt{clk}, \texttt{x}, \texttt{y}, \texttt{o}\}$. It is made up of three processes. A clock generator toggles the clock signal every one unit of time. The middle process computes the output signal `o` from the input signals `x` and `y` on high edges of the clock. A testbench feeds values to the input signals.

We assume each program statement ${}^lC$ is uniquely tagged with a label l. The label of the unique statement which follows ${}^lC$ in the control flow graph of the enclosing process can be fetched with $next(l)$. The current point of execution of an individual process is determined by a control point c. For running processes, the control point simply amounts to the label of the statement that has to be executed next. Whereas, the control point of a suspended process is augmented with a list of signals W and a duration t. The duration is either a strictly positive integer or ∞ to indicate the absence of timeout. We denote by $\mathcal{P}$ the set of all control points.

For each signal x, defined in the description, a global environment ρ stores its current and future value, respectively at address x and x^+. So the environment is a mapping from the set of all memory addresses $\mathcal{A}$ to values in $\mathbb{V}$. At this point of the exposition, $\mathbb{V}$ equates to the set of std_logic values, that is $\mathbb{L} = \{\texttt{'0'}, \texttt{'1'}, \texttt{'U'}\}$.

We impose the syntactic restriction on descriptions that no signal can be assigned by more than one process. That way, it is sufficient to store only one future value for every signal. We could introduce resolution functions as defined in the standard (see section 2.4 in [IEE02]) at the price of a slight complexification of the semantics definition. It is also possible to emulate resolution functions with additional processes.

At last, a state of execution is a tuple $(c_1, \ldots, c_n, \rho)$ of control points, followed by a global environment. We denote by Σ, the set of all possible states of execution.

An expression ***exp*** evaluates to a value $v \in \mathbb{V}$ in an environment ρ, which we express by judgments of the form $\rho \vdash \textbf{\textit{exp}} \Longrightarrow v$. The meaning of expressions is defined in Fig. 3.2 by structural induction in the classical way [Plo81]. Let us simply point out that the function `lrnd()` arbitrarily picks a std_logic between `'1'` and `'0'`.

Figure 3.3 depicts the sequential execution of an individual process. Signal assignments schedule modifications of the shared memory for the next simulation cycle. The expression on the right-hand side of the statement is evaluated. Then, the resulting value is stored at the address that holds the future value of the assigned signal. Note that a subsequent assignment to the same signal before the next synchronization would replace this value. A

$$\text{sig}\frac{{}^{l}\texttt{x <= exp} \qquad \rho \vdash \texttt{exp} \Longrightarrow v}{(l,\rho) \rightharpoonup (next(l), \rho[\texttt{x}^{+} \leftarrow v])}$$

$$\text{wait}\frac{{}^{l}\texttt{wait on } W \texttt{ for } t \qquad c = (next(l), W, t)}{(l,\rho) \rightharpoonup (c,\rho)}$$

$$\text{enter}\frac{{}^{l}\texttt{forever do } {}^{l'}C\texttt{;}P}{(l,\rho) \rightharpoonup (l',\rho)}$$

$$\text{true}\frac{{}^{l}\texttt{if } b \texttt{ then } {}^{l'}C\texttt{;}P \qquad \rho \vdash b \Longrightarrow \texttt{true}}{(l,\rho) \rightharpoonup (l',\rho)} \qquad \text{false}\frac{{}^{l}\texttt{if } b \texttt{ then } P \qquad \rho \vdash b \Longrightarrow \texttt{false}}{(l,\rho) \rightharpoonup (next(l),\rho)}$$

Figure 3.3: Sequential execution

$$\Pi\text{-}i\frac{(c_i,\rho) \rightharpoonup (c_i',\rho')}{(c_1,\ldots,c_i,\ldots,c_n,\rho) \rightarrow (c_1,\ldots,c_i',\ldots,c_n,\rho')}$$

$$\Delta\frac{\forall i : c_i = (l_i, W_i, t_i) \qquad \exists i : wake(W_i,\rho) \qquad \forall i : c_i' = \begin{cases} l_i & \text{if } wake(W_i,\rho) \\ c_i & \text{otherwise} \end{cases} \qquad \rho' = update(\rho)}{(c_1,\ldots,c_n,\rho) \rightarrow (c_1',\ldots,c_n',\rho')}$$

$$\Theta\frac{\forall i : c_i = (l_i, W_i, t_i) \qquad \forall i : \neg wake(W_i,\rho) \qquad \begin{array}{c}\exists i : t_i \neq \infty \\ t = \min\{t_i \neq \infty\}\end{array} \qquad \forall i : c_i' = \begin{cases} l_i & \text{if } t_i = t \\ (l_i, W_i, t_i - t) & \text{if } t_i \neq \infty \\ c_i & \text{otherwise} \end{cases} \qquad \rho' = update(\rho)}{(c_1,\ldots,c_n,\rho) \rightarrow (c_1',\ldots,c_n',\rho')}$$

Figure 3.4: Simulation algorithm

wait statement suspends the process. It augments the control point with the conditions that control the awakening. The process will resume its activity if an event occurs on some of the signals in the sensitivity list W or the timeout t is elapsed. The last two statements control the flow of execution and the corresponding rules are classical. Execution simply enters the body of the forever loop. The fact that execution returns to the head of the loop is implicitly encoded by the *next* function. For the guard statement, the point where execution is transfered depends on the evaluation of the boolean condition.

Example 3.2. The operational semantics for the individual processes of our running example 3.1 is:

$$\begin{aligned}
\rightharpoonup \;=\; & \{(1,\rho) \rightharpoonup (2,\rho)\} \cup \{(2,\rho) \rightharpoonup (3,\rho[\mathtt{clk}^+ \leftarrow [\![\mathtt{lnot}]\!](\mathtt{clk})]\} \\
& \cup\{(3,\rho) \rightharpoonup ((1,\emptyset,1),\rho)\} \cup \{(4,\rho) \rightharpoonup (5,\rho)\} \\
& \cup\{(5,\rho) \rightharpoonup (6,\rho) \mid \rho(\mathtt{clk}) = \mathtt{'1'}\} \cup \{(5,\rho) \rightharpoonup (7,\rho) \mid \rho(\mathtt{clk}) \neq \mathtt{'1'}\} \\
& \cup\{(6,\rho) \rightharpoonup (7,\rho[\mathtt{o}^+ \leftarrow [\![\mathtt{land}]\!](\mathtt{x},[\![\mathtt{lnot}]\!](\mathtt{y})])\} \\
& \cup\{(7,\rho) \rightharpoonup ((4,\{\mathtt{clk}\},\infty),\rho)\} \\
& \cup\{(8,\rho) \rightharpoonup (9,\rho[\mathtt{x}^+ \leftarrow \mathtt{'0'}])\} \cup \{(9,\rho) \rightharpoonup (10,\rho[\mathtt{y}^+ \leftarrow \mathtt{'1'}])\} \\
& \cup\{(10,\rho) \rightharpoonup ((11,\{\mathtt{clk}\},\infty),\rho)\} \cup \{(11,\rho) \rightharpoonup (12,\rho)\} \\
& \cup\{(12,\rho) \rightharpoonup (13,\rho[\mathtt{x}^+ \leftarrow v]) \mid v = \mathtt{'0'} \vee v = \mathtt{'1'}\} \\
& \cup\{(13,\rho) \rightharpoonup ((11,\{\mathtt{clk}\},\infty),\rho)\}
\end{aligned}$$

Now, the three rules of Fig. 3.4 are enough to completely characterize the simulation algorithm of our kernel language. Processes are run concurrently as long as possible thanks to the first rule. Successive steps of execution with the first rule between the application of one of the last two rules constitutes a simulation cycle.

Once all processes are suspended, one of the rules Δ or Θ applies. The first rule is taken whenever an event has occurred on some of the signals in the sensitivity list of one of the processes.

$$wake(W,\rho) = \exists x \in W : \rho(x) \neq \rho(x^+)\,.$$

The Δ rule revives any process which satisfies the predicate *wake*. Then, the shared memory is updated, so that signal assignments encountered during the previous simulation cycle now take effect:

$$\forall x \in Sig : update(\rho)(x) = \rho(x^+)\,.$$

c_1	c_2	c_3	clk/clk$^+$	x/x$^+$	y/y$^+$	o/o$^+$
1	4	8	'0'/'0'	'0'/'0'	'0'/'0'	'0'/'0'
2	4	8	'0'/'0'	'0'/'0'	'0'/'0'	'0'/'0'
3	4	8	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	4	8	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	5	8	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	7	8	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	8	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	9	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	10	'0'/'1'	'0'/'0'	'0'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11, {clk}, ∞	'0'/'1'	'0'/'0'	'0'/'1'	'0'/'0'
1, $\emptyset$, 1	4	11	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	5	11	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	6	11	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	7	11	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	12	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	13	'1'/'1'	'0'/'1'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11, {clk}, ∞	'1'/'1'	'0'/'1'	'1'/'1'	'0'/'0'
1	4, {clk}, ∞	11, {clk}, ∞	'1'/'1'	'1'/'1'	'1'/'1'	'0'/'0'
2	4, {clk}, ∞	11, {clk}, ∞	'1'/'1'	'1'/'1'	'1'/'1'	'0'/'0'
3	4, {clk}, ∞	11, {clk}, ∞	'1'/'0'	'1'/'1'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11, {clk}, ∞	'1'/'0'	'1'/'1'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4	11	'0'/'0'	'1'/'1'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	5	11	'0'/'0'	'1'/'1'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	7	11	'0'/'0'	'1'/'1'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11	'0'/'0'	'1'/'1'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	12	'0'/'0'	'1'/'1'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	13	'0'/'0'	'1'/'1'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11, {clk}, ∞	'0'/'0'	'1'/'1'	'1'/'1'	'0'/'0'

Table 3.1: A run of the simulation algorithm

In this case, no physical time has elapsed. In the VHDL phraseology, the next cycle will be a delta cycle. Delta cycles are the way for an event driven simulation to emulate the instantaneous propagation of electricity through wires.

The Δ rule has the priority over the last one, which advances the simulation time by the smallest timeout. All the processes for which the timeout has just expired resume their execution. It may be that neither the delta nor the time-advance rule are fired. This happens when all sensitive signals remain unmodified and all processes wait for ever, in which case the execution is simply blocked. However, simulation usually runs forever.

The initial configuration s_0 of a description ${}^{l_1}P_1 \mid \ldots \mid {}^{l_n}P_n$ is of the form $(l_1, \ldots, l_n, \rho_0)$. It is entirely determined by the initial environment ρ_0 where current and future values of signals must coincide. By default std_logic signals begin with the `'U'` value and integers are initially equal to 0. However, different initial value may be specified in the description.

Example 3.3. We tabulate, see table 3.1, the first four cycles of a possible simulation run for the code introduced in example 3.1. All signals are given the initial value `'0'`.

3.6 More data types: integers and aggregates

Mini-VHDL descriptions not only handle std_logic values but also integer and multi-dimensional arrays:

$$\begin{aligned} \beta &= \text{int} \mid \text{std\_logic} \\ \tau &= \beta \mid \tau[l \ldots u] \,. \end{aligned}$$

We denote by $x : \tau$ the fact that signal x has been defined with the particular type τ. The set of addresses that are created to store the values of a signal are defined recursively. To build the address of an individual cell, its indices are simply concatenated to the name of the signal:

$$\begin{cases} @(x, \beta) = \{x\} \\ @(x, \tau[l \ldots u]) = \{a.i \mid a \in @(x, \tau), l \leq i \leq u\} \,. \end{cases}$$

For any set X, we define X^+, to be $\{x^+ \mid x \in X\}$. The set of all addresses allocated to store the current and future value of signals is:

$$\begin{aligned} \mathcal{A}|_{Sig} &= \{a \in @(x, \tau) \mid x \in Sig, x : \tau\} \\ \mathcal{A} &= \mathcal{A}|_{Sig} \cup \mathcal{A}|^+_{Sig} \,. \end{aligned}$$

The domain of the global environment now incorporates all these new addresses. In addition to std_logics, its range also contains integers:

$$\begin{aligned} \mathbb{V} &= \mathbb{L} \cup \mathbb{Z} \\ \rho &: \mathcal{A} \to \mathbb{V} \,. \end{aligned}$$

Example 3.4. If the only signal defined in the description is an array `b` of type std_logic[1 . . . 3][0 . . . 3], then the set of memory locations is:

$$\begin{aligned} \mathcal{A} = \{&\mathtt{b}.1.0, \mathtt{b}.1.0^+, \mathtt{b}.1.1, \mathtt{b}.1.1^+, \mathtt{b}.1.2, \mathtt{b}.1.2^+, \mathtt{b}.1.3, \mathtt{b}.1.3^+, \\ &\mathtt{b}.2.0, \mathtt{b}.2.0^+, \mathtt{b}.2.1, \mathtt{b}.2.1^+, \mathtt{b}.2.2, \mathtt{b}.2.2^+, \mathtt{b}.2.3, \mathtt{b}.2.3^+, \\ &\mathtt{b}.3.0, \mathtt{b}.3.0^+, \mathtt{b}.3.1, \mathtt{b}.3.1^+, \mathtt{b}.3.2, \mathtt{b}.3.2^+, \mathtt{b}.3.3, \mathtt{b}.3.3^+\} \,. \end{aligned}$$

The evaluation of expressions that manipulate integers is nothing but usual:

$$\frac{}{\rho \vdash n \Longrightarrow n}$$

$$\begin{aligned} [\![\mathtt{+}]\!](v_1, v_2) &= v_1 + v_2 \\ [\![\mathtt{-}]\!](v_1, v_2) &= v_1 - v_2 \\ [\![\mathtt{<}]\!](v_1, v_2) &= v_1 < v_2 \end{aligned}$$

We introduce the judgment $\rho \vdash_L \mathit{lval} \Longrightarrow a : \tau$ that evaluates left values to memory addresses and also returns their type:

$$\frac{\mathit{x} : \tau}{\rho \vdash_L \mathit{x} \Longrightarrow \mathit{x} : \tau}$$

$$\frac{\rho \vdash_L \mathit{lval} \Longrightarrow a : \tau[l \ldots u] \qquad \rho \vdash \mathit{exp} \Longrightarrow i \qquad l \leq i \leq u}{\rho \vdash_L \mathit{lval}\,\mathtt{[}\mathit{exp}\mathtt{]} \Longrightarrow a.i : \tau}$$

Notice that this semantics checks that every index lies within correct bounds. An out-of-bounds would block the execution. At last, the semantics of memory accesses and signal assignments are modified accordingly as follows:

$$\frac{\rho \vdash_L \mathit{lval} \Longrightarrow a : \tau}{\rho \vdash \mathit{lval} \Longrightarrow \rho(a)}$$

$$\text{sig}\frac{^{l}\,\mathit{lval}\ \mathtt{<=}\ \mathit{exp} \qquad \rho \vdash_L \mathit{lval} \Longrightarrow a : \beta \qquad \rho \vdash \mathit{exp} \Longrightarrow v}{(l, \rho) \rightharpoonup (\mathit{next}(l), \rho[a^+ \leftarrow v])}$$

During the update of the memory, the future value of each scalar signal and of each element of signal array replaces its corresponding current value:

$$\mathit{update}(\rho)(a) = \begin{cases} \rho(a^+) & a \in \mathcal{A}|_{Sig} \\ \rho(a) & \text{otherwise} \,. \end{cases}$$

3.7 Local variables, conversion functions

We allow variables in mini-VHDL descriptions as long as their use is confined to individual processes. Shared variables, as introduced in the '93 version of the standard, are risky as they possibly make the outcome of a computation depend on the order into which processes are executed.

The domain of global environments is extended with the addresses for the variables *Var* defined in the description:

$$\begin{aligned} \mathcal{A}|_{Var} &= \{a \in @(x,\tau) \mid x \in Var, x:\tau\} \\ \mathcal{A} &= \mathcal{A}|_{Var} \cup \mathcal{A}|_{Sig} \cup \mathcal{A}|^{+}_{Sig}\,. \end{aligned}$$

Variables are assigned in an identical way to what happens with classical programming languages:

$$\text{var}\frac{{}^{l}\texttt{lval := exp} \qquad \rho \vdash_L \texttt{lval} \Longrightarrow a:\beta \qquad \rho \vdash \texttt{exp} \Longrightarrow v}{(l,\rho) \rightharpoonup (next(l), \rho[a \leftarrow v])}$$

Integer variables are particularly helpful to control iterations of while loops. Let us recall the behavior of loops:

$$\text{enter}\frac{{}^{l}\texttt{while } b \texttt{ do } {}^{l'}C;P \qquad \rho \vdash b \Longrightarrow \texttt{true}}{(l,\rho) \rightharpoonup (l',\rho)}$$

$$\text{exit}\frac{{}^{l}\texttt{while } b \texttt{ do } P \qquad \rho \vdash b \Longrightarrow \texttt{false}}{(l,\rho) \rightharpoonup (next(l),\rho)}$$

Variables also play the rôle of intermediate storage when converting between integers and vectors of std_logic. We have two conversion functions: `lvec_to_int` behaves somewhat like the `conv_integer` of the official VHDL package std_logic_unsigned, and `int_to_lvec` is found in the std_logic_arith package under the name `conv_std_logic_vector`. The argument n denotes the length of the array y. So we assume that y : std_logic$[0 \ldots n-1]$. This is checked during the preliminary typing phase. The semantics of the conversion functions is developed in figure 3.5. By convention the left-most cell of the array contains the strongest bit.

Example 3.5. At this point we have enough definitions to ascribe a semantics to a piece of code like the following:

```
i := 0;
while i < 16 do
```

$$\frac{{}^{l}\texttt{lvec\_to\_int}(y,\ n,\ x) \qquad \forall i \in [0, n-1] : \rho(y.i) \in \{\texttt{'0'}, \texttt{'1'}\} \qquad \varepsilon_i = \begin{cases} 1 & \text{if } \rho(y.i) = \texttt{'1'} \\ 0 & \text{otherwise} \end{cases} \qquad v = \sum_{i=1}^{i=n} 2^{n-i}\varepsilon_i}{(l, \rho) \rightharpoonup (next(l), \rho[x \leftarrow v])}$$

$$\frac{{}^{l}\texttt{int\_to\_lvec}(x,\ n,\ y) \qquad v = \rho(x) \qquad \forall i \in [1, n] : \varepsilon_i = \begin{cases} \texttt{'1'} & \text{if } (v/2^{n-i})\%2 = 1 \\ \texttt{'0'} & \text{if } (v/2^{n-i})\%2 = 0 \end{cases} \qquad \rho' = \rho[y.0 \leftarrow \varepsilon_0] \ldots [y.(n-1) \leftarrow \varepsilon_{n-1}]}{(l, \rho) \rightharpoonup (next(l), \rho')}$$

Figure 3.5: Conversion functions

```
int_to_lvec(i, 4, tmp);
a[i][0] <= tmp[0];
a[i][1] <= tmp[1];
a[i][2] <= tmp[2];
a[i][3] <= tmp[3];
i := i + 1;
```

When this program terminates, the future value of the i^{th} element of array `a` contains the binary representation of i.

3.8 Rising edges and clocks

As explained in section 3.5, for each signal, we store the current value and the value that is scheduled for the next simulation cycle. In addition, for some signals, it is necessary to retain the value they had during the previous simulation cycle. We denote by Clk the set of all such signals. Designs are synchronized on the edges of clocks. Hence, the unique purpose of the previous value is to detect rising edges, that is transitions from `'0'` to `'1'`.

Let us update one last time the definition of the set of memory addresses

and the memory update during synchronization:

$$\begin{aligned} \mathcal{A}|_{Clk} &= \{a \in @(x,\tau) \mid x \in Clk, x:\tau\} \\ X^- &= \{x^- \mid x \in X\} \\ \mathcal{A} &= \mathcal{A}|_{Var} \cup \mathcal{A}|_{Sig} \cup \mathcal{A}|^+_{Sig} \cup \mathcal{A}|^-_{Clk} \cup \mathcal{A}|_{Clk} \cup \mathcal{A}|^+_{Clk} \end{aligned}$$

$$\begin{cases} update(\rho)(a) = \rho(a^+) & a \in \mathcal{A}|_{Sig} \cup \mathcal{A}|_{Clk} \\ update(\rho)(a^-) = \rho(a) & a \in \mathcal{A}|_{Clk} \\ update(\rho)(a) = \rho(a) & \text{otherwise}\,. \end{cases}$$

We describe the semantics of the `rising_edge` operator below:

$$\frac{v = \begin{cases} \texttt{true} & \text{if } \rho(x^-) = \texttt{'0'} \wedge \rho(x) = \texttt{'1'} \\ \texttt{false} & \text{otherwise} \end{cases}}{\rho \vdash \texttt{rising\_edge(}x\texttt{)} \Longrightarrow v}$$

3.9 Remaining statements

3.9.1 Suspension

The VHDL full suspension statement is slightly richer than the one introduced in section 3.5. In addition to the sensitivity list W and the timeout t, a boolean condition b is another argument of the wait statement. The semantics of wait becomes:

$$\text{wait}\frac{^l\texttt{wait on } W \texttt{ until } b \texttt{ for } t \qquad c = (next(l), W, b, t)}{(l,\rho) \rightharpoonup (c,\rho)}$$

For a suspended process to wake up when some signal in the list W, the condition b must evaluate to `true` in the environment *after it has been updated*. The revised Δ and Θ rules appear in Fig. 3.6.

3.9.2 Alternative

The alternative construct, introduced by Dijkstra in [Dij75], is somewhat similar to the selection statement. It picks arbitrarily one of the guarded processes whose guard evaluates to `true` and runs it:

$$\text{choose}\frac{^l\Sigma_{i\in[1,n]} b_i \texttt{ -> } ^{l_i}P_i \qquad \rho \vdash b_j \Longrightarrow \texttt{true}}{(l,\rho) \rightharpoonup (l_j,\rho)}$$

$$\Delta \frac{\forall i : c_i = (l_i, W_i, b_i, t_i) \qquad \rho' = update(\rho) \qquad \exists i : wake(W_i, b_i, \rho, \rho') \qquad \forall i : c_i' = \begin{cases} l_i & \text{if } wake(W_i, b_i, \rho, \rho') \\ c_i & \text{otherwise} \end{cases}}{(c_1, \ldots, c_n, \rho) \rightarrow (c_1', \ldots, c_n', \rho')}$$

$$\Theta \frac{\forall i : c_i = (l_i, W_i, b_i, t_i) \qquad \rho' = update(\rho) \qquad \forall i : \neg wake(W_i, b_i, \rho, \rho') \qquad \begin{array}{c} \exists i : t_i \neq \infty \\ t = \min\{t_i \neq \infty\} \end{array} \qquad \forall i : c_i' = \begin{cases} l_i & \text{if } t_i = t \\ (l_i, W_i, b_i, t_i - t) & \text{if } t_i \neq \infty \\ c_i & \text{otherwise} \end{cases}}{(c_1, \ldots, c_n, \rho) \rightarrow (c_1', \ldots, c_n', \rho')}$$

$$wake(W, b, \rho, \rho') = (\exists x \in W : \rho(x) \neq \rho'(x)) \wedge (\rho' \vdash b \Longrightarrow \texttt{true})$$

Figure 3.6: Synchronization rules

3.9.3 Display

The display command outputs the value of signals and variables to the screen. However, here, it appears not to do much, since we decide not to model screen outputs in our semantics. It proves useful later as a means to report results of the analysis.

$$\text{show} \frac{^{l}\texttt{display(}x\texttt{, ..., }x\texttt{)}}{(l, \rho) \rightharpoonup (next(l), \rho)}$$

3.10 A benign parallelism

To account for the concurrency in VHDL, we adopted an interleaving model: according to the Π rule, processes can run in any possible order. It can be computationally very expensive to fully explore all the interleaved executions of processes. Fortunately, this is not necessary. In fact, VHDL's parallelism is very weak. It is sufficient to fix a particular ordering for the execution of processes. We replace the Π rule by the more restrictive Ψ rule which always evaluates the left-most awake process first:

$$\Psi\text{-}i \frac{\forall j < i : c_j = (l_j, W_j, b_j, t_j) \qquad (c_i, \rho) \rightharpoonup (c_i', \rho')}{(c_1, \ldots, c_i, \ldots, c_n, \rho) \rightarrow (c_1, \ldots, c_i', \ldots, c_n, \rho')}$$

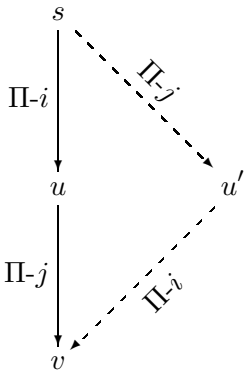


Figure 3.7: Free swap.

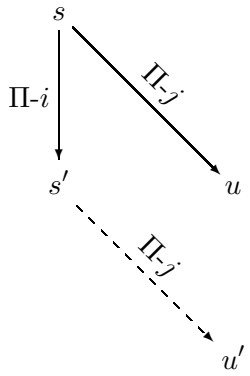


Figure 3.8: Non blocking.

In VHDL, processes modify distinct portion of the memory, so the order into which processes are run during the simulation cycle does not matter. This fact ensures the previous optimization of the semantics is safe.

To prove this claim, we concentrate on the transition relations $\xrightarrow{\Pi}$ and $\xrightarrow{\Psi}$ respectively induced by the Π and Ψ rules only. We wish to show that the order of execution of the processes does not affect the set of states that are reached at synchronization points. On the contrary to [Goo95], we can not rely on the determinism of VHDL to deduce the confluence of $\xrightarrow{\Pi}$. Operators `rnd()` and `lrnd()` make mini-VHDL intrinsically non-determinist. Because of the while loop, the relation $\xrightarrow{\Pi}$ is not necessarily noetherian either.

So, it appears we have no choice but to exploit the independence of processes in their concurrent execution. The intuition is formalized by the two properties that are illustrated in Fig. 3.7 and 3.8. First, consecutive execution steps of two distinct processes can be freely swapped:

Lemma 3.1.

$$\forall s, u, v : s \xrightarrow{\Pi\text{-}i} u \wedge u \xrightarrow{\Pi\text{-}j} v \Rightarrow \exists u' : s \xrightarrow{\Pi\text{-}j} u' \wedge u' \xrightarrow{\Pi\text{-}i} v \; .$$

Second, performing one step of a process does not eliminate the possibility of executing another:

Lemma 3.2.

$$\forall s, s', u : s \xrightarrow{\Pi\text{-}i} s' \wedge s \xrightarrow{\Pi\text{-}j} u \wedge i \neq j \Rightarrow \exists u' : s' \xrightarrow{\Pi\text{-}j} u' \; .$$

The proofs of these two lemmas is immediate by inspection of the various cases that define the operational semantics.

We can now state the theorem that warrants the soundness of the optimized version of our semantics. The maximal execution of the transition relations $\xrightarrow{\Pi}$ and $\xrightarrow{\Psi}$ exactly yield the same set of final[1] states:

Theorem 3.1.

$$\forall s : \{s' \mid s \xrightarrow{\Pi}{}^{!} s'\} = \{s' \mid s \xrightarrow{\Psi}{}^{!} s'\} .$$

Proof. The fact that $\{s' \mid s \xrightarrow{\Psi}{}^{!} s'\} \subseteq \{s' \mid s \xrightarrow{\Pi}{}^{!} s'\}$ is straightforward, since $\xrightarrow{\Psi}$ is a restriction of $\xrightarrow{\Pi}$.

For the converse inclusion, we must establish that if $s \xrightarrow{\Pi}{}^{!} s'$, then $s \xrightarrow{\Psi}{}^{!} s'$. This is proved by induction on the length of the derivation of $\xrightarrow{\Pi}$ from s to s'.

- The base case is trivial, since s is terminal and $s = s'$,
- For the induction step, consider the derivation:

  $$s_0 \xrightarrow{\Pi\text{-}i_0} s_1 \ldots \xrightarrow{\Pi\text{-}i_n} s_{n+1} .$$

  We set j to be the index of the process that would be executed from s_0 with $\xrightarrow{\Psi}$. More precisely, j is such that there exists some state s' and $s_0 \xrightarrow{\Psi\text{-}j} s'$.

  Now, let k be the smallest integer such that $i_k = j$. It necessarily exists. Indeed, if it doesn't then repetitive application of lemma 3.2 allows us to establish that there is s_{n+2} and $s_{n+1} \xrightarrow{\Pi\text{-}j} s_{n+2}$. This is clearly impossible, since s_{n+1} is terminal.

  If we can show that there exists a derivation:

  $$s_0 \xrightarrow{\Pi\text{-}j} s'_1 \ldots \xrightarrow{\Pi} s_{n+1} ,$$

  whose first step is to execute process j, then we can apply the induction hypothesis and conclude that $s_0 \xrightarrow{\Psi}{}^{!} s_{n+1}$.

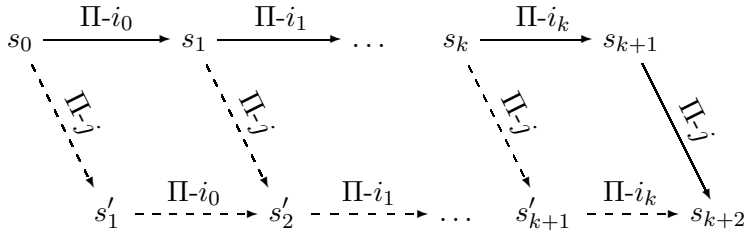
  To build this derivation, the idea is to exchange the Π-j step between state s_k and s_{k+1} with the one occurring between s_0 and s'_1. This is

[1] Notation $s \rightarrow^{!} s'$ stands for $s \rightarrow^{*} s' \wedge s \not\rightarrow$

done by degrees thanks to lemma 3.1. We prove by induction on k that if:

$$\begin{cases} s_0 \xrightarrow{\Pi\text{-}i_0} s_1 \ldots \xrightarrow{\Pi\text{-}i_{k-1}} s_k \wedge \forall l \in [0,k[\colon i_l \neq j \\ s_k \xrightarrow{\Pi\text{-}j} s_{k+1}\ , \end{cases}$$

then, there exists $s_0 \xrightarrow{\Pi\text{-}j} s'_1 \xrightarrow{\Pi\text{-}i_0} s'_2 \ldots \xrightarrow{\Pi\text{-}i_{k-1}} s_{k+1}$. This is trivial for the base case. For the induction step, by lemma (3.1), we can swap the two last transitions. It comes that there is s'_{k+1} such that $s_k \xrightarrow{\Pi\text{-}j} s'_{k+1} \xrightarrow{\Pi\text{-}i_k} s_{k+2}$. We can then apply the induction hypothesis to complete the construction. This argumentation is pictured below:



□

3.11 Discussion

We recapitulate the domains of the semantics definition, the evaluation of expressions, the sequential execution of processes and the formalisation of the simulation algorithm in Fig. 3.9, 3.10, 3.11, 3.12, 3.13 and 3.14.

This formalization of the simulation algorithm is the basis to the verification tool that is developed in the next chapters. The tool is designed so as to be sound with respect to this particular semantics. Hence, we here make the implicit assumption that synthesis tools that translate VHDL descriptions into physical hardware are coherent with the same semantics. We believe this choice to be acceptable since we strived to be consistent with the only official reference of the VHDL language, that is the IEEE standard. We hope that synthesis tools comply with the same interpretation of the standard as us. If this is not the case, then even when verification succeeds, the final hardware can not be certified correct. Let me back up this claim with a small VHDL example:

```
if (x = '0') then
```

```
    y <= '1';
  else
    y <= '0';
  end if;
```

Now, suppose signal `x` is initialized by the simulator with the value `'U'`. Then it is correct to state that `y` takes the value `'0'`. However, this piece of code is probably compiled into a multiplexor gate. So, post-synthesis the signal `x` initially takes one of the value `'0'` or `'1'` which results in `y` being either `'0'` or `'1'` too. A similar argument is made in [Fos02] with a Verilog example.

Gaps between semantics allow bugs with daunting consequences to lurk unnoticed through the design flow. So, in practice, it is of the uttermost importance that the various tools that manipulate the design have a common interpretation of its semantics. To convince oneself this is indeed the case is an orthogonal matter, see for instance [Riv04]. This difficult yet necessary task far exceeds the scope of the thesis. Thus, the convenient assumption that synthesis tools are consistent with our view of semantics, is needed to separate concerns.

Alternatively, we could have introduced a simple ad-hoc hardware description language that would suit our verification goals best. We feel this approach is not realistic as it ignores issues that could be lost in potential over-simplifications. Also, hardware engineers work with VHDL models and use VHDL synthesizers. If we want to demonstrate that integration into existing design practice is feasible, we believe it is necessary to work from the same input. Hopefully, we can confine ourselves to the fraction of the language that is actually used by engineers. To aim at providing a formal semantics for whole VHDL is unrealistic. The formalization would be so complex as to become useless. To our relief, the subset we have identified in actual hardware designs seems reasonable. It lends itself well to concise and relatively simple semantics definitions.

We nevertheless believe the essential characteristics of VHDL are present in our semantics. Hence, the semantics embeds a sufficient amount of complexity so as to be realistic for our feasibility study. Also, minor discrepancies can be corrected later and the design of our verification tool can be readily adapted.

$$l \text{ and } (l, W, b, t) \in \mathcal{P}$$

$$\begin{aligned} \beta &= \text{int} \mid \text{std\_logic} \\ \tau &= \beta \mid \tau[l \ldots u] \end{aligned}$$

$$\begin{cases} @(x, \beta) = \{x\} \\ @(x, \tau[l \ldots u]) = \{a.i \mid a \in @(x, \tau), l \leq i \leq u\} \end{cases}$$

$$\begin{aligned} \mathcal{A}|_{Var} &= \{a \in @(x, \tau) \mid x \in Var, x : \tau\} \\ \mathcal{A}|_{Sig} &= \{a \in @(x, \tau) \mid x \in Sig, x : \tau\} \\ \mathcal{A}|_{Clk} &= \{a \in @(x, \tau) \mid x \in Clk, x : \tau\} \\ \mathcal{A} &= \mathcal{A}|_{Var} \cup \mathcal{A}|_{Sig} \cup \mathcal{A}|^{+}_{Sig} \cup \mathcal{A}|^{-}_{Clk} \cup \mathcal{A}|_{Clk} \cup \mathcal{A}|^{+}_{Clk} \end{aligned}$$

$$\begin{aligned} \mathbb{L} &= \{\texttt{'0'}, \texttt{'1'}, \texttt{'U'}\} \\ \mathbb{V} &= \mathbb{L} \cup \mathbb{Z} \\ \rho &: \mathcal{A} \to \mathbb{V} \end{aligned}$$

Figure 3.9: Control points, types, memory locations, values and environments

$$
\begin{aligned}
[\![\texttt{lnot}]\!](v) &= \begin{cases} \texttt{'0'} & \text{if } v = \texttt{'1'} \\ \texttt{'1'} & \text{if } v = \texttt{'0'} \\ \texttt{'U'} & \text{otherwise} \end{cases} \\
[\![\texttt{land}]\!](v_1, v_2) &= \begin{cases} \texttt{'1'} & \text{if } v_1 = \texttt{'1'} \text{ and } v_2 = \texttt{'1'} \\ \texttt{'0'} & \text{if } v_1 = \texttt{'0'} \text{ or } v_2 = \texttt{'0'} \\ \texttt{'U'} & \text{otherwise} \end{cases} \\
[\![\texttt{lxor}]\!](v_1, v_2) &= \begin{cases} \texttt{'U'} & \text{if } v_1 = \texttt{'U'} \text{ or } v_2 = \texttt{'U'} \\ \texttt{'1'} & \text{else if } v_1 \neq v_2 \\ \texttt{'0'} & \text{otherwise} \end{cases} \\
[\![\texttt{not}]\!](v) &= \neg v \\
[\![\texttt{and}]\!](v_1, v_2) &= v_1 \wedge v_2 \\
[\![\texttt{or}]\!](v_1, v_2) &= v_1 \vee v_2 \\
[\![\texttt{=}]\!](v_1, v_2) &= v_1 = v_2 \\
[\![\texttt{+}]\!](v_1, v_2) &= v_1 + v_2 \\
[\![\texttt{-}]\!](v_1, v_2) &= v_1 - v_2 \\
[\![\texttt{<}]\!](v_1, v_2) &= v_1 < v_2
\end{aligned}
$$

Figure 3.10: Operators

$$\frac{\boldsymbol{x} : \tau}{\rho \vdash_L \boldsymbol{x} \Longrightarrow \boldsymbol{x} : \tau}$$

$$\frac{\rho \vdash_L \mathit{lval} \Longrightarrow a : \tau[l \ldots u] \qquad \rho \vdash \mathit{exp} \Longrightarrow i \qquad l \leq i \leq u}{\rho \vdash_L \mathit{lval}\,\texttt{[}\mathit{exp}\texttt{]} \Longrightarrow a.i : \tau}$$

Figure 3.11: Left values

$$\frac{}{\rho \vdash \texttt{'0'} \Longrightarrow \texttt{'0'}} \qquad \frac{}{\rho \vdash \texttt{'1'} \Longrightarrow \texttt{'1'}} \qquad \frac{}{\rho \vdash \texttt{'U'} \Longrightarrow \texttt{'U'}}$$

$$\frac{}{\rho \vdash n \Longrightarrow n} \qquad \frac{}{\rho \vdash \texttt{true} \Longrightarrow \texttt{true}} \qquad \frac{}{\rho \vdash \texttt{false} \Longrightarrow \texttt{false}}$$

$$\frac{v \in \{\texttt{true}, \texttt{false}\}}{\rho \vdash \texttt{rnd()} \Longrightarrow v} \qquad \frac{v \in \{\texttt{'0'}, \texttt{'1'}\}}{\rho \vdash \texttt{lrnd()} \Longrightarrow v}$$

$$\frac{\rho \vdash_L \mathit{lval} \Longrightarrow a : \tau}{\rho \vdash \mathit{lval} \Longrightarrow \rho(a)}$$

$$\frac{\rho \vdash \mathit{exp} \Longrightarrow v}{\rho \vdash \mathit{op}\ \ \mathit{exp} \Longrightarrow [\![op]\!](v)} \qquad \frac{\rho \vdash \mathit{exp}_1 \Longrightarrow v_1 \qquad \rho \vdash \mathit{exp}_2 \Longrightarrow v_2}{\rho \vdash \mathit{exp}_1\ \ \mathit{op}\ \ \mathit{exp}_2 \Longrightarrow [\![op]\!](v_1, v_2)}$$

$$\frac{v = \begin{cases} \texttt{true} & \text{if } \rho(x^-) = \texttt{'0'} \wedge \rho(x) = \texttt{'1'} \\ \texttt{false} & \text{otherwise} \end{cases}}{\rho \vdash \texttt{rising\_edge(}x\texttt{)} \Longrightarrow v}$$

Figure 3.12: Evaluation of expressions

$$\text{sig}\frac{{}^{l}\texttt{lval <= exp} \qquad \rho \vdash_L \texttt{lval} \Longrightarrow a : \beta \qquad \rho \vdash \texttt{exp} \Longrightarrow v}{(l, \rho) \rightharpoonup (next(l), \rho[a^{+} \leftarrow v])}$$

$$\text{var}\frac{{}^{l}\texttt{lval := exp} \qquad \rho \vdash_L \texttt{lval} \Longrightarrow a : \beta \qquad \rho \vdash \texttt{exp} \Longrightarrow v}{(l, \rho) \rightharpoonup (next(l), \rho[a \leftarrow v])}$$

$$\text{wait}\frac{{}^{l}\texttt{wait on } W \texttt{ until } b \texttt{ for } t \qquad c = (next(l), W, b, t)}{(l, \rho) \rightharpoonup (c, \rho)}$$

$$\text{enter}\frac{{}^{l}\texttt{while } b \texttt{ do } {}^{l'}C\texttt{;}P \qquad \rho \vdash b \Longrightarrow \texttt{true}}{(l, \rho) \rightharpoonup (l', \rho)}$$

$$\text{exit}\frac{{}^{l}\texttt{while } b \texttt{ do } P \qquad \rho \vdash b \Longrightarrow \texttt{false}}{(l, \rho) \rightharpoonup (next(l), \rho)}$$

$$\text{choose}\frac{{}^{l}\Sigma_{i \in [1,n]} b_i \texttt{ -> } {}^{l_i}P_i \qquad \rho \vdash b_j \Longrightarrow \texttt{true}}{(l, \rho) \rightharpoonup (l_j, \rho)}$$

$$\text{show}\frac{{}^{l}\texttt{display(}\boldsymbol{x}\texttt{, ..., }\boldsymbol{x}\texttt{)}}{(l, \rho) \rightharpoonup (next(l), \rho)}$$

$$\frac{\begin{array}{c}{}^{l}\texttt{lvec\_to\_int(}y\texttt{, }n\texttt{, }x\texttt{)} \\ \forall i \in [0, n-1] : \rho(y.i) \in \{\texttt{'0'}, \texttt{'1'}\} \qquad \varepsilon_i = \begin{cases} 1 & \text{if } \rho(y.i) = \texttt{'1'} \\ 0 & \text{otherwise} \end{cases} \\ v = \sum_{i=1}^{i=n} 2^{n-i}\varepsilon_i \end{array}}{(l, \rho) \rightharpoonup (next(l), \rho[x \leftarrow v])}$$

$$\frac{\begin{array}{c}{}^{l}\texttt{int\_to\_lvec(}x\texttt{, }n\texttt{, }y\texttt{)} \\ v = \rho(x) \qquad \forall i \in [1, n] : \varepsilon_i = \begin{cases} \texttt{'1'} & \text{if } (v/2^{n-i})\%2 = 1 \\ \texttt{'0'} & \text{if } (v/2^{n-i})\%2 = 0 \end{cases} \\ \rho' = \rho[y.0 \leftarrow \varepsilon_0] \ldots [y.(n-1) \leftarrow \varepsilon_{n-1}] \end{array}}{(l, \rho) \rightharpoonup (next(l), \rho')}$$

Figure 3.13: Sequential process execution

$$\Psi\text{-}i\frac{\forall j < i : c_j = (l_j, W_j, b_j, t_j) \qquad (c_i, \rho) \rightharpoonup (c_i', \rho')}{(c_1, \ldots, c_i, \ldots, c_n, \rho) \to (c_1, \ldots, c_i', \ldots, c_n, \rho')}$$

$$\Delta\frac{\begin{array}{c}\forall i : c_i = (l_i, W_i, b_i, t_i) \qquad \rho' = update(\rho) \\ \exists i : wake(W_i, b_i, \rho, \rho') \qquad \forall i : c_i' = \begin{cases} l_i & \text{if } wake(W_i, b_i, \rho, \rho') \\ c_i & \text{otherwise} \end{cases}\end{array}}{(c_1, \ldots, c_n, \rho) \to (c_1', \ldots, c_n', \rho')}$$

$$\Theta\frac{\begin{array}{c}\forall i : c_i = (l_i, W_i, b_i, t_i) \qquad \rho' = update(\rho) \qquad \forall i : \neg wake(W_i, b_i, \rho, \rho') \\ \exists i : t_i \neq \infty \\ t = \min\{t_i \neq \infty\} \qquad \forall i : c_i' = \begin{cases} l_i & \text{if } t_i = t \\ (l_i, W_i, b_i, t_i - t) & \text{if } t_i \neq \infty \\ c_i & \text{otherwise} \end{cases}\end{array}}{(c_1, \ldots, c_n, \rho) \to (c_1', \ldots, c_n', \rho')}$$

$$\begin{cases} update(\rho)(a) = \rho(a^+) & a \in \mathcal{A}|_{Sig} \cup \mathcal{A}|_{Clk} \\ update(\rho)(a^-) = \rho(a) & a \in \mathcal{A}|_{Clk} \\ update(\rho)(a) = \rho(a) & \text{otherwise} \end{cases}$$

$$wake(W, b, \rho, \rho') = (\exists x \in W : \rho(x) \neq \rho'(x)) \wedge (\rho' \vdash b \Longrightarrow \texttt{true})$$

Figure 3.14: Simulation algorithm

Chapter 4

Abstract Interpretation

4.1 Collecting semantics

To show the absence of errors in a VHDL design, we need to estimate the states that are reachable from some initial configuration s_0:

$$\mathcal{O} = \{s \mid s_0 \rightarrow^* s\} .$$

This set can be equivalently expressed as the least fixpoint of the continuous function $\mathbb{F}$ on the complete lattice of set of states $(\wp(\Sigma), \emptyset, \Sigma, \cup, \cap)$ where:

$$\mathbb{F}(X) = \{s_0\} \cup \{s' \mid \exists s \in X : s \rightarrow s'\} .$$

Unfortunately, because of the excessive size of the state space, the computation of this fixpoint often turns out to be too expensive.

4.2 Abstract Interpretation

We follow the methodology of abstract interpretation [Cou78, CC77, Cou81, CC92] to design a tool that computes a superset of the reachable states. We proceed in two steps. First, we need to choose our representation for sets of states. In other words, we must pick an abstract domain. In our case, the abstract domain is a complete partial order $(D^\sharp, \sqsubseteq, \bot, \sqcup)$. The meaning of an element of $D^\sharp$ is given, in terms of set of states, thanks to a monotonic concretization function:

$$\gamma : D^\sharp \rightarrow \wp(\Sigma) .$$

The bottom element $\bot$ provides us with an abstraction of the empty set and the abstract join $\sqcup$ computes an upper-bound of two elements in $D^\sharp$.

Second, we systematically derive from $\mathbb{F}$ a monotonic function that operates on the abstract domain:

$$\mathbb{F}^\sharp : D^\sharp \to D^\sharp \,.$$

The abstract operator $\mathbb{F}^\sharp$ must be sound. Namely the application of $\mathbb{F}^\sharp$ yields a result that contains all the states obtained when applying $\mathbb{F}$:

$$\mathbb{F} \circ \gamma \subseteq \gamma \circ \mathbb{F}^\sharp \,.$$

This local soundness condition ensures that the least fixpoint of $\mathbb{F}^\sharp$ is a sound over-approximation of the reachable states:

Theorem 4.1 (Soundness).

$$\mathrm{lfp}_\emptyset \mathbb{F} \subseteq \gamma(\mathrm{lfp}_\bot \mathbb{F}^\sharp) \,.$$

Proof. Let $\mathbb{F}_n$ and $\mathbb{F}^\sharp_n$ respectively be the n^{th} iterate from $\emptyset$ and $\bot$ of $\mathbb{F}$ and $\mathbb{F}^\sharp$. We prove by induction, that for all integer n, $\mathbb{F}_n \subseteq \gamma(\mathbb{F}^\sharp_n)$. The base case is immediate since $\emptyset \subseteq \gamma(\bot)$. Suppose we know $\mathbb{F}_n \subseteq \gamma(\mathbb{F}^\sharp_n)$, then by monotonicity of $\mathbb{F}$, it comes that $\mathbb{F}(\mathbb{F}_n) \subseteq \mathbb{F}(\gamma(\mathbb{F}^\sharp_n))$. We deduce thanks to the local correctness that $\mathbb{F}_{n+1} \subseteq \gamma(\mathbb{F}^\sharp(\mathbb{F}^\sharp_n))$. This ends the induction.

Since γ is monotonic, we may establish that $\gamma(\mathbb{F}^\sharp_n) \subseteq \gamma(\bigsqcup_n \mathbb{F}^\sharp_n)$. So we have that $\mathbb{F}_n \subseteq \gamma(\bigsqcup_n \mathbb{F}^\sharp_n)$. By definition of the least upper bound, it is also true that:

$$\bigcup_n \mathbb{F}_n \subseteq \gamma(\bigsqcup_n \mathbb{F}^\sharp_n) \,.$$

On one hand, since $\mathbb{F}$ is a continuous map on the complete lattice of set of states, by Kleene's theorem [Kle52], it comes that:

$$\mathrm{lfp}_\emptyset \mathbb{F} = \bigcup_n \mathbb{F}_n \,.$$

On the other hand, the constructive version of Tarski's fixpoint theorem [CC79a], guarantees the existence of the least fixpoint of the monotonic map $\mathbb{F}^\sharp$ on the complete partial order $(D^\sharp, \sqsubseteq, \bot, \sqcup)$. By a simple induction one checks that $\bigsqcup_n \mathbb{F}^\sharp_n \sqsubseteq \mathrm{lfp}_\bot \mathbb{F}^\sharp$. By monotonicity of γ, we have:

$$\gamma(\bigsqcup_n \mathbb{F}^\sharp_n) \subseteq \gamma(\mathrm{lfp}_\bot \mathbb{F}^\sharp) \,.$$

We have established the following sequence of inequalities:

$$\mathrm{lfp}_\emptyset \mathbb{F} = \bigcup_n \mathbb{F}_n \subseteq \gamma(\bigsqcup_n \mathbb{F}^\sharp_n) \subseteq \gamma(\mathrm{lfp}_\bot \mathbb{F}^\sharp) \,.$$

We conclude by transitivity. □

When the abstract domain is of finite height, the constructive version of Tarski's theorem [CC79a] yields a naive algorithm to obtain the abstract semantics. It suffices to compute the successive iterates $\bot$, $\mathbb{F}^\sharp(\bot)$, $\mathbb{F}^\sharp(\mathbb{F}^\sharp(\bot))$, ... until stabilization.

4.3 Abstract domain

We suppose we are given an abstract numerical domain to describe sets of environments. A numerical domain is a complete partial order $(\mathcal{N}, \sqsubseteq_\mathcal{N}, \bot_\mathcal{N}, \sqcup_\mathcal{N})$. The meaning of its elements is given in terms of sets of environments by a monotonic concretization function:

$$\gamma_\mathcal{N} : (\mathcal{N}, \sqsubseteq_\mathcal{N}) \longrightarrow (\wp(\mathcal{A} \to \mathbb{V}), \subseteq) \,.$$

The abstract bottom and join operators must be such that:

$$\begin{array}{rcl} \emptyset & = & \gamma_\mathcal{N}(\bot_\mathcal{N}) \\ \gamma_\mathcal{N}(X) \cup \gamma_\mathcal{N}(Y) & \subseteq & \gamma_\mathcal{N}(X \sqcup_\mathcal{N} Y) \,. \end{array}$$

We abstract a set of states by a function Y that maps each tuple of program points to an abstract environment:

$$Y : \mathcal{P}^n \to \mathcal{N} \,.$$

This mapping represents all the states $(c_1, \ldots, c_n, \rho)$ for which the environment ρ satisfies the constraints associated to the program points $(c_1, \ldots, c_n)$:

$$\gamma(Y) = \{(c_1, \ldots, c_n, \rho) \mid \rho \in \gamma_\mathcal{N}(Y(c_1, \ldots, c_n))\} \,.$$

Monotonicity of γ is readily checked. The abstract domain $(\mathcal{P}^n \to \mathcal{N}, \sqsubseteq, \bot, \sqcup)$ is a complete partial order whose operations are the pointwise extensions of the ones on the numerical domain $\mathcal{N}$.

4.4 Abstract semantics transformer

4.4.1 Expressions

There is no way we can reference the future value of signals or the past value of clocks with the current syntax of expressions. In order to have

access to any single memory location, we extend the syntax of left values in the following manner:

$$\begin{array}{rcl} \texttt{lval} & = & \texttt{loc} \\ & | & \texttt{future(loc)} \\ & | & \texttt{past(loc)} \\ \texttt{loc} & = & \texttt{x} \\ & | & \texttt{loc[exp]}\,. \end{array}$$

While the semantics of modifiers `future` and `past` is:

$$\frac{\rho \vdash_L \texttt{loc} \Longrightarrow a : \tau}{\rho \vdash_L \texttt{future(loc)} \Longrightarrow a^+ : \tau}$$

$$\frac{\rho \vdash_L \texttt{loc} \Longrightarrow a : \tau}{\rho \vdash_L \texttt{past(loc)} \Longrightarrow a^- : \tau}$$

We can now safely replace any `rising_edge(x)` by the semantically equivalent:

```
past(x) = '0' and x = '1'.
```

It will also be helpful to generate arbitrary integers, so that we add the integer random generator `irnd()`:

$$\frac{i \in \mathbb{Z}}{\rho \vdash \texttt{irnd()} \Longrightarrow i}$$

4.4.2 Sequential statements

The equations in Fig. 4.1 specify the abstract semantics for each sequential statement in the language. They rely on the existence of a few primitives that manipulate the numerical domain $\mathcal{N}$: assign undertakes assignments, select asserts boolean conditions and get_val returns the value of an expression when possible, Ω otherwise. Each operation must obey a soundness condition:

$$\begin{array}{rcl} \left\{ \rho[a \leftarrow v] \,\middle|\, \begin{array}{c} \rho \in \gamma_{\mathcal{N}}(R) \wedge \rho \vdash_L \texttt{lval} \Longrightarrow a \\ \rho \vdash \texttt{exp} \Longrightarrow a \end{array} \right\} & \subseteq & \gamma_{\mathcal{N}}(\mathsf{assign}_{\texttt{lval} \leftarrow \texttt{exp}}(R)) \\ \{\rho \mid \rho \in \gamma_{\mathcal{N}}(R) \wedge \rho \vdash \texttt{exp} \Longrightarrow \texttt{true}\} & \subseteq & \gamma_{\mathcal{N}}(\mathsf{select}_{\texttt{exp}}(R)) \\ \{v \mid \exists \rho \in \gamma_{\mathcal{N}}(R) \wedge \rho \vdash \texttt{exp} \Longrightarrow v\} & \subseteq & \gamma_{\mathcal{V}}(\mathsf{get\_val}_{\texttt{exp}}(R)) \\ \text{with } \gamma_{\mathcal{V}}(\Omega) & = & \mathbb{V} \\ \gamma_{\mathcal{V}}(v) & = & \{v\}\,. \end{array}$$

$$
\begin{aligned}
[\![^l \texttt{lval <= e}]\!]^\sharp R &= \{(next(l), \mathsf{assign}_{\texttt{future(lval)} \leftarrow \texttt{e}}(R))\} \\
[\![^l \texttt{lval := e}]\!]^\sharp R &= \{(next(l), \mathsf{assign}_{\texttt{lval} \leftarrow \texttt{e}}(R))\} \\
[\![^l \texttt{wait on } W \texttt{ until } b \texttt{ for } t]\!]^\sharp R &= \{(c, R) \mid c = (next(l), W, b, t)\} \\
[\![^l \texttt{while } b \texttt{ do } {}^{l'}C; P]\!]^\sharp R &= \{(l', \mathsf{select}_b(R))\} \\
&\quad \cup \{(next(l), \mathsf{select}_{\texttt{not } b}(R))\} \\
[\![^l \Sigma_{i \in [1,n]} b_i \texttt{ -> } {}^{l_i} P_i]\!]^\sharp R &= \{(l_i, \mathsf{select}_{b_i}(R)) \mid i \in [1,n]\} \\
[\![^l \texttt{display(x, ..., x)}]\!]^\sharp R &= \{(next(l), R)\} \\
[\![^l \texttt{lvec\_to\_int}(y,\ n,\ x)]\!]^\sharp R &= \{(next(l), \mathsf{lvec\_to\_int}_{y,n,x}(R))\} \\
[\![^l \texttt{int\_to\_lvec}(x,\ n,\ y)]\!]^\sharp R &= \{(next(l), \mathsf{int\_to\_lvec}_{x,n,y}(R))\}
\end{aligned}
$$

where:

$$
\mathsf{lvec\_to\_int}_{y,x}(R) = \mathsf{assign}_{x \leftarrow v}(R)
$$

$$
v = \begin{cases}
\displaystyle\sum_{i=0}^{i=n-1} 2^{n-1-i} \varepsilon_i & \text{if } \forall i \in [0, n-1] : \mathsf{get\_val}_{y\texttt{[}i\texttt{]}}(R) \in \{\texttt{'0'}, \texttt{'1'}\} \\
& \text{and } \varepsilon_i = \begin{cases} 1 & \text{if } \mathsf{get\_val}_{y\texttt{[}i\texttt{]}}(R) = \texttt{'1'} \\ 0 & \text{otherwise} \end{cases} \\
\texttt{irnd()} & \text{otherwise}
\end{cases}
$$

and:

$$
\begin{aligned}
\mathsf{int\_to\_lvec}_{x,y}(R) &= \bigodot_{i=0}^{i=n-1} \mathsf{assign}_{y\texttt{[}i\texttt{]} \leftarrow \varepsilon_i}(R) \\
v &= \mathsf{get\_val}_x(R) \\
\forall i \in [1,n] : \varepsilon_i &= \begin{cases} \texttt{lrnd()} & \text{if } v \notin \mathbb{Z} \\ \texttt{'1'} & \text{if } (v/2^{n-i})\%2 = 1 \\ \texttt{'0'} & \text{if } (v/2^{n-i})\%2 = 0 \end{cases}
\end{aligned}
$$

Figure 4.1: Equations for the abstract sequential execution

The equations mimic all the possible executions of the sequential statements in the abstract domain. Note that the abstract conversion functions are not very precise. For instance, the conversion `lvec_to_int(`y`,` n`,` x`)` is performed only when every element of the array y is known exactly. In all other cases, the conversion statement is handled as if it assigned any random integer value to the target x. It is likewise for `int_to_lvec(`x`,` n`,` y`)`, where all cells of the array y must be assigned in turn.

As long as the previous conditions hold about the numerical operators, we can show that the abstract semantics $[\![^l C]\!]^\sharp$ is sound:

Proposition 4.1. *If $\rho \in \gamma_{\mathcal{N}}(R)$ and $(l, \rho) \rightharpoonup (c', \rho')$ then there exists R' such that $(c', R') \in [\![^l C]\!]^\sharp R$ and $\rho' \in \gamma_{\mathcal{N}}(R')$.*

Proof. Assume $\rho \in \gamma_{\mathcal{N}}(R)$ and $(l, \rho) \rightharpoonup (c', \rho')$. We examine all cases depending on the statement that is labeled with l:

- **Assignments.** If the statement is of the form ^{l} *`lval <= exp`*, then necessarily:

  $$c' = next(l)\,;$$

  and

  $$\exists a, v : \rho' = \rho[a^+ \leftarrow v] \wedge \rho \vdash_L \textit{lval} \Longrightarrow a \wedge \rho \vdash \textit{exp} \Longrightarrow v\,.$$

  Thanks to the semantics of the modifier `future` the later property is equivalent to:

  $$\exists a, v : \rho' = \rho[a \leftarrow v] \wedge \rho \vdash_L \texttt{future(}\textit{lval}\texttt{)} \Longrightarrow a \wedge \rho \vdash \textit{exp} \Longrightarrow v\,.$$

  From the soundness condition imposed on the operator assign, it immediately comes that:

  $$\rho' \in \gamma_{\mathcal{N}}(\mathsf{assign}_{\texttt{future(}\textit{lval}\texttt{)} \leftarrow \textit{exp}}(R))\,.$$

  This concludes, since we have:

  $$[\![^l \textit{lval} \texttt{ <= } \textit{exp}]\!]^\sharp R = \{(next(l), \mathsf{assign}_{\texttt{future(}\textit{lval}\texttt{)} \leftarrow \textit{exp}}(R))\}\,.$$

  Variable assignment is even more straightforward.

- **Suspension.** When $C = {}^l\texttt{wait on } W \texttt{ until } b \texttt{ for } t$, then

  $$[\![C]\!]^\sharp R = \{(c, R) \mid c = (next(l), W, b, t)\}\,.$$

The semantics of $\rightharpoonup$ is defined such that:

$$c' = (next(l), W, b, t) \,,$$

and the environment is kept unmodified:

$$\rho' = \rho \,.$$

So in this case soundness is trivial.

- **Control.** If $C = {}^{l}\Sigma_{i\in[1,n]} b_i$ `->` ${}^{l_i}P_i$, then there exists i for which:

$$c' = l_i \wedge \rho' = \rho$$
$$\rho \vdash b_i \Longrightarrow \texttt{true} \,.$$

Thanks to the soundness condition put on select we have:

$$\rho' \in \mathsf{select}_{b_i}(R) \,.$$

This ends the proof for this case, since:

$$[\![C]\!]^\sharp R = \{(l_i, \mathsf{select}_{b_i}(R)) \mid i \in [1, n]\} \,.$$

Soundness of while statements is performed in an almost identical way.

- **Conversions.** Consider ^{l}`lvec_to_int(`y`,` n`,` x`)`. Necessarily,

$$\rho' = \rho[x \leftarrow v]$$
$$\forall i \in [0, n-1] : \rho(y.i) \in \{\texttt{'0'}, \texttt{'1'}\} \,,$$

where

$$\begin{aligned} v &= \sum_{i=1}^{i=n} 2^{n-i}\varepsilon_i \\ \varepsilon_i &= \begin{cases} 1 & \text{if } \rho(y.i) = \texttt{'1'} \\ 0 & \text{otherwise} \,. \end{cases} \end{aligned}$$

There are two possibilities:

  - Suppose first that for all $i \in [0, n-1]$:

$$\mathsf{get\_val}_{y[i]}(R) \in \{\texttt{'0'}, \texttt{'1'}\} \,.$$

So, from the soundness condition of get_val, it comes that:

$$\{w \mid \exists \varrho \in \gamma_{\mathcal{N}}(R) \wedge \varrho \vdash y\texttt{[}i\texttt{]} \Longrightarrow w\} \subseteq \{\mathsf{get\_val}_{y\texttt{[}i\texttt{]}}(R)\} .$$

Since, $\rho \in \gamma_{\mathcal{N}}(R)$, this means in particular that:

$$\{w \mid \rho \vdash y\texttt{[}i\texttt{]} \Longrightarrow w\} \subseteq \{\mathsf{get\_val}_{y\texttt{[}i\texttt{]}}(R)\} .$$

In other words, by definition of the evaluation of expressions, it must be the case that:

$$\rho(y.i) = \mathsf{get\_val}_{y\texttt{[}i\texttt{]}}(R) .$$

This implies that:

$$\begin{aligned} v &= v' \\ \text{where } v' &= \sum_{i=1}^{i=n} 2^{n-i}\varepsilon'_i \\ \text{and } \varepsilon'_i &= \begin{cases} 1 & \text{if } \mathsf{get\_val}_{y\texttt{[}i\texttt{]}}(R) = \texttt{'1'} \\ 0 & \text{otherwise} . \end{cases} \end{aligned}$$

By soundness of the abstract assignment, it comes that:

$$\rho' \in \gamma_{\mathcal{N}}(\mathsf{assign}_{x \leftarrow v'}(R)) .$$

Which ends the proof of this case.

– If there is an i for which:

$$\mathsf{get\_val}_{y\texttt{[}i\texttt{]}}(R) \notin \{\texttt{'0'}, \texttt{'1'}\} ,$$

then $[\![^{l}\texttt{lvec\_to\_int(}y\texttt{, }n\texttt{, }x\texttt{)}]\!]^{\sharp}R$ is equivalent to:

$$\mathsf{assign}_{x \leftarrow \texttt{irnd()}}(R) .$$

By definition of the evaluation of expressions, we have:

$$v \in \{w \mid \rho \vdash \texttt{irdn()} \Longrightarrow w\} .$$

It follows, by soundness of the abstract assignment, that:

$$\rho' \in \gamma_{\mathcal{N}}(\mathsf{assign}_{x \leftarrow \texttt{irnd()}}(R)) .$$

The case of $^{l}\texttt{int\_to\_lvec(}x\texttt{, }n\texttt{, }y\texttt{)}$ follows a similar thread of reasoning.

□

$$\Psi^\sharp\text{-}i\frac{\forall j < i : c_j = (l_j, W_j, b_j, t_j) \qquad c_i = l_i \qquad (c_i', R') \in [\![^{l_i} C]\!]^\sharp R}{(c_1, \ldots, c_i, \ldots, c_n, R) \rightsquigarrow (c_1, \ldots, c_i', \ldots, c_n, R')}$$

$$\Delta^\sharp\frac{\begin{array}{c}\forall i : c_i = (l_i, W_i, b_i, t_i) \\ (c_1', \ldots, c_n', R') \in \mathsf{explore}(c_1, \ldots, c_n, R) \qquad \exists i : c_i' \neq c_i\end{array}}{(c_1, \ldots, c_n, R) \rightsquigarrow (c_1', \ldots, c_n', \mathsf{update}(R'))}$$

$$\Theta^\sharp\frac{\begin{array}{c}\forall i : c_i = (l_i, W_i, b_i, t_i) \qquad (c_1, \ldots, c_n, R') \in \mathsf{explore}(c_1, \ldots, c_n, R) \\ \begin{array}{cc}\exists i : t_i \neq \infty \\ t = \min\{t_i \neq \infty\}\end{array} \qquad \forall i : c_i' = \begin{cases} l_i & \text{if } t_i = t \\ (l_i, W_i, b_i, t_i - t) & \text{if } t_i \neq \infty \\ c_i & \text{otherwise}\end{cases}\end{array}}{(c_1, \ldots, c_n, R) \rightsquigarrow (c_1', \ldots, c_n', \mathsf{update}(R'))}$$

Figure 4.2: Abstract simulation semantics

4.4.3 Abstract simulation algorithm

We systematically derive from the simulation algorithm, the abstract transition relation $\rightsquigarrow$ found in Fig. 4.2. The first rule uses the equations previously introduced to run processes concurrently. The synchronization rules $\Delta^\sharp$ and $\Theta^\sharp$ are slightly more complex. First, explore generates all the possible awakenings (and lack thereof) of processes according to the evaluation of predicate *wake*. Then, update modifies the abstract environment to reflect the memory update that occurs during synchronization.

We begin with the exact specification of operator update:

$$\begin{aligned}\mathsf{update}(R) &= \mathsf{update\_present}(\mathsf{update\_past}(R)) \\ \mathsf{update\_past}(R) &= \bigodot_{a \in \mathcal{A}_{|Clk}} \mathsf{assign}_{\mathsf{past}(\mathsf{lval}(a)) \leftarrow \mathsf{lval}(a)}(R) \\ \mathsf{update\_present}(R) &= \bigodot_{a \in \mathcal{A}_{|Clk} \cup \mathcal{A}_{|Sig}} \mathsf{assign}_{\mathsf{lval}(a) \leftarrow \mathtt{future}(\mathsf{lval}(a))}(R)\,.\end{aligned}$$

In the previous definition lval inputs an address and returns a left value that always evaluate to this address. It is trivial to check that update is indeed sound with respect to its concrete counterpart:

Lemma 4.1.

$$\{update(\rho) \mid \rho \in \gamma_{\mathcal{N}}(R)\} \subseteq \gamma_{\mathcal{N}}(\mathsf{update}(R))\,.$$

Recall that the predicate $wake(W, b, \rho, \rho')$ which governs the resumption of processes is composed of two clauses:

$$\exists x \in W : \rho(x) \neq \rho'(x)$$
$$\rho' \vdash b \Longrightarrow \texttt{true}\,.$$

Remark that when ρ' is equal to $update(\rho)$, the first clause boils down to:

$$\exists x \in W : \rho(x) \neq \rho(x^+)\,.$$

It is possible to build an expression $\mathsf{sensitive}(W)$ that evaluates to `true` if and only if the previous assertion is true:

$$\begin{aligned}
\mathsf{event}(x) &= \texttt{not (}\mathsf{lval}(x)\texttt{ = future(}\mathsf{lval}(x)\texttt{))}\\
\mathsf{sensitive}(\emptyset) &= \texttt{false}\\
\mathsf{sensitive}(\{x\} \cup W) &= \mathsf{event}(x)\ \texttt{or}\ \mathsf{sensitive}(W)\,.
\end{aligned}$$

Lemma 4.2. *For all environment ρ and $W \subseteq \mathcal{A}|_{Sig} \cup \mathcal{A}|_{Clk}$, are equivalent:*

$$\exists x \in W : \rho(x) \neq update(\rho(x))$$
$$\rho \vdash \mathsf{sensitive}(W) \Longrightarrow \texttt{true}\,.$$

We introduce the transformation delay. It shifts any memory reference that appears in an expression by one step into the future:

$$\begin{aligned}
\mathsf{delay}(x\texttt{[}e_1\texttt{]}\ldots\texttt{[}e_n\texttt{]}) &= \begin{cases}\texttt{future(}x\texttt{[}e_1\texttt{]}\ldots\texttt{[}e_n\texttt{])} & x \in Sig \cup Clk\\ x\texttt{[}e_1\texttt{]}\ldots\texttt{[}e_n\texttt{]} & \text{otherwise}\end{cases}\\
\mathsf{delay}(\texttt{past(}e\texttt{)}) &= e\\
\mathsf{delay}(\texttt{op}\ \ e) &= \texttt{op}\ \ \mathsf{delay}(e)\\
\mathsf{delay}(e_1\ \ \texttt{op}\ \ e_2) &= \mathsf{delay}(e_1)\ \ \texttt{op}\ \ \mathsf{delay}(e_2)\\
\text{otherwise } \mathsf{delay}(e) &= e\,.
\end{aligned}$$

For the sake of simplicity, we assume the expressions $e_1, \ldots, e_n$ used as array indices in $x[e_1]\ldots[e_n]$ do not contain any reference to a signal. As long as expression b does not contain any reference to the future value of a signal, it is easy to show that evaluating $\mathsf{delay}(b)$ in ρ amounts to evaluating b in $update(\rho)$:

Lemma 4.3. *For all environment ρ, value v and expression b. If b has no sub-expression of the form* `future(`e`)`*, then the following assertions are equivalent:*

$$update(\rho) \vdash b \Longrightarrow v$$
$$\rho \vdash \mathsf{delay}(b) \Longrightarrow v\,.$$

Proof. Straightforward by structural induction on expression b. □

Hence, we are able to compile (W, b) into the following expression that evaluates to `true` in an environment ρ if and only if $wake(W, b, \rho, update(\rho))$ is true:

$$\mathsf{wake}(W, b) = \mathsf{sensitive}(W) \ \texttt{and} \ \mathsf{delay}(b) \, .$$

Lemma 4.4. *For all environment ρ, $W \subseteq \mathcal{A}|_{Sig} \cup \mathcal{A}|_{Clk}$ and expression b, the following propositions are equivalent:*

$$\rho \vdash \mathsf{wake}(W, b) \Longrightarrow \texttt{true}$$
$$wake(W, b, \rho, update(\rho)) \, .$$

Proof. Follows from lemma 4.2 and lemma 4.3. □

We now have all the elements to properly define the operation explore:

$$\begin{aligned}
\mathsf{explore}(\vec{c}, \bot_{\mathcal{N}}) &= \emptyset \\
\mathsf{explore}(R) &= \{R\} \\
\mathsf{explore}(c_1, \vec{c}, R) &= \\
\{(c_1, c_2', \ldots c_n', R') \mid (c_2', \ldots c_n', R') &\in \mathsf{explore}(\vec{c}, \mathsf{select}_{\texttt{not (}\mathsf{wake}(W_1, b_1)\texttt{)}}(R))\} \\
\cup \{(l_1, c_2', \ldots c_n', R') \mid (c_2', \ldots c_n', R') &\in \mathsf{explore}(\vec{c}, \mathsf{select}_{(\mathsf{wake}(W_1, b_1))}(R))\} \\
&\text{where } (l_1, W_1, b_1, _) = c_1 \, .
\end{aligned}$$

Essentially, explore constrains the abstract environment according to which processes wake up. For every control point (l, W, b, t), two options are examined: either $\mathsf{wake}(W, b)$ is satisfied and the control point becomes l, or if `not` $(\mathsf{wake}(W, b))$ holds, then the control point remains unmodified. To reduce the cost of this operation, we stop the exploration as soon as the abstract environment equals to $\bot_{\mathcal{N}}$. It is sound to do so because $\gamma_{\mathcal{N}}$ is $\bot$-strict, i.e. $\gamma_{\mathcal{N}}(\bot_{\mathcal{N}}) = \emptyset$. The soundness of explore is set forth as follows:

Lemma 4.5. *If $\rho \in \gamma_{\mathcal{N}}(R)$ and*

$$\forall i : c_i' = \begin{cases} l_i & wake(W_i, b_i, \rho, update(\rho)) \\ c_i & otherwise \, , \end{cases}$$

then, there exists R' such that $(c_1', \ldots, c_n', R') \in \mathsf{explore}(c_1, \ldots, c_n, R)$ and $\rho \in \gamma_{\mathcal{N}}(R')$.

Proof. First, remark that, when $R = \bot_{\mathcal{N}}$, the implication immediately holds on a false premise because $\gamma_{\mathcal{N}}(\bot_{\mathcal{N}}) = \emptyset$. For the rest of the proof, we proceed by induction on the number n of control points:

- If $n = 0$, then $\mathsf{explore}(R) = R$. Thus, assuming that $\rho \in \gamma_{\mathcal{N}}(R)$, it is trivial to establish that $\rho \in \gamma_{\mathcal{N}}(\mathsf{explore}(R))$.
- For the inductive case, we assume the property is true for n.

  Suppose first that $c'_1 = l_1$ and so $wake(W_1, b_1, \rho, update(\rho))$ is true. This implies by lemma 4.4 that:

  $$\rho \vdash \mathsf{wake}(W_1, b_1) \Longrightarrow \texttt{true}\ .$$

  From the soundness condition imposed on select, we deduce that:

  $$\rho \in \gamma_{\mathcal{N}}(\mathsf{select}_{\mathsf{wake}(W_1,b_1)}(R))\ .$$

  We apply the induction hypothesis and it yields R' such that:

  $$\begin{cases} (c'_2, \ldots, c'_{n+1}, R') \in \mathsf{explore}(c_2, \ldots, c_{n+1}, \mathsf{select}_{\mathsf{wake}(W_1,b_1)}(R)) \\ \rho \in \gamma_{\mathcal{N}}(R')\ . \end{cases}$$

  By definition, $(l_1, c'_2, \ldots, c'_{n+1}, R') \in \mathsf{explore}(c_1, c_2, \ldots, c_{n+1}, R)$.

  The case when $c'_1 = c_1$ is almost identical.

□

To increase the precision of $\mathsf{explore}$, we propagate the constraints that the environment must satisfy through the recursive calls:

$$\begin{array}{rcl} \mathsf{gen}(\vec{c}, \bot_{\mathcal{N}}, e) & = & \emptyset \\ \mathsf{gen}(R, e) & = & \{R\} \\ \mathsf{gen}(c_1, \vec{c}, R, e) & = & \{c_1, c'_2, \ldots c'_n, R') \mid (c'_2, \ldots c'_n, R') \in \mathsf{gen}(\vec{c}, \mathsf{select}_{\tilde{b}}(R), \tilde{b})\} \\ & \cup & \{l_1, c'_2, \ldots c'_n, R') \mid (c'_2, \ldots c'_n, R') \in \mathsf{gen}(\vec{c}, \mathsf{select}_{b}(R), b)\} \\ & & \text{where } \begin{cases} (l_1, W_1, b_1, \_) = c_1 \\ \tilde{b} = e \texttt{ and not (}\mathsf{wake}(W_1, b_1)\texttt{)} \\ b = e \texttt{ and } \mathsf{wake}(W_1, b_1)\ . \end{cases} \end{array}$$

The soundness proof of $\mathsf{explore}$ as defined below is almost identical as before and is therefore not included.

$$\mathsf{explore}(c_1, \ldots, c_n, R) = \mathsf{gen}(c_1, \ldots, c_n, R, \texttt{true})\ .$$

Soundness of the whole abstract simulation algorithm $\rightsquigarrow$ ensues:

Proposition 4.2. *If $\rho \in \gamma_{\mathcal{N}}(R)$ and $(c, \rho) \to (c', \rho')$, then there exists R' such that $(c, R) \rightsquigarrow (c', R')$ and $\rho' \in \gamma_{\mathcal{N}}(R')$.*

Proof. Suppose $\rho \in \gamma_{\mathcal{N}}(R)$ and $(c, \rho) \to (c', \rho')$ We examine all the possible cases in turn. They depend on the rule that was used to derive (c', ρ'):

- Case Ψ-i: we know $(c_i, \rho) \rightharpoonup (c_i', \rho')$. We apply proposition 4.1 and it comes that there exists R' such that $(c_i', R') \in [\![^{l_i} C]\!]^\sharp R$ and $\rho' \in \gamma_{\mathcal{N}}(R)$. It is then straightforward to construct a derivation with rule $\Psi^\sharp$-i.

- Case Δ: We have that $\rho' = \mathit{update}(\rho)$ and

$$\forall i : c_i' = \begin{cases} l_i & \mathit{wake}(W_i, b_i, \rho, \rho') \\ c_i & \text{otherwise}\,. \end{cases}$$

  Then, by application of lemma 4.5, it comes that there exists R' such that $(c', R') \in \mathsf{explore}(c, R)$ and $\rho \in \gamma_{\mathcal{N}}(R')$. Since, by lemma 4.1, $\rho' \in \gamma_{\mathcal{N}}(\mathsf{update}(R'))$ this case succeeds.

- Case Θ: this is fairly similar to the previous case.

□

4.4.4 Semantics transformer

The abstract representation of the initial configuration $s_0 = (l_1, \ldots, l_n, \rho_0)$ is:

$$X_0(c_1, \ldots, c_n) = \begin{cases} \mathsf{singleton}(\rho_0) & \forall i : c_i = l_i \\ \bot_{\mathcal{N}} & \text{otherwise}\,. \end{cases}$$

The additional operator $\mathsf{singleton}$ yields an abstract environment that is a sound approximation of a concrete environment. It must thus obey the soundness condition:

$$\rho \in \gamma_{\mathcal{N}}(\mathsf{singleton}(\rho))\,.$$

We complete the theoretical design of the static analysis with the abstract counterpart of the semantics transformer $\mathbb{F}$:

$$\mathbb{F}^\sharp(X)(c') = X_0(c') \sqcup_{\mathcal{N}} \bigsqcup\nolimits_{\mathcal{N}} \{R' \mid \exists (c, R) : X(c) = R \wedge (c, R) \rightsquigarrow (c', R')\}\,.$$

The monotonicity of $\mathbb{F}^\sharp$ is readily checked. Besides, the properties enforced on the basic operators ensure its soundness:

Proposition 4.3 (Soundness).

$$\mathbb{F}(\gamma(X)) \subseteq \gamma(\mathbb{F}^\sharp(X)) \,.$$

Proof. First, we show that:

$$\{s_0\} \subseteq \{(c', \rho') \mid \rho' \in \gamma_{\mathcal{N}}(X_0(c'))\} \,. \tag{4.1}$$

Assuming that the initial configuration s_0 is equal to $(l_1, \ldots, l_n, \rho_0)$, then $X_0(l_1, \ldots, l_n) = \mathsf{singleton}(\rho_0)$. From the soundness condition of $\mathsf{singleton}$, the validity of equation (4.1) is ensured, since:

$$\rho_0 \in \gamma_{\mathcal{N}}(\mathsf{singleton}(\rho_0)) \,.$$

Second, we need to establish that:

$$\begin{aligned} &\{(c', \rho') \mid \exists (c, \rho) : \rho \in \gamma_{\mathcal{N}}(X(c)) \wedge (c, \rho) \to (c', \rho')\} \\ &\quad \subseteq \{(c', \rho') \mid \exists R' : \exists (c, R) : R = X(c) \wedge (c, R) \rightsquigarrow (c', R') \wedge \rho' \in \gamma_{\mathcal{N}}(R')\} \,. \end{aligned} \tag{4.2}$$

Assume (c', ρ') belongs to the set on the left-hand side of equation (4.2). Then, we know there is (c, ρ) such that:

$$\rho \in \gamma_{\mathcal{N}}(X(c)) \wedge (c, \rho) \to (c', \rho') \,.$$

By proposition 4.2, it comes that there exists R' such that:

$$(c, X(c)) \rightsquigarrow (c', R') \wedge \rho' \in \gamma_{\mathcal{N}}(R') \,.$$

This shows (c', ρ') also belongs to the set on the right-hand side of (4.2).

Putting (4.1) and (4.2) together, we get:

$$\begin{aligned} &\{s_0\} \cup \{(c', \rho') \mid \exists (c, \rho) : \rho \in \gamma_{\mathcal{N}}(X(c)) \wedge (c, \rho) \to (c', \rho')\} \subseteq \\ &\qquad \{(c', \rho') \mid \rho' \in \gamma_{\mathcal{N}}(X_0(c'))\} \\ &\cup \{(c', \rho') \mid \exists R' : \exists (c, R) : R = X(c) \wedge (c, R) \rightsquigarrow (c', R') \wedge \rho' \in \gamma_{\mathcal{N}}(R')\} \,. \end{aligned} \tag{4.3}$$

On one hand we have:

$$\begin{aligned} & \{s_0\} \cup \{(c', \rho') \mid \exists (c, \rho) : \rho \in \gamma_{\mathcal{N}}(X(c)) \wedge (c, \rho) \to (c', \rho')\} \\ = \;& \{s_0\} \cup \{(c', \rho') \mid \exists (c, \rho) : (c, \rho) \in \gamma(X) \wedge (c, \rho) \to (c', \rho')\} \\ = \;& \mathbb{F}(\gamma(X)) \,. \end{aligned}$$

On the other hand, it comes that:

$$\begin{aligned} & \{(c',\rho') \mid \exists R' : \exists (c,R) : R = X(c) \wedge (c,R) \rightsquigarrow (c',R') \wedge \rho' \in \gamma_{\mathcal{N}}(R')\} \\ = \; & \{(c',\rho') \mid \rho' \in \bigcup \{\gamma_{\mathcal{N}}(R') \mid \exists (c,R) : R = X(c) \wedge (c,R) \rightsquigarrow (c',R')\}\} \\ \subseteq \; & \left\{(c',\rho') \mid \rho' \in \gamma_{\mathcal{N}}\left(\bigsqcup\nolimits_{\mathcal{N}} \{R' \mid \exists c : (c,X(c)) \rightsquigarrow (c',R')\}\right)\right\} . \end{aligned}$$

We then have:

$$\begin{aligned} & \{(c',\rho') \mid \rho' \in \gamma_{\mathcal{N}}(X_0(c'))\} \cup \\ & \left\{(c',\rho') \mid \rho' \in \gamma_{\mathcal{N}}\left(\bigsqcup\nolimits_{\mathcal{N}} \{R' \mid \exists c : (c,X(c)) \rightsquigarrow (c',R')\}\right)\right\} \\ \subseteq \; & \{(c',\rho') \mid \rho' \in \gamma_{\mathcal{N}}(\mathbb{F}^\sharp(X)(c'))\} \\ = \; & \gamma(\mathbb{F}^\sharp(X)) . \end{aligned}$$

From these remarks we may deduce that (4.3) implies:

$$\mathbb{F}(\gamma(X)) \subseteq \gamma(\mathbb{F}^\sharp(X)) .$$

□

4.5 Example

As an illustration, we show how to perform the verification on the small VHDL description introduced in example 3.1 of section 3.5. We recall this piece of code below:

```
  [1]forever do
    [2]clk <= lnot clk;
    [3]wait on {} for 1;
|
  [4]forever do
    [5]if clk = '1' then [6]o <= x land (lnot y);
    [7]wait on {clk} for ever;
|
  [8]x <= '0'; [9]y <= '1';
  [10]wait on clk for ever;
  [11]forever do
    [12]x <= lrnd();
    [13]display(o);
    [14]wait on {clk} for ever;
```

We would like to check that the `display` instruction at label 13 can never print `'1'` out. To perform this automatically, we use the generic static analysis described in the previous section to compute a superset of the reachable states. We need to instantiate the analysis with a particular numerical domain so that:

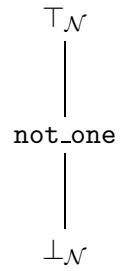
- the computation is as efficient as possible,
- the result is precise enough and allows to check the property.

The literature about numerical domains is vast, see [Kil73, CC77, CH78, Min01, Min02, Fer01, Kar76, Gra91, SKH02, MC04, HK01, CKZR02, GB03, GB04, SSM04, RCK04, BT00, Fer04, Mau94]. We could encode environments using BDDs [Bry86]. This would without any doubt produce accurate information about the behaviour of the circuit. However, this would clearly be an overkill: it is unnecessarily expensive for the purpose of showing that signal `o` is never `'1'`.

We would like to choose a non-relational domain. Non-relational domains do not track the relationships between the various variables of the environment. As such, their computational costs and memory requirements are low. An abstract element is simply a vector R, that maps each variable x to an approximation $R(x)$ of the values it may take:

$$\gamma_{\mathcal{N}}(R) = \{\rho \mid \forall x \in \mathcal{A} : \rho(x) \in \gamma_{\mathbb{V}}(R(x))\} \,.$$

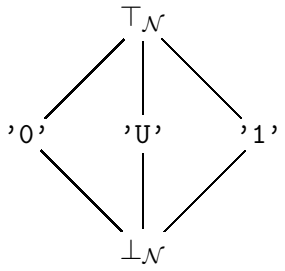
In the previous equation, the $\gamma_{\mathbb{V}}$ is the concretization function of the abstract domain $\mathbb{V}^\sharp$ necessary to approximate sets of values. At first, since we want to show that signal `o` is never equal to `'1'`, we could set $\mathbb{V}^\sharp$ to be:



$$\begin{aligned} \gamma_{\mathbb{V}}(\top^{\mathcal{N}}) &= \mathbb{V} \\ \gamma_{\mathbb{V}}(\texttt{not\_one}) &= \mathbb{V} \setminus \{\texttt{'1'}\} \\ \gamma_{\mathbb{V}}(\bot_{\mathcal{N}}) &= \emptyset \,. \end{aligned}$$

Obviously this domain is not precise enough. Inevitably, to establish the desired property, it is necessary to check that the expression `x land (lnot y)` always evaluates to `'0'`. Hence, we need the fact that signal `y` is always `'1'`. So we adopt the domain of constants. The domain of constants was

originally used in optimizing compilers [Kil73] to suppress superfluous code. The corresponding Hasse diagram and concretization function are as follows:



$$\begin{aligned}
\gamma_{\mathbb{V}}(\top^{\mathcal{N}}) &= \mathbb{V} \\
\gamma_{\mathbb{V}}(\texttt{'0'}) &= \{\texttt{'0'}\} \\
\gamma_{\mathbb{V}}(\texttt{'U'}) &= \{\texttt{'U'}\} \\
\gamma_{\mathbb{V}}(\texttt{'1'}) &= \{\texttt{'1'}\} \\
\gamma_{\mathbb{V}}(\bot_{\mathcal{N}}) &= \emptyset\,.
\end{aligned}$$

The operation of the domain of constant are standard: see for instance [Cou99] and the associated implementation for a thorough description. We run the analysis, starting from an initial state where all signals are set to '0'. The table below shows the successive steps of computation. Each step adds a new abstract environment which is merged with the information collected so far.

c_1	c_2	c_3	clk/clk$^+$	x/x$^+$	y/y$^+$	o/o$^+$
1	4	8	'0'/'0'	'0'/'0'	'0'/'0'	'0'/'0'
2	4	8	'0'/'0'	'0'/'0'	'0'/'0'	'0'/'0'
3	4	8	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	4	8	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	5	8	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	7	8	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	8	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	9	'0'/'1'	'0'/'0'	'0'/'0'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	10	'0'/'1'	'0'/'0'	'0'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11, {clk}, ∞	'0'/'1'	'0'/'0'	'0'/'1'	'0'/'0'
1, $\emptyset$, 1	4	11	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	5	11	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	6	11	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	7	11	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	12	'1'/'1'	'0'/'0'	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	13	'1'/'1'	'0'/$\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	14	'1'/'1'	'0'/$\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11, {clk}, ∞	$\top$/'1'	'0'/$\top$	$\top$/'1'	'0'/'0'

We stop at this point to underscore the difference between the two versions of the operator explore. We let c_1, c_2 and c_3 respectively be $(1, \emptyset, 1)$,

$(4, \{\texttt{clk}\}, \infty)$ and $(11, \{\texttt{clk}\}, \infty)$. We also define $Q = R[\texttt{clk} \leftarrow \texttt{'1'}]$ and clk_evt to be not (clk = future(clk)). The first version of explore produces the following results:

$$
\begin{aligned}
& \mathsf{explore}(c_1, c_2, c_3, R) \\
= \; & \{(c_1, c_2', c_3', R') \mid (c_2', c_3', R') \in \mathsf{explore}(c_2, c_3, \mathsf{select}_{\texttt{true}}(R))\} \\
& \cup \{(1, c_2', c_3', R') \mid (c_2', c_3', R') \in \mathsf{explore}(c_2, c_3, \mathsf{select}_{\texttt{false}}(R))\} \\
= \; & \{(c_1, c_2, c_3', R') \mid (c_3', R') \in \mathsf{explore}(c_3, \mathsf{select}_{\texttt{not clk\_evt}}(R))\} \\
& \cup \{(c_1, 4, c_3', R') \mid (c_3', R') \in \mathsf{explore}(c_3, \mathsf{select}_{\texttt{clk\_evt}}(R))\} \\
= \; & \{(c_1, c_2, c_3, R') \mid R' \in \mathsf{explore}(c_3, \mathsf{select}_{\texttt{not (clk\_evt)}}(Q))\} \\
& \cup \{(c_1, c_2, 11, R') \mid R' \in \mathsf{explore}(\mathsf{select}_{\texttt{clk\_evt}}(Q))\} \\
& \cup \{(c_1, 4, c_3, R') \mid R' \in \mathsf{explore}(\mathsf{select}_{\texttt{not clk\_evt}}(R))\} \\
& \cup \{(c_1, 4, 11, R') \mid R' \in \mathsf{select}_{\texttt{clk\_evt}}(R)\} \\
= \; & \{(c_1, c_2, c_3, Q), (c_1, 4, c_3, R), (c_1, 4, 11, R)\} \, .
\end{aligned}
$$

Whereas the second version yields[1]:

$$
\begin{aligned}
& \mathsf{explore}(c_1, c_2, c_3, R) \\
= \; & \mathsf{gen}(c_1, c_2, c_3, R, \texttt{true}) \\
= \; & \{(c_1, c_2', c_3', R') \mid (c_2', c_3', R') \in \mathsf{gen}(c_2, c_3, \mathsf{select}_{\texttt{true}}(R), \texttt{true})\} \\
& \cup \{(1, c_2', c_3', R') \mid (c_2', c_3', R') \in \mathsf{gen}(c_2, c_3, \mathsf{select}_{\texttt{false}}(R), \texttt{false})\} \\
= \; & \{(c_1, c_2, c_3', R') \mid (c_3', R') \in \mathsf{gen}(c_3, \mathsf{select}_{\texttt{not clk\_evt}}(R), \texttt{not clk\_evt})\} \\
& \cup \{(c_1, 4, c_3', R') \mid (c_3', R') \in \mathsf{gen}(c_3, \mathsf{select}_{\texttt{clk\_evt}}(R), \texttt{clk\_evt})\} \\
= \; & \{(c_1, c_2, c_3, R') \mid R' \in \mathsf{gen}(\mathsf{select}_{\texttt{not clk\_evt}}(Q), \texttt{not clk\_evt})\} \\
& \cup \{(c_1, c_2, 11, R') \mid R' \in \mathsf{gen}(\mathsf{select}_{\texttt{false}}(Q), \texttt{false})\} \\
& \cup \{(c_1, 4, c_3, R') \mid R' \in \mathsf{gen}(\mathsf{select}_{\texttt{false}}(Q), \texttt{false})\} \\
& \cup \{(c_1, 4, 11, R') \mid R' \in \mathsf{gen}(\mathsf{select}_{\texttt{clk\_event}}(Q), \texttt{clk\_event})\} \\
= \; & \{(c_1, c_2, c_3, Q), (c_1, 4, 11, R)\} \, .
\end{aligned}
$$

Without any doubt the second version is more precise. The difference arises because select does not filter out the case when clk_event is successively true then false:

$$\mathsf{select}_{\texttt{not clk\_event}}(\mathsf{select}_{\texttt{clk\_event}}(R)) = R \, .$$

[1] We simplify the conditions on the fly, as is done in the implementation.

This leads to the unwanted abstract state $(c_1, 4, c_3, R)$ to appear in the result. The second version of explore easily removes this case since:

$$\mathsf{select}_{\texttt{clk\_event and not clk\_event}}(R) = \mathsf{select}_{\texttt{false}}(R) = \bot_{\mathcal{N}} .$$

Eventually, the computations are led to their end:

c_1	c_2	c_3	clk/clk$^+$	x/x$^+$	y/y$^+$	o/o$^+$
1	4, {clk}, ∞	11, {clk}, ∞	'1'/'1'	$\top/\top$	'1'/'1'	'0'/'0'
2	4, {clk}, ∞	11, {clk}, ∞	'1'/'1'	$\top/\top$	'1'/'1'	'0'/'0'
3	4, {clk}, ∞	11, {clk}, ∞	'1'/'0'	$\top/\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11, {clk}, ∞	$\top/\top$	$\top/\top$	$\top$/'1'	'0'/'0'
1, $\emptyset$, 1	4	11	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	5	11	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	6	11	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	7	11	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	12	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	13	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	14	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11, {clk}, ∞	$\top/\top$	$\top/\top$	$\top$/'1'	'0'/'0'
1	4, {clk}, ∞	11, {clk}, ∞	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'
2	4, {clk}, ∞	11, {clk}, ∞	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'
3	4, {clk}, ∞	11, {clk}, ∞	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'
1, $\emptyset$, 1	4, {clk}, ∞	11, {clk}, ∞	$\top/\top$	$\top/\top$	$\top$/'1'	'0'/'0'

The abstract environment associated with control point number 13 that was computed by the analysis is:

c_1	c_2	c_3	clk/clk$^+$	x/x$^+$	y/y$^+$	o/o$^+$
1, $\emptyset$, 1	4, {clk}, ∞	13	$\top/\top$	$\top/\top$	'1'/'1'	'0'/'0'

From this information, we can safely deduce that, when execution reaches the `display` command, the value '0' is necessarily printed out. Thus, we have automatically proved the property of interest. It is noteworthy that if both signals `clk` and `x` were to start with the value `'1'` this property would not hold.

This example is fairly simple. Its purpose was merely to present the static analysis in action. Also, it allowed to introduce the methodology associated with the tool:

- identify the properties of interest,
- choose an appropriate abstract domain and instantiate the static analysis,

$$\text{enter}\frac{{}^{l}\texttt{while } b \texttt{ do } {}^{l'}C;P \qquad \rho \vdash b \Longrightarrow \texttt{true} \qquad \kappa' = \kappa[l \leftarrow \min(\kappa(l)+1, \text{max\_unfold}(l))]}{(l,\kappa,\rho) \rightharpoonup_c (l',\kappa',\rho)}$$

$$\text{exit}\frac{{}^{l}\texttt{while } b \texttt{ do } P \qquad \rho \vdash b \Longrightarrow \texttt{false} \qquad \kappa' = \kappa[l \leftarrow 0]}{(l,\kappa,\rho) \rightharpoonup_c (next(l),\kappa',\rho)}$$

Figure 4.3: Instrumented semantics of loops

- run the resulting tool and check the correctness of the design.

It must be pointed out that the first two steps should be done by an expert once and for all for a given domain of application. Only the last step is performed by the end user.

4.6 Loop unfolding

To increase the precision of the tool, we wish to distinguish between distinct executions of loop bodies. To do so, we simply unfold loops at analysis time.

In order to formalize the construction, we augment the state (c,ρ) of sequential process with counters $\kappa \in \mathcal{K}$. The counter $\kappa(l)$ is associated to the loop whose entry node is labeled with l. It tracks the number of times execution goes through the loop up to a maximum max_unfold(l). The instrumented semantics of loops is depicted in Fig. 4.3. For all the other constructs, the semantics is almost the same: the counters are simply dragged around unchanged.

The rule for the parallel execution of processes in replaced by:

$$\Psi\text{-}i\frac{\forall j < i : c_j = (l_j, W_j, b_j, t_j) \qquad (c_i,\kappa,\rho) \rightharpoonup (c_i',\kappa',\rho')}{(c_1,\ldots,c_i,\ldots,c_n,\kappa,\rho) \rightarrow (c_1,\ldots,c_i',\ldots,c_n,\kappa,\rho')}$$

The remaining rules just carry the counters along. Initially, all counters are set to 0.

Semantics equivalence

If we ignore counters, the standard and instrumented semantics are equivalent. Indeed the transition systems $\rightharpoonup$ and $\rightharpoonup_c$ are in a strong bisimulation

[Mil90]. The bisimulation relation $\sim$ is induced by a translation π from nonstandard to standard states:

$$\sigma \sim \upsilon \Longleftrightarrow \sigma = \pi(\upsilon) .$$

The translation function π simply erases counters:

$$\pi(l, \kappa, \rho) = (l, \rho) .$$

The equivalence between standard and nonstandard semantics is expressed as:

Theorem 4.2. *The relation $\sim$ is a strong bisimulation. In other words, whenever $\sigma = \pi(\upsilon)$:*

- *if $\sigma \rightharpoonup \sigma'$ then there is υ' such that $\upsilon \rightharpoonup_c \upsilon'$ and $\sigma' = \pi(\upsilon')$,*
- *if $\upsilon \rightharpoonup_c \upsilon'$ then there is σ' such that $\sigma \rightharpoonup \sigma'$ and $\sigma' = \pi(\upsilon')$.*

Proof. Trivial by inspection of the various rules of the transition relations. □

The previous bisimulation implies in particular that both simulation algorithms generate the same set of reachable states:

Corollary 4.1. *For any initial configurations σ_0 and υ_0, if $\sigma_0 = \pi(\upsilon_0)$, then:*

$$\{\sigma \mid \sigma_0 \rightarrow^* \sigma\} = \{\pi(\upsilon) \mid \upsilon_0 \rightarrow_c^* \upsilon\},$$

where

$$\pi(c_0, \ldots, c_n, \kappa, \rho) = (c_0, \ldots, c_n, \rho) .$$

Proof. Easy by induction on the number of steps necessary to reach each state. □

Abstract interpretation

We now use:

$$\mathcal{P}^n \times \mathcal{K} \to \mathcal{N} ,$$

as our abstract domain. Finiteness of the abstract domain is ensured because the number of loops in a description is finite and each of them can not be unfolded more than some limit. The concretization function γ is defined as:

$$\gamma(Y) = \{(c_1, \ldots, c_n, \kappa, \rho) \mid \rho \in \gamma_{\mathcal{N}}(c_1, \ldots, c_n, \kappa)\} .$$

The abstract counterpart of the semantics is mostly similar to what was done in section 4.4. It is exhaustively described in the summary section 4.7 of the current chapter. For now, we just unveil the abstract semantics for loops:

$$\begin{aligned}[\![^{l}\texttt{while } b \texttt{ do } {}^{l'}C\,;P]\!]^{\sharp}(\kappa, R) = \\ \{(l', \kappa', \mathsf{select}_b(R)) \mid \kappa' = \kappa[l \leftarrow \min(\kappa(l)+1, \mathrm{max\_unfold}(l))]\} \\ \cup \{(next(l), \kappa', \mathsf{select}_{\texttt{not } b}(R)) \mid \kappa' = \kappa[l \leftarrow 0]\}\ .\end{aligned}$$

4.7 Recapitulation

We spend the remaining pages of this chapter to recapitulate the whole abstract interpretation of VHDL. The analysis is generic in the underlying numerical domain. In other words, it is expressed in terms of a few operators that manipulate any numerical domain of choice. This means we can rapidly adapt the tool to reach the best compromise between precision and efficiency. To do so, we must pick a numerical domain $(\mathcal{N}, \sqsubseteq_{\mathcal{N}})$ to encode abstract environments. The meaning of elements of the numerical domain must be specified thanks to a monotonic concretization function $\gamma_{\mathcal{N}}$. Finally, we must describe and implement the various primitives: $\bot_{\mathcal{N}}$, $\sqcup_{\mathcal{N}}$, assign, select, get_val and singleton. We must check that they all verify their respective soundness condition:

$$\begin{gathered}\emptyset = \gamma_{\mathcal{N}}(\bot_{\mathcal{N}}) \\ \gamma_{\mathcal{N}}(X) \cup \gamma_{\mathcal{N}}(Y) \subseteq \gamma_{\mathcal{N}}(X \sqcup_{\mathcal{N}} Y) \\ \left\{ \rho[a \leftarrow v] \;\middle|\; \begin{array}{c} \rho \in \gamma_{\mathcal{N}}(R) \\ \rho \vdash_L \mathit{lval} \Longrightarrow a \wedge \rho \vdash \mathit{exp} \Longrightarrow a \end{array} \right\} \subseteq \gamma_{\mathcal{N}}(\mathsf{assign}_{\mathit{lval} \leftarrow \mathit{exp}}(R)) \\ \{\rho \mid \rho \in \gamma_{\mathcal{N}}(R) \wedge \rho \vdash \mathit{exp} \Longrightarrow \texttt{true}\} \subseteq \gamma_{\mathcal{N}}(\mathsf{select}_{\mathit{exp}}(R)) \\ \{v \mid \exists \rho \in \gamma_{\mathcal{N}}(R) \wedge \rho \vdash \mathit{exp} \Longrightarrow v\} \subseteq \gamma_{\mathcal{V}}(\mathsf{get\_val}_{\mathit{exp}}(R)) \\ \rho \in \gamma_{\mathcal{N}}(\mathsf{singleton}(\rho))\ .\end{gathered}$$

We would like to emphasize that the tool is in no way limited to synthesizable VHDL descriptions only. Its definition directly stems from a formalization of the simulation algorithm. Hence, it does not require VHDL description to be translated into a finite state machine (or any finite model whatsoever).

The challenge of verifying a design, or in general a family of designs with common traits, is reduced to the choice of an appropriate numerical

domain. We need not worry about the idiosyncrasies of the VHDL dialect anymore. The last chapters present a possible instantiation of the abstract interpretation. It is tailored for the specific application domain of linear error correcting codes. In the next chapter, we describe the main features of the implementation. Then, chapter 6 explains how we successfully used the prototype implementation to verify a Reed Solomon error correcting code.

$$[\![^l \texttt{lval <= exp}]\!]^\sharp(R,\kappa) = \{(next(l), \kappa, \mathsf{assign}_{\texttt{future(lval)} \leftarrow \texttt{exp}}(R))\}$$
$$[\![^l \texttt{lval := exp}]\!]^\sharp(R,\kappa) = \{(next(l), \kappa, \mathsf{assign}_{\texttt{lval} \leftarrow \texttt{exp}}(R))\}$$
$$[\![^l \texttt{wait on } W \texttt{ until } b \texttt{ for } t]\!]^\sharp(R,\kappa) = \{(c, \kappa, R) \mid c = (next(l), W, b, t)\}$$
$$[\![^l \texttt{while } b \texttt{ do } {}^{l'}C; P]\!]^\sharp(R,\kappa) =$$
$$\{(l', \kappa', \mathsf{select}_b(R)) \mid \kappa' = \kappa[l \leftarrow \min(\kappa(l)+1, \max\_\mathrm{unfold}(l))]\}$$
$$\cup \{(next(l), \kappa', \mathsf{select}_{\texttt{not } b}(R)) \mid \kappa' = \kappa[l \leftarrow 0]\}$$
$$[\![^l \Sigma_{i\in[1,n]} b_i \texttt{ -> } {}^{l_i} P_i]\!]^\sharp(R,\kappa) = \{(l_i, \kappa, \mathsf{select}_{b_i}(R)) \mid i \in [1,n]\}$$
$$[\![^l \texttt{display(x, ..., x)}]\!]^\sharp(R,\kappa) = \{(next(l), \kappa, R)\}$$
$$[\![^l \texttt{lvec\_to\_int}(y,\ n,\ x)]\!]^\sharp(R,\kappa) = \{(next(l), \kappa, \mathsf{lvec\_to\_int}_{y,n,x}(R))\}$$
$$[\![^l \texttt{int\_to\_lvec}(x,\ n,\ y)]\!]^\sharp(R,\kappa) = \{(next(l), \kappa, \mathsf{int\_to\_lvec}_{x,n,y}(R))\}$$

where:

$$\mathsf{lvec\_to\_int}_{y,x}(R) = \mathsf{assign}_{x \leftarrow v}(R)$$
$$v = \begin{cases} \sum_{i=0}^{i=n-1} 2^{n-1-i}\varepsilon_i & \text{if } \forall i \in [0,n-1] : \mathsf{get\_val}_{y\texttt{[}i\texttt{]}}(R) \in \{\texttt{'0'}, \texttt{'1'}\} \\ & \text{and } \varepsilon_i = \begin{cases} 1 & \text{if } \mathsf{get\_val}_{y\texttt{[}i\texttt{]}}(R) = \texttt{'1'} \\ 0 & \text{otherwise} \end{cases} \\ \texttt{irnd()} & \text{otherwise} \end{cases}$$

and:

$$\mathsf{int\_to\_lvec}_{x,y}(R) = \bigodot_{i=0}^{i=n-1} \mathsf{assign}_{y\texttt{[}i\texttt{]} \leftarrow \varepsilon_i}(R)$$
$$v = \mathsf{get\_val}_x(R)$$
$$\forall i \in [1,n] : \varepsilon_i = \begin{cases} \texttt{lrnd()} & \text{if } v \notin \mathbb{Z} \\ \texttt{'1'} & \text{if } (v/2^{n-i})\%2 = 1 \\ \texttt{'0'} & \text{if } (v/2^{n-i})\%2 = 0 \end{cases}$$

Figure 4.4: Abstract equations for sequential processes

$$\Psi^\sharp\text{-}i\ \frac{\forall j < i : c_j = (l_j, W_j, b_j, t_j) \qquad c_i = l_i \qquad (c_i', \kappa', R') \in [\![^{l_i} C]\!]^\sharp(R, \kappa)}{(c_1, \ldots, c_i, \ldots, c_n, \kappa, R) \rightsquigarrow (c_1, \ldots, c_i', \ldots, c_n, \kappa', R')}$$

$$\Delta^\sharp\ \frac{\begin{array}{c}\forall i : c_i = (l_i, W_i, b_i, t_i) \\ (c_1', \ldots, c_n', R') \in \mathsf{explore}(c_1, \ldots, c_n, R) \qquad \exists i : c_i' \neq c_i\end{array}}{(c_1, \ldots, c_n, \kappa, R) \rightsquigarrow (c_1', \ldots, c_n', \kappa, \mathsf{update}(R'))}$$

$$\Theta^\sharp\ \frac{\begin{array}{c}\forall i : c_i = (l_i, W_i, b_i, t_i) \qquad (c_1, \ldots, c_n, R') \in \mathsf{explore}(c_1, \ldots, c_n, R) \\ \exists i : t_i \neq \infty \\ t = \min\{t_i \neq \infty\} \qquad \forall i : c_i' = \begin{cases} l_i & \text{if } t_i = t \\ (l_i, W_i, b_i, t_i - t) & \text{if } t_i \neq \infty \\ c_i & \text{otherwise} \end{cases}\end{array}}{(c_1, \ldots, c_n, \kappa, R) \rightsquigarrow (c_1', \ldots, c_n', \kappa, \mathsf{update}(R'))}$$

Figure 4.5: Abstract simulation algorithm

$$X_0(c_1, \ldots, c_n, \kappa) = \begin{cases} \mathsf{singleton}(\rho_0) & \forall i c_i = l_i \wedge \forall l : \kappa(l) = 0 \\ \bot_{\mathcal{N}} & \text{otherwise} \end{cases}$$

$$\mathbb{F}^\sharp(X)(c', \kappa') =$$
$$X_0(c', \kappa') \sqcup_{\mathcal{N}} \bigsqcup\nolimits_{\mathcal{N}} \{R' \mid \exists (c, \kappa, R) : X(c, \kappa) = R \wedge (c, \kappa, R) \rightsquigarrow (c', \kappa', R')\}$$

$$\mathcal{S}^\sharp = \mathrm{lfp}_\bot \mathbb{F}^\sharp$$

Figure 4.6: Generic abstract interpretation of VHDL

Chapter 5

Implementation

We have implemented the analysis in a prototype. There are two tools. A preprocessor `vhdlc` inputs VHDL files and transforms them into a format suitable for the analysis. Then, the abstract simulator `vhdla` performs the analysis itself. Both tools are written in OCaml [LDG+02].

5.1 Preprocessor

The purpose of the preprocessor is to simplify as much as possible the input of the static analysis. The powerful but complex VHDL constructs are broken down into smaller parts. Ambiguous operations are clarified. The numerous VHDL commands are translated into the small kernel presented in section 3.5. The preprocessor performs multiple steps. Each step individually is fairly simple. However, their effects add up to the point that an important burden is relieved from the static analysis.

The various preprocessing stages are depicted in Fig. 5.1. First the tool parses the input files. For each of them, it produces a corresponding abstract syntax tree. Then, the design hierarchy is flattened. Beginning from the top level module, the tool recursively goes through the various abstract syntax trees. Component instantiation statements are replaced by the body of the module they are referring to. While visiting the nodes of the abstract syntax tree, the tool also normalizes the various VHDL statements. At last, local variables and signals are brought to the global scope. Obviously, identifiers are renamed so as to avoid potential clashes between names. For instance, consider the following process:

```
process(A, B)
begin
```

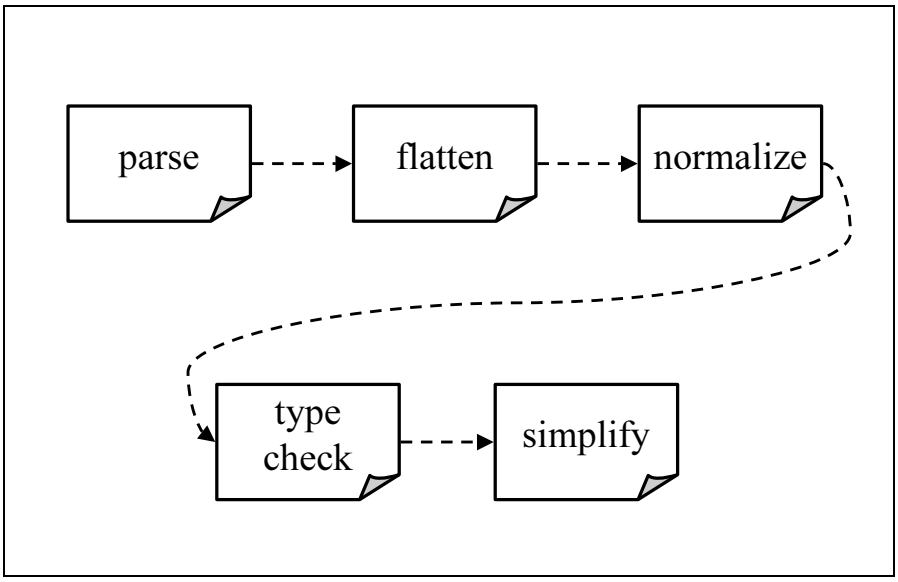


Figure 5.1: Phases of the preprocessor

```
  if (A = '1') then
    O <= B;
  else
    O <= '0';
  end if;
end process;
```

It is sensitive on the two std_logic signals `A` and `B`. It will be normalized to:

```
while true do
    (v0 = '1') -> v2 <= v1;
  + (not (v0 = '1')) -> v2 <= '0';
  wait on {v0, v1} until true for ever;
```

This first phase is somewhat similar to the elaboration procedure as described in the IEEE standard.

The resulting normalized abstract syntax tree is then type checked. The type checking algorithm is very standard, see [App98, DM82]. The only unusual detail is that the length of arrays must be part of the type information. In fact, the following is a valid VHDL expression which eases the

creation of arrays of constants:

```
('0', '0', '1', others => '0')
```

It denotes an array whose first elements are `'0'`, `'0'`, `'1'`. The remaining cells are filled with the value `'0'`. The length of this array entirely depends on the context where the expression is put. For instance, in the following statement, it must be the same as the length of array `a`:

```
a <= ('0', '0', '1', others => '0');
```

Similarly, most VHDL operators are overloaded. As an example, the `and` operator is not only defined for booleans but also for one-dimensional arrays. When its operands are arrays, they must have the same length and the operation is applied between matching elements.

Thanks to the type information, we can simplify the description a step further. Array assignments are scalarized. Namely, they are transformed into an equivalent sequence of simple scalar assignments. So, if `a`, `b` and `c` are arrays with respective range `0 to 2`, `2 to 4` and `2 downto 0`, then the piece of code:

```
a <= b xor c;
```

becomes:

```
a[0] <= b[2] xor c[2];
a[1] <= b[3] xor c[1];
a[2] <= b[4] xor c[0];
```

Care must be taken to perform this simplification in a safe way. In fact, it is sometimes necessary to introduce intermediate variables. It is obviously incorrect to translate:

```
a := (a(2), a(1), a(0));
```

into:

```
a[0] := a[2];
a[1] := a[1];
a[2] := a[0];
```

Rather, it must become:

```
tmp[0] := a[2];
tmp[1] := a[1];
tmp[2] := a[0];
a[0] := tmp[0]; a[1] := tmp[1]; a[2] := tmp[2];
```

In depth description of scalarization (for Verilog) can be found in [SV00].

Of course, the link between the entities of the resulting code and the ones from the original description is kept. This is necessary to provide readable feedback to the user after the analysis is run. All in all, the preprocessor accounts for approximately 2000 lines of OCaml code.

5.2 Abstract simulator

The abstract simulator implements the static analysis described in chapter 4. It is relatively compact: only 3000 lines of OCaml code.

Initialization

The abstract simulator reads the output provided by the preprocessor. During a preliminary initialization phase, it labels each sequential statement in the description with a unique integer. It also builds the abstract sequential equations that were presented in chapter 4, Fig. 4.4. These equations will be interpreted during the fixpoint computation phase. They are stored in an array for easy retrieval. At last, the abstract simulator creates the initial abstract configuration. It is a tuple of the first label of each sequential process, followed by a set of loop counters all equal to 0 and the initial abstract environment.

Fixpoint engine

We use a hashtable to implement the abstract domain. The keys of the hashtable are tuples of control points and loop counters, while the data are abstract environments. Naturally, the abstract environments equal to $\bot_{\mathcal{N}}$ are not stored in the hashtable.

To compute the fixpoint, we can choose from a variety of algorithms, see [Yi01] for an exhaustive survey. We opted for the simple yet efficient worklist algorithm of [HDT87]. Figure 5.2 contains a pseudo-code description of the fixpoint engine. Initially, the hashtable is created with a unique entry: the initial abstract environment `init_r` is associated to the initial tuple of control points and loop counters `init_c`. The worklist, a priority queue, is

```
reachable(init_c, init_r) {
  tbl := create_hashtbl();
  add(tbl, init_c, init_r);
  worklist := create_heap();
  insert(worklist, init_c);

  while not empty(worklist) {
    c := extract(worklist);
    r := get(tbl, c);
    succ := run(c, r);

    for (c', r') in succ {
      previous_r' := get(tbl, c');
      if not contains(previous_r', r') {
        r' := merge(previous_r', r');
        add(tbl, c', r');
        insert(worklist, c');
      }
    }
  }
}
```

Figure 5.2: Fixpoint engine

filled with a single element. While the heap is not empty, we extract the head element. We retrieve the corresponding abstract environment from the hashtable. The set of successors reached by one step of the abstract simulation algorithm from chapter 4 table 4.5 is computed by the function `run`. If necessary, the hashtable is updated and the heap is filled with new elements. Once the heap becomes empty, we have reached the fixpoint. For the worklist, we picked the data-structure of a heap so that the body of loops and alternative constructs is visited before the next instruction. In this piece of code, functions `contains` and `merge` are operations of the numerical domain. They respectively implement $\sqsubseteq_{\mathcal{N}}$ and $\sqcup_{\mathcal{N}}$. In the next section, we explain the choice of the numerical domains we put as a back-end to the tool.

5.3 The back-end: a numerical domain

For the back-end of the analyzer, there is a tremendous variety of numerical domains that we can choose from. To cite but a few see [Kil73, CC77, Min01, SKH02, MC04, Kar76, CH78, Gra91, GB04, RCK04]. We implemented a numerical domain which we believe is well adapted to the verification of linear error correcting codes (ECCs). The set $\mathbb{B}^n$ of all binary vectors of length n is a vector space over $\mathbb{B}$ of dimension n. The addition and multiplication by an element $\lambda \in \mathbb{B}$ are defined as:

$$\begin{aligned}(u_1, \ldots, u_n) \oplus (v_1, \ldots, v_n) &= (u_1 \oplus v_1, \ldots, u_n \oplus v_n)\\ \lambda \wedge (v_1, \ldots, v_n) &= (\lambda \wedge v_1, \ldots, \lambda \wedge v_n)\,.\end{aligned}$$

A map f from $\mathbb{B}^n$ to $\mathbb{B}^m$ is linear if for all $\lambda \in \mathbb{B}$ and $u, v \in \mathbb{B}^n$, we have:

$$\begin{aligned}f(u \oplus v) &= f(u) \oplus f(v)\\ f(\lambda \wedge v) &= \lambda \wedge f(v)\,.\end{aligned}$$

Alternately, a linear map $f : \mathbb{B}^n \to \mathbb{B}^m$ can be characterized by the existence of an $m \times n$ matrix $A = (a_{ij})$ such that for all $v \in \mathbb{B}^n$:

$$\forall i \in [1, m] : f(v)_i = \sum_{j=1}^{n} a_{ij} \wedge v_j\,.$$

Linear ECCs are defined by linear maps.

5.3.1 Boolean affine relationships

The domain $(\mathcal{K}_V, \gamma_\mathcal{K})$ of Karr [Kar76] tracks the affine equalities that hold between integer valued variables in V. An element of Karr domain is a system of linear constraints. The system is kept in row-echelon normal form thanks to the Gauss pivot algorithm. Karr describes all the primitives we need to manipulate the domain: $\bot_\mathcal{K}$, $\sqcup_\mathcal{K}$, $\mathsf{assign}_\mathcal{K}$, $\mathsf{select}_\mathcal{K}$, $\mathsf{singleton}_\mathcal{K}$ and $\mathsf{get\_val}_\mathcal{K}$. Each of these operations obey their respective soundness condition. We also need two additional primitives: extend and $\mathsf{project}$. They respectively add and remove variables from the domain. If V and W are sets of variables such that $V \subseteq W$ then:

$$\begin{array}{rcl} \{\rho \mid \rho|_V \in \gamma_\mathcal{K}(R)\} & \subseteq & \gamma_\mathcal{K}(\mathsf{extend}_W(R)) \\ \{\rho|_V \mid \rho \in \gamma_\mathcal{K}(R)\} & \subseteq & \gamma_\mathcal{K}(\mathsf{project}_V(R))\,. \end{array}$$

It is completely straightforward to adapt the algorithms of Karr so that they manipulate boolean affine equalities. Unfortunately, we can not use this domain as is: VHDL implementations embed $\mathbb{B}$ into the type std_logic. But variables of type std_logic do not only take the values `'0'` or `'1'` but also `'U'`. So we must track the set b of std_logic variables which are definitely different from `'U'`. Our numerical domain for linear relationships between std_logic is then:

$$\begin{array}{rcl} \mathcal{D} & = & \{(b,k) \mid b \subseteq V \wedge k \in \mathcal{K}_b\} \cup \{\bot\} \\ \gamma(b,k) & = & \{\rho \mid \rho|_b \in \gamma_\mathcal{K}(k)\} \\ \gamma(\bot) & = & \emptyset\,. \end{array}$$

Abstract inclusion and union are defined as:

$$\begin{array}{rcl} (b_1,k_1) \sqsubseteq (b_2,k_2) & = & b_2 \subseteq b_1 \wedge \mathsf{project}_{b_2}(k_1) \sqsubseteq_\mathcal{K} k_2 \\ (b_1,k_1) \sqcup (b_2,k_2) & = & (b, \mathsf{project}_b(k_1) \sqcup_\mathcal{K} \mathsf{project}_b(k_2)) \\ & & \text{where } b = b_1 \cap b_2\,. \end{array}$$

The assignment of an expression e to variable x is carried on in the Karr domain only when we are sure that e evaluates to `'0'` or `'1'`:

$$\mathsf{assign}_{x \leftarrow e}(b,k) = \begin{cases} (b_1, \mathsf{assign}_{\mathcal{K} x \leftarrow e}(\mathsf{extend}_{b_1}(k))) & \text{if } \mathsf{in01}_b(e) \\ (b_2, \mathsf{project}_{b_2}(k)) & \text{otherwise} \end{cases}$$
$$\text{where } b_1 = b \cup \{x\} \text{ and } b_2 = b \setminus \{x\}\,.$$

Predicate $\mathsf{in01}$ is given by:

$$\begin{cases} \mathsf{in01}_b(c) = \texttt{true} & \text{if } c \in \{\texttt{'0'}, \texttt{'1'}\} \\ \mathsf{in01}_b(\texttt{lrnd}) = \texttt{true} & \\ \mathsf{in01}_b(x) = (x \in b) & \\ \mathsf{in01}_b(\texttt{lnot } e) = \mathsf{in01}_b(e) & \\ \mathsf{in01}_b(e_1 \ \texttt{op} \ e_2) = \mathsf{in01}_b(e_1) \wedge \mathsf{in01}_b(e_2) & \text{if } \texttt{op} \in \{\texttt{land}, \texttt{lxor}\} \\ \mathsf{in01}_b(e) = \texttt{false} & \text{for all other expressions .} \end{cases}$$

Whenever the predicate $\mathsf{in01}_b(e)$ is equal to `true`, we can be sure that e evaluates to `'0'` or `'1'`:

Lemma 5.1. *For any environment ρ and set of variables b such that:*

$$\forall x \in b : \rho(x) \in \{\texttt{'0'}, \texttt{'1'}\},$$

if $\mathsf{in01}_b(e) = \texttt{true}$ *and* $\rho \vdash e \Longrightarrow v$ *then* $v \in \{\texttt{'0'}, \texttt{'1'}\}$.

Proof. By a straightforward structural induction. □

In a similar way to assignments, the selection operation refines the Karr's component of our domain only when possible:

$$\begin{cases} \mathsf{select}_{e_1 \texttt{ = } e_2}(b, k) = (b, \mathsf{select}_{\mathcal{K}e_1 \texttt{ = } e_2}(k)) & \text{if } \mathsf{in01}_b(e_1) \wedge \mathsf{in01}_b(e_2) \\ \mathsf{select}_{\mathsf{e}}(b, k) = (b, k) & \text{otherwise .} \end{cases}$$

We do not need a precise $\mathsf{get\_val}$ operator, so we may define it as:

$$\mathsf{get\_val}_e(b, k) = \Omega \,.$$

The $\mathsf{singleton}$ primitive is simply:

$$\mathsf{singleton}(\rho) = (b, \mathsf{singleton}_{\mathcal{K}}(\rho|_b)) \qquad \text{with } b = \{x \mid \rho(x) \in \{\texttt{'0'}, \texttt{'1'}\}\} \,.$$

We check that the primitives obey their respective soundness condition:

Proposition 5.1. *The following propositions are true:*

$$\begin{array}{rcl} R \sqsubseteq R' & \Longrightarrow & \gamma(R) \subseteq \gamma(R') \\ \emptyset & = & \gamma(\bot) \\ \gamma(R) \cup \gamma(R') & \subseteq & \gamma(R \sqcup R') \\ \{\rho[x \leftarrow v] \mid \rho \in \gamma(R) \wedge \rho \vdash e \Longrightarrow v\} & \subseteq & \gamma(\mathsf{assign}_{x \leftarrow e}(R)) \\ \{\rho \mid \rho \in \gamma(R) \wedge \rho \vdash e \Longrightarrow \texttt{true}\} & \subseteq & \gamma(\mathsf{select}_e(R)) \\ \{v \mid \exists \rho \in \gamma(R) \wedge \rho \vdash e \Longrightarrow v\} & \subseteq & \gamma_{\mathcal{V}}(\mathsf{get\_val}_e(R)) \\ \rho & \in & \gamma(\mathsf{singleton}(\rho)) \,. \end{array}$$

Proof. We examine each primitive in turn:

- **Inclusion**: Assuming that $(b_1, k_1) \sqsubseteq (b_2, k_2)$, we have the following sequence of relations:

$$\begin{aligned}
&\gamma(b_1, k_1) \\
&= \{\rho \mid \rho|_{b_1} \in \gamma_{\mathcal{K}}(k_1)\} && \lfloor \text{by definition} \rfloor \\
&\subseteq \{\rho \mid \rho|_{b_2} \in \gamma_{\mathcal{K}}(\mathsf{project}_{b_2}(k_1))\} && \lfloor \text{soundness of } \mathsf{project} \text{ and } b_2 \subseteq b_1 \rfloor \\
&\subseteq \{\rho \mid \rho|_{b_2} \in \gamma_{\mathcal{K}}(k_2)\} && \lfloor \text{soundness of } \sqsubseteq_{\mathcal{K}} \rfloor \\
&= \gamma(b_2, k_2) && \lfloor \text{by definition} \rfloor
\end{aligned}$$

- **Bottom element**: trivial by definition.

- **Union**: By definition of γ, we have:

$$\gamma(b_1, k_1) = \{\rho \mid \rho|_{b_1} \in \gamma_{\mathcal{K}}(k_1)\} .$$

  Obviously $b_1 \cap b_2 \subseteq b_1$ so soundness of $\mathsf{project}$ implies that:

$$\gamma(b_1, k_1) \subseteq \{\rho \mid \rho|_{b_1 \cap b_2} \in \gamma_{\mathcal{K}}(\mathsf{project}_{b_1 \cap b_2}(k_1))\} .$$

  A similar inclusion holds with (b_2, k_2). So thanks to the soundness of $\sqcup_{\mathcal{K}}$ we conclude by:

$$\begin{aligned}
& \gamma(b_1, k_1) \cup \gamma(b_2, k_2) \\
\subseteq\ & \{\rho \mid \rho|_{b_1 \cap b_2} \in \gamma_{\mathcal{K}}(\mathsf{project}_{b_1 \cap b_2}(k_1) \sqcup_{\mathcal{K}} \mathsf{project}_{b_1 \cap b_2}(k_2))\} \\
=\ & \gamma((b_1, k_1) \sqcup (b_2, k_2)) .
\end{aligned}$$

- **Assignment**: Assume $\rho \in \gamma(b, k)$ and $\rho \vdash e \Longrightarrow v$. This means that $\rho|_b \in \gamma_{\mathcal{K}}(k)$. So we can safely assert that:

$$\forall x \in b : \rho(x) \in \{\texttt{'0'}, \texttt{'1'}\} .$$

  There are now two possibilities depending on the value of $\mathsf{in01}_b(e)$:

  – If $\mathsf{in01}_b(e)$ is true, then by lemma 5.1, e must evaluate to some value in $\{\texttt{'0'}, \texttt{'1'}\}$. This ensures that $\mathsf{assign}_{\mathcal{K} x \leftarrow e}(\mathsf{extend}_{b \cup \{x\}}(k))$ is well defined. Moreover, the soundness conditions on $\mathsf{extend}_{\mathcal{K}}$ and $\mathsf{assign}_{\mathcal{K}}$ imply that:

$$\rho|_{b \cup \{x\}}[x \leftarrow v] \in \gamma_{\mathcal{K}}(\mathsf{assign}_{\mathcal{K} x \leftarrow e}(\mathsf{extend}_{b \cup \{x\}}(k))) .$$

    Hence, for this case, it comes that $\rho[x \leftarrow v] \in \gamma(\mathsf{assign}_{x \leftarrow e}(b, k))$.

– If $\mathsf{in01}_b(e)$ is false, the proof is easy. The soundness of $\mathsf{project}$ implies that $\rho|_{b\setminus\{x\}} \in \mathsf{project}_{b\setminus\{x\}}(k)$. Whatever the value v, $\rho[x \leftarrow v] \in \gamma(b \setminus \{x\}, \mathsf{project}_{b\setminus\{x\}}(k))$

- **Selection**: this case is similar to the proof for the assignment.
- **Value extraction**: trivial since $\gamma_{\mathcal{V}}(\Omega) = \mathbb{V}$.
- **Singleton**: Because of the soundness of $\mathsf{singleton}_{\mathcal{K}}$:

$$\rho|_b \in \gamma_{\mathcal{K}}(\mathsf{singleton}_{\mathcal{K}}(\rho|_b)) \,.$$

So, by definition of γ, this implies that:

$$\rho \in \gamma(b, \mathsf{singleton}_{\mathcal{K}}(\rho|_b)) \,.$$

□

In the implementation, we adopt a sparse matrix representation for the system of linear equalities. The set of variables in $\{\texttt{'0'}, \texttt{'1'}\}$ is encoded by a bitfield. We let n be the maximum number of variables. The memory usage of the abstract domain is of the order of n^2 while the complexity of the most expensive operation is in n^3.

We do not necessarily need to always collect all the linear equalities of a design. In particular, sometimes we only care about the functional relationship that hold between each variable y and some variables in a set X. If this is the case, we may safely trim the system of constraints to speed up the computation and free some memory. To do so, we first normalize the system in row-echelon form. We use an ordering of the column where the variables in X come last. Then, we discard any constraint that involves more than one variable not in X.

Example 5.1. We illustrate on the following system of equalities:

$$\begin{aligned} y_2 \oplus y_3 \oplus y_4 &= 1 \\ y_1 \oplus y_3 \oplus x_1 &= 0 \\ y_3 \oplus x_2 &= 1 \,. \end{aligned}$$

Suppose $X = \{x_1, x_2\}$. We put the system in normalized row-echelon form with the order $(y_1, y_2, y_3, x_1, x_2)$:

$$\begin{array}{ccccccccccc} y_1 & & & & & \oplus & x_1 & \oplus & x_2 & = & 1 \\ & y_2 & & \oplus & y_4 & & & \oplus & x_2 & = & 0 \\ & & y_3 & & & & & \oplus & x_2 & = & 1 \,. \end{array}$$

Any constraint with more than one variable not belonging to X is removed:

$$\begin{aligned} y_1 &= x_1 \oplus x_2 \oplus 1 \\ y_3 &= x_2 \oplus 1 \,. \end{aligned}$$

We implemented a tool, called `vhdla+`, where this optimization is performed after each join operation.

5.3.2 Constants

In addition to the domain of linear equalities, we also use the domain of constants [Kil73]. Among others, the domain of constants allows us to identify undefined std_logic variables (value `'U'`) and to evaluate the indices of arrays. Following the indication of [BCC+03], constants information are stored in a balanced binary tree. The major advantage is to improve sharing, which in turn speeds up many operations. To improve the precision of the operations, we implemented the various techniques that [Cou99] recounts: backward interpretation of expressions, on the fly simplifications of conditions and reductive iterations [Gra92].

To summarize, our abstract domain for environments is the product of two domains: constants and boolean linear equalities. The product is reduced [CC79b] to $(\bot, \bot)$ as soon as one component becomes the bottom element.

5.3.3 Arrays

Noticeably, references to array elements were absent from the previous expositions of the linear equality or constant domains. The reason is that we handle arrays in a very conventional way, as in [BCC+03]. All arrays are expanded, so that each array cell corresponds to a distinct location in the abstract environment. Expression are simplified: any array reference for which we can not exactly evaluate the indices is replaced by the expression `lrnd` or `irnd` according to its type. Assignments fall into two categories: strong or weak update. If the location denoted by the left hand side can be uniquely determined, then a strong update to that location occurs. Otherwise, a weak update takes place. In this case, all cells in the array are either the target of the assignment or retain their previous values. Weak update are performed by merging together the results of all the possible outcomes.

5.4 A trial run

We launch the prototype on our running example. First, we compile the description to the intermediate language:

```
$ vhdlc running.vhd > running.khl
```

Now, the content of the file `running.khl` is:

```
let
  signal v0 : logic with (v0 := ZERO);
  signal v1 : logic with (v1 := ZERO);
  signal v2 : logic with (v2 := ZERO);
  signal v3 : logic with (v3 := ZERO);
in
  v0 <= (lnot v0);
  wait on {} until true for 1;
  |
  if
    | (v0 =l= ONE) ->
    v1 <= (v2 land (lnot v3));
    | (bnot (v0 =l= ONE)) ->
  fi;
  wait on {v0} until true for ever;
  |
  v2 <= ZERO;
  v3 <= ONE;
  wait on {v0} until true for ever;
  while true do
    v2 <= lrnd;
    @(419, 426)display(v1);
    wait on {v0} until true for ever;
  od;
where
  v0 -> clk : std_logic;
  v1 -> o : std_logic;
  v2 -> x : std_logic;
  v3 -> y : std_logic;
```

Then, we launch the abstract simulator:

```
$ vhdla running.vhd running.khl
```

The abstract simulator outputs its results in `html` formal. All `display` directives are clickable. The constraints inferred about the arguments of the `display` command appear in the box below the code. Figure 5.3 shows a screenshot of the results after we clicked on the only `display` command. The analyzer tells us that the signal `o` is equal to `'0'`. This is a superset of the possible values of `x` at this point. So it must be understood as: if execution ever reaches instruction `display`, then the only possible value for `x` is `'0'`. Value `'1'` is definitively ruled out by the tool. Verification of this simple property is a success.

In the next chapter, we apply the tool on a more realistic example: a Reed-Solomon error correcting code.

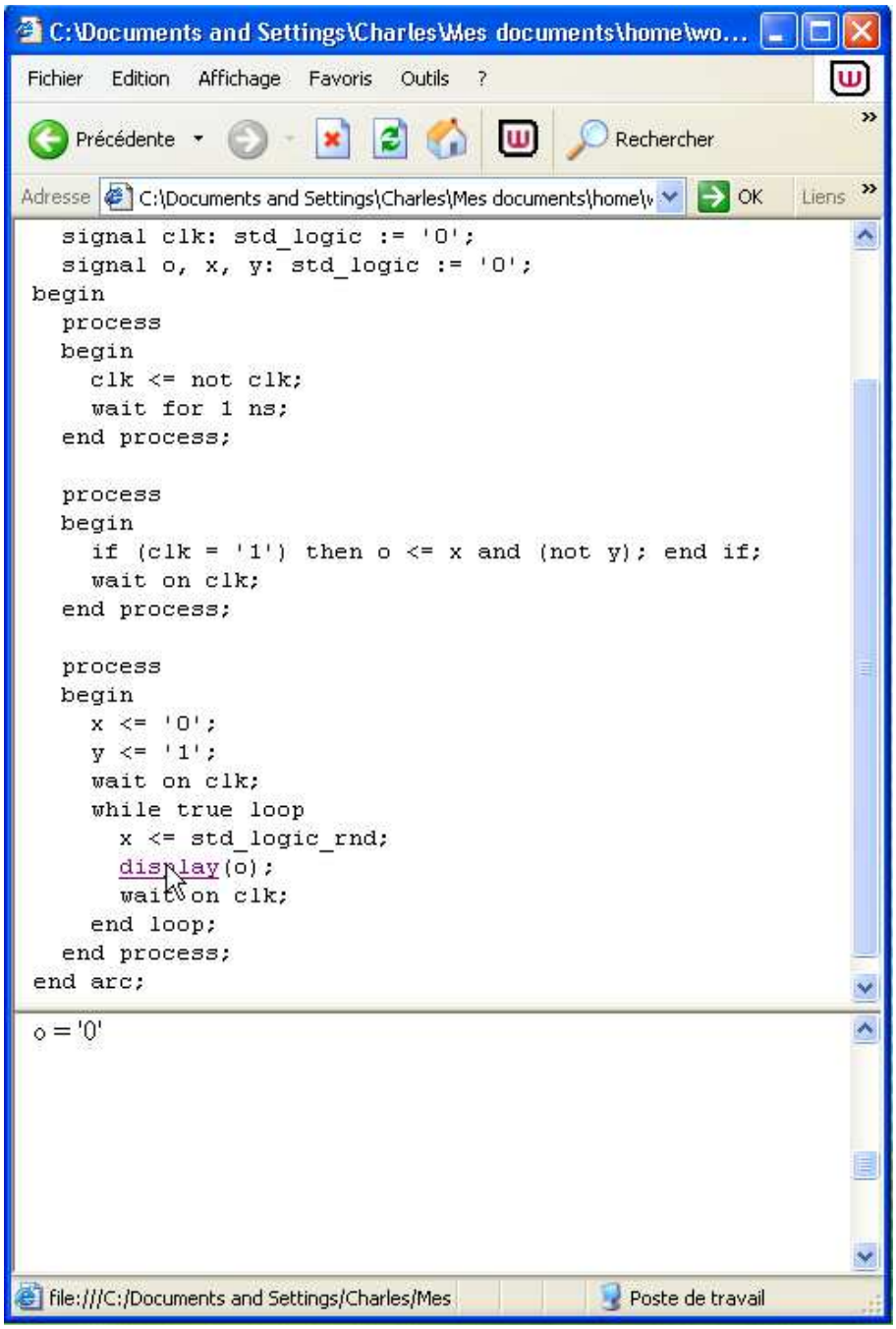


Figure 5.3: Results presented by the tool

Chapter 6

Reed Solomon error correcting code

6.1 Motivation

Our prototype is exercised on a hardware implementation of a Reed Solomon (RS) error correcting code (ECC). Before revealing the technical details, let us briefly expose the motivations for the automatic verification of components like the RS code. In order to design today's complex system-on-a-chips (SoCs), the ability to reuse existing Intellectual Properties (IPs) has become a necessity. Ideally, IP reuse shortens time-to-market and bounds design costs. However, in practice, the assembly of components from multiple sources on a single chips turns out not to be simple. First, the behavior of each IP must be clearly documented. Second, it must be checked that the IP matches its documentation. If it doesn't, then finding the origin of a flaw during whole chip simulation becomes a nightmare. In this chapter, we carry out both tasks of specification and verification for the RS component. Its expected behavior is described by non-deterministic testbenches written as behavioral VHDL processes. Then, the tool automatically establishes the correctness of the design. It is important to note that the VHDL description was provided to us by industrial hardware engineers. In no way was it written purposely for formal verification.

6.2 Reed Solomon

Data transmitted over a communication channel or stored on a memory device may undergo corruption. Error correcting codes elude possible infor-

```
for i = 0 to 65535 loop -- for any possible input
  -- conversion of an integer into an array of 16 std_logic
  din := conv_std_logic_vector(i, 16);
  enc := ref_enc(din);
  for s = 0 to 5 loop -- for any of the 6 symbols
     -- for any possible corruption of that symbol
    for x := 0 to 15 loop
      cor := enc;
      -- the sth symbol is corrupted
      cor(4*s+3 downto 4*s) := conv_std_logic_vector(x, 4);
      dout := ref_dec(cor);
      if (dout /= din) then  -- output different from input?
        report "Property failed.";
      end if;
    end loop;
  end loop;
end loop;
report "Property ok.";
```

Figure 6.1: Exhaustive simulation of the high-level specifications

mation loss by adding redundancy. In our information based society[1], ECCs have become ubiquitous: CDs, modems, digital television and wireless communication all incorporate them. We wish to validate a synthesizable Register Transfer Level (RTL) VHDL description of a Reed Solomon ECC [RS60] encoder and decoder. The informal documentation of the components explains how 16 bits messages are encoded by adding 8 bits of redundancy. The bits are packed in groups of 4 consecutive bits called symbols. The decoder is able to recover from a corruption that affects a unique symbol, i.e. at most 4 bits in the same block. The set of all recoverable corruptions of a vector of bits x is:

$$\text{corrupt}(x) = \{y \mid \exists s \in [0 \dots 5] : \forall i \notin [4 * s \dots 4 * s + 3] : y_i = x_i\} .$$

Then, the characteristic property of the RS code can be stated by:

$$\forall x \in \mathbb{B}^{16} : \forall y \in \mathbb{B}^{24} : y \in \text{corrupt}(\texttt{ref\_enc}(x)) \Longrightarrow \texttt{ref\_dec}(y) = x . \quad (6.1)$$

We write VHDL behavioral implementations of the encoding `ref_enc` and decoding functions `ref_dec`. These descriptions are going to be our golden

[1] or is it rather based on cheap energy?

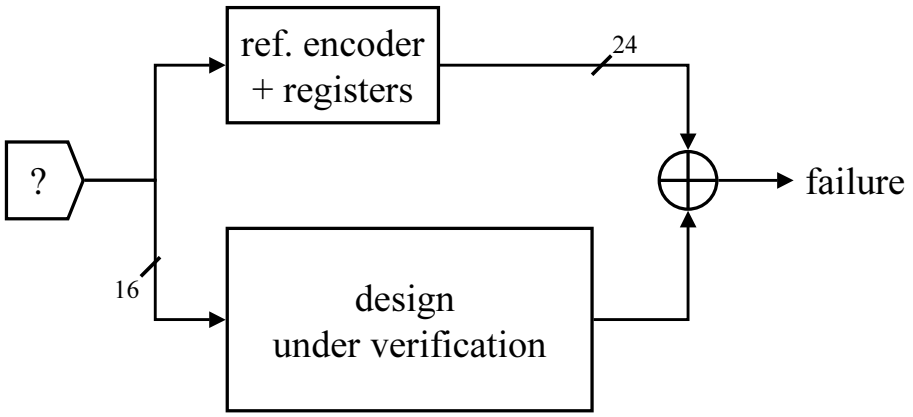


Figure 6.2: Design and verification harness

reference model. Hence, we need to make sure they observe property 6.1. This is checked by simulation of the code in Fig. 6.1. Note how booleans are encoded in the VHDL type std_logic. The function `conv_std_logic_vector` translates an integer into a std_logic_vector, i.e. an array of std_logic. It will be translated into the directive `lvec_to_int` by the preprocessor.

6.3 Encoder

6.3.1 Verification harness

It may seem that it is as simple to validate the actual components as it is to check the specification. This is far from true. Let us consider the encoder. Its role is not to encode one 16 bits message and then terminate. Rather, at every clock cycle it takes a different 16 bits message and encodes it. The design is pipelined, so that encoding is performed in 2 clock cycles. Our goal is not to show that the encoder encodes correctly one, or two, or a bounded number of messages. Rather, we want to ensure it will function correctly forever. At all times and whatever the input, the component must compute exactly the same sequence of values as the `ref_enc` function would. This property is expressed by the non-deterministic testbench of Fig. 6.3. The testbench is put together with the decoder and a clock generator. A graphical explanation of the configuration is given by Fig. 6.2. The signals `din` and `dout` are respectively the input and output signals of the encoder. The function `lvec_rnd` generates a vector of std_logics taking

```
-- input generator
process begin
  din <= lvec_rnd(16); wait until rising_edge(clk);
end process;
-- cycle-accurate, bit-accurate reference implementation
process begin
  tmp_dout <= ref_enc(din); ref_dout <= tmp_dout;
  wait until rising_edge(clk);
end process;
-- monitor
process begin
  wait until rising_edge(clk); wait until rising_edge(clk);
  wait until rising_edge(clk);
  -- the encoder has an asynchronous active '0' reset so:
  rst <= '1';
  wait until rising_edge(clk); wait until rising_edge(clk);
  wait until rising_edge(clk);

  while true loop
    if (dout /= ref_dout) then report "Failure."; end if;
    wait until rising_edge(clk);
  end loop;
end process;
```

Figure 6.3: Testbench for the RS encoder

arbitrary value among `'0'` and `'1'`. The monitor first resets the decoder. Then, it asserts that the output of the decoder and the result computed by the reference implementation are forever equal. Put in other terms, we simply compare a simple cycle-accurate, bit-accurate description with the detailed RTL synthesizable description of the RS encoder. At this point, showing the correctness of the encoder boils down to proving that no simulation run ever executes the `report ¨Failure.¨;` statement. We use our tool to check this automatically.

6.3.2 Results

Figure 6.4 displays the results computed by the tool. The tool statistics indicate that computation took 110 seconds and that the maximum memory

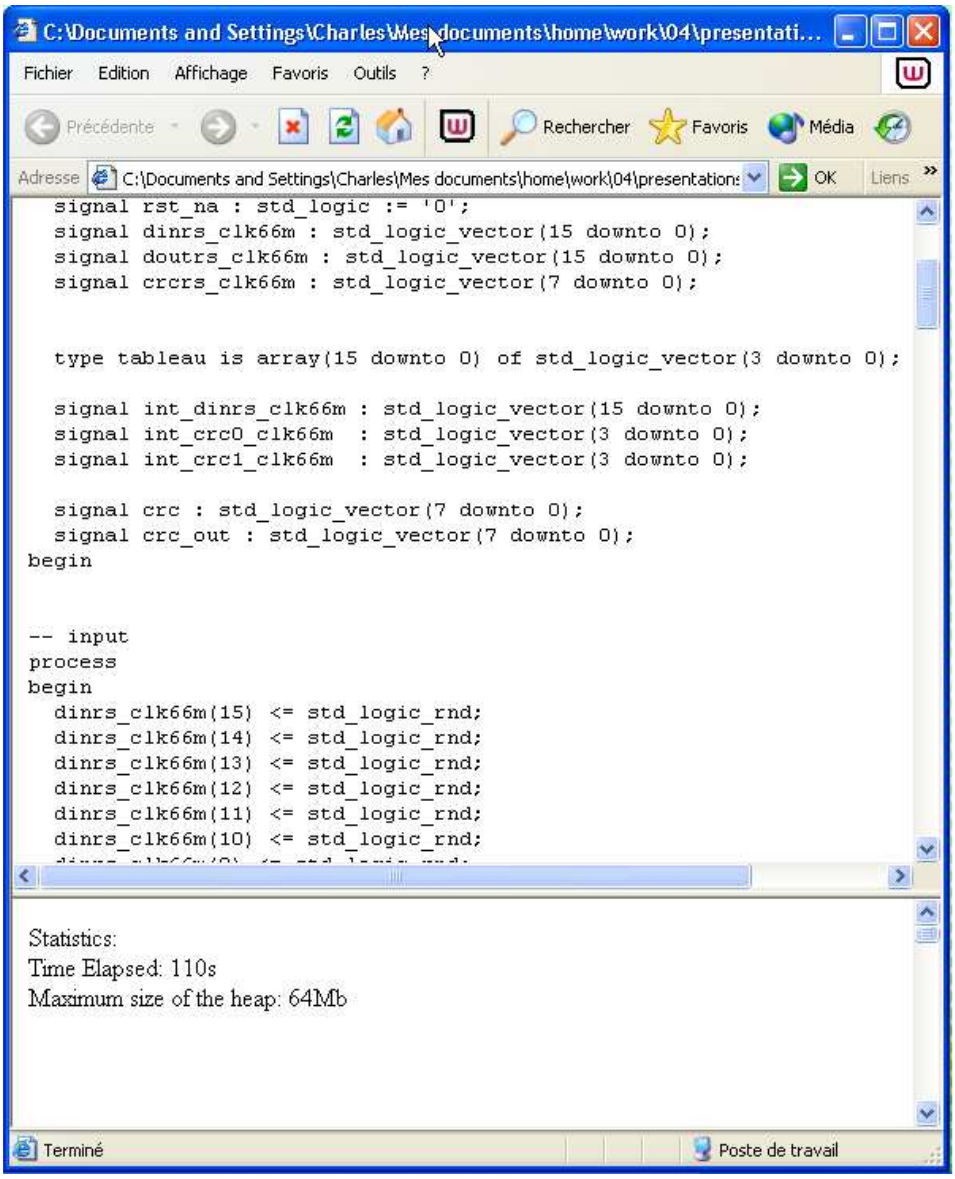


Figure 6.4: Results and statistics

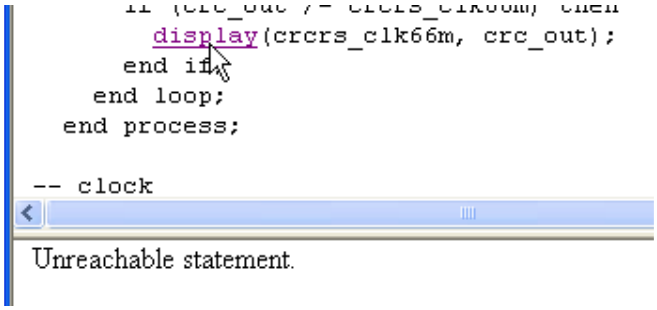
```
-- comparison
  process
  begin
  ❸ display(crc_out(0));
    wait until rising_edge(clk66m);
    wait until rising_edge(clk66m);
    wait until rising_edge(clk66m);
    rst_na <= '1';
    wait until rising_edge(clk66m);
    wait until rising_edge(clk66m);
    wait until rising_edge(clk66m);

    while true loop
      wait until rising_edge(clk66m);
      display(int_crc1_clk66m, int_dinrs_clk66m);
      display(int_crc1_clk66m(1), int_dinrs_clk66m);

   ❶ display(crcrs_clk66m, crc_out);
      if (crc_out /= crcrs_clk66m) then
     ❷ display(crcrs_clk66m, crc_out);
      end if;
    end loop;
  end process;
```

Figure 6.5: Monitor

consumption was 64Mb. We scroll down to the reference monitor of Fig. 6.5. Then we click on the display labeled with 1 in Fig. 6.5 to find out that the output of the RS encoder (here `crcrs_clk66m` and its specification (`crc_out`) are always equal:

```
crc_out(0) xor crcrs_clk66m(0) = '0'
crc_out(1) xor crcrs_clk66m(1) = '0'
crc_out(2) xor crcrs_clk66m(2) = '0'
crc_out(3) xor crcrs_clk66m(3) = '0'
crc_out(4) xor crcrs_clk66m(4) = '0'
crc_out(5) xor crcrs_clk66m(5) = '0'
crc_out(6) xor crcrs_clk66m(6) = '0'
crc_out(7) xor crcrs_clk66m(7) = '0'
```

This allows the tool to conclude that the failure statement at label 2 in Fig. 6.5 is:



The automatic verification of the encoder is a success. Note that if we click on the display labeled by 3 in Fig. 6.5, we can check that indeed `crc_out(0) = 'U'`.

6.3.3 Coding style

The whole description (RS encoder and the testbench) takes up 250 lines of VHDL code. The description is optimized for synthesis. It is therefore quite easy to understand the structure of the circuit that will be generated. But it can be somewhat difficult for a human reader to understand its functionality. The abstract simulator must handle this kind of code without any loss of precision. Here is a short but representative example:

```
process
type t is array(7 downto 0) of std_logic_vector(3 downto 0);
constant lut : t:=(x"9",x"6",x"E",x"1",x"F",x"F",x"C",x"5");
variable x   : std_logic_vector(7 downto 0);
variable tmp : t;
variable y   : std_logic_vector(3 downto 0);
variable i   : integer range 0 to 8;
begin
  x := lvec_rnd(8);
  i := 0;
  while i < 8 loop
    if (x(i)='1') then tmp(i) := lut(i);
    else tmp(i):=(others=>'0'); end if;
    i := i + 1;
  end loop;
  i := 0;
  while i < 8 loop
    tmp(i) := tmp(i) xor tmp(i+1); i := i + 2;
```

```
    end loop;
    i := 0;
    while i < 8 loop
      tmp(i) := tmp(i) xor tmp(i+2); i := i + 4;
    end loop;
    y := tmp(0) xor tmp(4);
    display(x, y);
  end process;
```

This description computes the vector `y` in terms of the 8 std_logics from `x`. In the first loop, each element from the vector `x` is combined with some mask by an `and` operation. The masks are defined in hexadecimal notation in the constant lookup table `lut`. Then, the last part of the code builds four trees of xors whose root are the four std_logics of the vector `y`. Once the loops are unfolded, the tool discovers, thanks to the integer constant domain, the exact index of each vector access. At the same time, it tracks the affine relationships that link the various std_logic variables. It propagates this information down to the display directive. It eventually obtains the following constraints:

```
y(0) xor x(0) xor x(2) xor x(3) xor x(4) xor x(7) = '0'
y(1) xor x(2) xor x(3) xor x(5) xor x(6) = '0'
y(2) xor x(0) xor x(1) xor x(2) xor x(3) xor x(5) xor x(6)='0'
y(3) xor x(1) xor x(2) xor x(3) xor x(5) xor x(7) = '0'
```

The abstract simulator is thus able to precisely infer the input-output relationship for this piece of code. If some constant in the lookup table is incorrect, or if the array accesses are wrong, then this description could be faulty. However, from these constraints, the tool would be able to catch possible mismatch with a reference implementation.

6.3.4 Combinational processes

The description is also optimized for simulation. Blocks of combinational logic, that is logic without memory elements, are implemented as stand-alone processes. All the signals that are read by a combinational process are included in its sensitivity list. As a consequence, a combinational process executes only when one of the signals it depends on is modified. This has the effect of speeding up simulation, as most of the time only a few combinational processes will have to be executed.

However, combinational processes make verification harder. For each of them, the abstract simulator must consider two cases: either it wakes up or

not. In case the process does not wake up, it is important that the abstract simulator uses the information that the input signals are unchanged. Otherwise, it cannot establish the input/output relation of the combinational process.

Consider the example below:

```
process(a, b) begin y <= a xor not b; end process;
process begin
  a <= '0'; b <= '0'; wait for 1 ns;
  while true loop
    a <= lrnd(); b <= lrnd(); wait for 1 ns;
    display(a, b, y);
  end loop;
end process;
```

The first process is combinational. Its sensitivity list contains signals `a` and `b`. It is compiled by our preprocessor `vhdlc` into the following code:

```
while true do
  y <= a lxor (lnot b);
  wait on {a, b} until true for ever;
```

The abstract simulator is able to infer the link between `a`, `b` and `y` at the display:

```
y xor a xor b = '1'
```

Combinational processes speed up conventional simulation, but unfortunately slow down abstract simulation. It is easy to write small pieces of code for which the abstract simulator exhibits terrible performances. Here is an example where 100 combinational processes copy a vector of 100 elements:

```
process (x(99)) begin y(99) <= x(99); end process;
process (x(98)) begin y(98) <= x(98); end process;
  ...
process (x(0)) begin y(0) <= x(0); end process;
```

Fortunately, such critical situations do not show up in the cases we have examined. Hence, combinational processes do not dramatically hurt the performances of our tool.

6.4 Decoder

6.4.1 Specification

The description of the decoder is much harder to validate than the encoder. The main reason is that the function computed by the decoder is not linear. The reference implementation `ref_dec` for the decoder works as follows:

```
syndrome := control(din);
dout := correct(din, syndrome);
```

The linear map `control` computes the syndrome of the message. The function `correct` then modifies the message according to the value of the syndrome. If we fix the syndrome, then `correct(din, syndrom)` becomes linear in `din`. The syndrome is made up of two symbols. Each symbol is 4 bits (here std_logics) wide. Consequently, we can split the domain of the reference decoding function `ref_dec` into 256 pieces. The restriction of `ref_dec` on any of these pieces is linear.

Moreover, the decoder is pipelined with a depth of five cycles. With the initial row of registers put on the inputs, this means that at most six different messages may be at the same time in different stages of the pipeline. If we naively distinguish the 256 cases for each of these stages then we are bound to suffer from the state explosion problem since:

$$256^6 = 2^{48} \approx 280000 \text{ billions}\,.$$

We formulate the correctness property so as to circumvent this problem. Let us consider a run of a conventional simulator on the decoder. We restrict our observation to the input `din` and output `dout` of the component at rising edges of the clock. We want to ensure that, 6 clock cycles after the input of a message, the expected result shows up:

$$\forall t : \forall x : \texttt{din}_t = x \Longrightarrow \texttt{dout}_{t+6} = \texttt{ref\_dec}(x)\,. \tag{6.2}$$

To check this property, we drive the component with the process in Fig. 6.6. First, the component is reset. Then, the component is fed with arbitrary values for some time. This first part of the driver expresses the $\forall t$ in the equation (6.2). Then, an input message `x` is randomly chosen. We wait 6 cycles for it to reach the output of the component. At last, the result from the component and the expected value `y` are compared. Again here, the statement `report "Failure"` must be shown unreachable. Since the function `ref_dec` is not linear, the verification can't be performed in one

```
process begin
  -- reset phase
  din <= (others => '0');
  wait until rising_edge(clk); wait until rising_edge(clk);
  wait until rising_edge(clk); wait until rising_edge(clk);
  rst_na <= '1';

  -- operating phase
  while rnd() loop
    din <= lvec_rnd(24); wait until rising_edge(clk);
  end loop;

  -- testing phase
  x := lvec_rnd(24); y := ref_dec(x);
  din <= x;
  wait until rising_edge(clk); din <= lvecrnd(24);
  wait until rising_edge(clk); din <= lvec_rnd(24);
  wait until rising_edge(clk); din <= lvec_rnd(24);
  wait until rising_edge(clk); din <= lvec_rnd(24);
  wait until rising_edge(clk); din <= lvec_rnd(24);
  wait until rising_edge(clk); din <= lvec_rnd(24);

  if (dout /= y) then report "Failure"; end if;
end process;
```

Figure 6.6: Driver for the RS decoder

run the current abstract simulation. Instead, we can equivalently check 256 simpler properties. We specialize the previous driver for each possible value of the syndrome. For instance, for a syndrome equal to `"10110111"`, the following code simply replaces the last part of the driver:

```
-- testing phase for a syndrome equal to "10110111"
x := lvec_rnd(24); syndrome := control(x);
if (syndrome = "10110111") then
  y := correct(x, syndrome);
  din <= x;
  wait until rising_edge(clk); din <= lvec_rnd(24);
  wait until rising_edge(clk); din <= lvec_rnd(24);
```

```
    wait until rising_edge(clk); din <= lvec_rnd(24);
    wait until rising_edge(clk); din <= lvec_rnd(24);
    wait until rising_edge(clk); din <= lvec_rnd(24);
    wait until rising_edge(clk); din <= lvec_rnd(24);

    if (dout /= y) then report "Failure"; end if;
  end if;
```

The driver and description amount to 400 lines of VHDL code. After preprocessing, the intermediate code is approximately 1000 lines long. It is fed to the analyzer.

6.4.2 Inlining combinational processes

To our surprise and dismay the tool fails. After inspection of the results, we find out it needs to establish intermediate invariants that involve non-linear constraints. Consider the following example:

```
process begin
  a <= lrnd(); b <= lrnd();
  wait for 1 ns;
  display(a, b, y);
  a <= '1'; b <= lrnd();
  wait for 1 ns;
  display(a, b, y);
end process;
process(x) begin y <= a and b; end process;
```

The abstract simulator is unable to infer the linear equality

$$\texttt{b} = \texttt{y}$$

that holds at the second `display` statement. This problem arises because of the combinational process. Let us follow the computation of the tool. At the first `display` statement, the value of signal `a`, `b` and `y` are linked by the relationship:

$$\texttt{a and b = y}\,.$$

However, simply because this constraint is not a boolean linear equality, it is not inferred. Then, when the tool reaches the second wait statement, it must explore the two possible outcomes of the combinational process. Either it wakes up and `y` is assigned the value of $\texttt{'1' and b} = \texttt{b}$. Or, it stays idle

in which case `y` is not modified. Unfortunately, the tool already lost all information about `y`. The restraint `y = b` can not be deduced.

The exact same sequence of events prevents the analysis of the decoder from being conclusive. To resolve this problem, we inline the code of each combinational process wherever the signals it is sensitive on may be modified. As for now, this is done by hand. In the future it can be easily made automatic and integrated as a final phase of the preprocessor `vhdlc`. On the previous example, the transformation produces:

```
process begin
  tmp_a := lrnd(); tmp_b := lrnd();
  a <= tmp_a; b <= tmp_b;
  y <= tmp_a and tmp_b;
  wait for 1 ns;
  display(a, b, y);
  tmp_a := '1'; tmp_b := lrnd();
  a <= tmp_a; b <= tmp_b;
  y <= tmp_a and tmp_b;
  wait for 1 ns;
  display(a, b, y);
end process;
```

At last, the abstract simulator succeeds. It computes during 849 seconds on an AMD Athlon MP 2200+ and consumes 141 megabytes at its peak. Of course, to completely validate the decoder, the abstract simulator must be run 256 times, i.e. one time for each possible syndrome.

6.4.3 Debugging

The tool may also be useful for the purpose of debugging. To trace the origin of errors in a faulty design, we can issue `display` commands at various places in the description. For instance, it is possible to discover the conditions on the input signals that lead the design into an abnormal state. Also, the driver described in Fig. 6.6 lets us easily follow the data flow through the pipeline. We can locate the first state in the pipeline where signals in the design do not match their expected values.

6.4.4 Statistics

We compared with the BDD-based model checker VIS [Gro96]. To our knowledge there is no freely available model checking tool that inputs VHDL

	Size in various metrics			
program	VHDL lines	IR lines	Verilog lines	VIS latches
encoder	251	338	235	337
decoder	395	963	519	647
dec. 1	215	370	206	286
dec. 1–2	265	484	310	486
dec. 1–3	305	602	358	529
dec. 1–4	358	828	415	617
	Verification time (s)			
program	vhdla	vhdla+	vis static	vis dynamic
encoder	58	45	1666	4379
decoder	849	593	>960	>172800 (48h)
dec. 1	37	37	>840	2774
dec. 1–2	152	166	>845	4540
dec. 1–3	259	261	>1826	78037 (21h)
dec. 1–4	513	420	>895	>172800 (48h)
	Peak memory consumption (Mb)			
program	vhdla	vhdla+	vis static	vis dynamic
encoder	50	37	693	173
decoder	141	88	>2000	>374
dec. 1	23	20	>2000	233
dec. 1–2	44	37	>2000	243
dec. 1–3	66	49	>2000	277
dec. 1–4	98	73	>2000	>333

Benchmarks dec. 1 to dec. 1–4 are troncated versions of the decoder: dec. 1 contains only the first stage of the pipeline, dec. 1–2, the first two stages and so on.

Table 6.1: Statistics on an AMD Athlon MP 2200+ with 2GB of memory

code. This is the reason why we chose VIS which reads synthesizable synchronous Verilog descriptions. The VHDL descriptions were translated by hand. The asynchronous reset had to be removed completely. We also transformed the driver of Fig. 6.6 into a form suitable for synthesis. The control flow must be made explicit by encoding the program point into a register. For instance, the behavioral code in Fig. 6.7 is rewritten into the synthesizable code of Fig. 6.8.

```
initial begin
  @(posedge clk);
  x = 1;
  @(posedge clk);
  x = 0;
end
```

Figure 6.7: Behavioral Verilog

```
always @(posedge clk) begin
  case (cp)
    0: begin x = 1; cp = 1; end
    1: begin x = 0; cp = 2; end
    2: begin end
  endcase end
```

Figure 6.8: Synthesizable version

Table 6.1 displays various statistics of our benchmarks. The sizes of the descriptions are expressed in number lines of VHDL code, of the intermediate representation and of Verilog. The number of latches of the circuit synthesized by VIS is also shown. Then, time and memory consumption are summed up. Line vhdla+ is the implementation of the stronger abstraction. We ran VIS with the check_invariant command. We tried both static and dynamic variables ordering for the BDDs. For the dynamic ordering, we used the window method. Dynamic ordering improves the memory consumption, but at the cost of increased computation time. Both methods failed: static ordering burns the 2GB of available memory very quickly, whereas dynamic ordering does not finish within a timeout of 48 hours. The last benchmarks are performed on restricted versions of the decoder where only the first few stages of the pipeline are considered. These benchmarks show specialized tools outperform general algorithms like BDD based model checking.

It is difficult to compare our technique with conventional simulation. Simulation lacks the full coverage that abstract simulation ensures. However, at the time the component was designed, the hardware engineers allocated two days to simulation. The whole verification effort to fully validate the encoder and decoder would take less than two days with our prototype implementation. So, we believe abstract simulation is competitive: for a similar amount of time, it produces a much higher valued result.

6.5 Related work

There exists numerous formal verification tools for hardware. Before we enumerate a few of them, let us point out that most, if not all, input synthesizable hardware descriptions. This crucial hypothesis greatly simplifies the task of verification. Indeed, the description can be compiled into a finite representation, be it a circuit, a transition system or an automaton, with few constructs and a clear and compact semantics. From there, it is not necessary to deal with std_logic variables, with the value undefined `'U'` or with integers. Boolean is the only data-type. There is no loops and no problem arises because of the peculiar semantics of combinational processes. In return, and in contrast to our approach, these tools are strictly limited to synthesizable descriptions. Thus, making them unfit for an early use in the design cycle. Moreover, they rely on the correctness of the compilation from the hardware language to their internal representation, a point which is often overlooked.

We have compared our approach with traditional BDD [Bry86] based model checking [CES83, VW86, BCL$^+$94, LP85]. Using the model checker VIS [Gro96], we observed an undeniable blowup in the size of the BDDs or the time devoted to simplify them on the fly.

In bounded model checking [BCC$^+$99, SSS00, CBRZ01], a violation of the property reachable in less than a bounded number of steps is searched for. The task is reduced to a propositional satisfiability problem and solved with regular sat-solvers [DP60, MMZ$^+$01]. The tool described in [CKY03] checks the consistency of a Verilog design with its specification written in ANSI-C. In theory, since synthesizable descriptions give rise to finite transition systems, the bound can be large enough so as to make bounded model checking complete. Essentially, in contrast to symbolic model checking, SAT based methods trade memory consumption for computation time. It proves very efficient to quickly find errors in designs. However, when the property holds, bounded model checking too tends to suffer from the state explosion problem.

Symbolic simulation algorithms [Bry90, WDB00] extend the power of traditional simulators by manipulating symbolic expressions instead of plain values. In particular, ternary symbolic simulation operates on BDDs whose nodes are variables denoting the input to the circuit and whose leaves are 0, 1 or X (for unknown). Symbolic trajectory evaluation [SB95] and its generalized form [Jai97] improve on ternary symbolic simulation by providing a logic to express the property to check. Symbolic trajectory evaluation was shown to be an abstract interpretation in [Cho99]. The efficiency of symbolic

simulation stems from the limited number of variables needed for the BDDs. Indeed, this number depends only on the property to check and not on the design. For instance, to establish the correctness of the RS decoder, only 24 variables would be needed. In comparison to the domain of linear constraints, which is polynomial, there is still the possibility of an exponential blowup. We couldn't find any implementation of symbolic simulation. So, unfortunately, we didn't have the opportunity to experiment thoroughly and compare with our technique. The implementation of a symbolic simulation for RT-level Verilog is described in [KKD01]. The authors claim to support the full IEEE 1364-1995 semantics. However, they do not state, even less prove, the soundness of their algorithm with respect to a formalization of the Verilog semantics.

As for the specification part, we could have used more complex formalisms like LTL [Pnu77], PSL/sugar [BBDE$^+$01], itself an extension of CTL [CE81] or even the logic of constraints [BBL$^+$01]. These logics are useful to specify complex control properties or liveness properties. But as we have seen, for the simple case study of RS, VHDL augmented with nondeterminism is already expressive enough.

Chapter 7

Conclusion

We described a static analysis for the hardware description language VHDL. The analysis computes an approximation of the states that would be reached during any run of a conventional simulator. Following the methodology of abstract interpretation, it is derived from a formalization of the simulation semantics of VHDL. A proof of soundness was produced, thus filling the gap between the analysis and the hardware description language. The analysis is parametric in the underlying representation of sets of states. Various numerical abstract domains may be plugged in and allow to achieve the right balance between cost and precision. We instantiated and implemented the analysis with a domain particularly suitable for the verification of linear error correcting code. The validation of a synthesizable implementation of a Reed Solomon error correcting code was achieved. The tool revealed excellent performances in practice.

This study demonstrates it is feasible to apply formal methods to the same description that hardware engineers already use for both synthesis and simulation. This is a step forward toward the integration of formal tools into existing design flows.

Also, this work confirms that, with the right abstraction, both precision and efficiency can be combined in one tool. Obviously there is a setback: we had to give up on generality. Fortunately, the cost we pay is relatively low, since only the back-end, i.e. the numerical domain, needs to be adapted for each application domain. Maybe, for hard verification problems, domain-specific verification tools have become a necessity.

The full potentiality of our analysis remains largely untapped. The case study was done on a synthesizable code, once the design is nearing completion. Yet, our tool can handle behavioral code too. It would, therefore, be

interesting to use it at an earlier stage in the design cycle. Also, the exploration of application domains, other than just linear error correcting codes, will certainly give rise to fascinating novel abstractions. Much future work is left. For now, we simply hope this study contributes to make abstract interpretation evolve from a craft to an engineering practice.

Bibliography

[App98] A.W. Appel. *Modern compiler implementation in ML.* Cambridge University Press, 1998.

[BBDE+01] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic sugar. In *13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367. Springer, 2001.

[BBL+01] F. Balarin, J.R. Burch, L. Lavagno, Y. Watanabe, R. Passerone, and A.L. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT'01)*, 2001.

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedure instead of BDDs. In *Proceedings of the 36th Design Automation Conference (DAC'99)*, pages 317–320. ACM Press, 1999.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM Press, 2003.

[BCL+94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[BFK94] P.T. Breuer, L.S. Fernández, and C.D. Kloos. Clean formal semantics for VHDL. In *The European Conference on Design*

Automation (EDAC), European Test Conference (ETC) and The European Event in ASIC Design (EUROASIC), pages 641–647. IEEE Computer Society Press, 1994.

[BFK95] P.T. Breuer, L.S. Fernández, and C.D. Kloos. A simple denotational semantics, proof theory and a validation condition generator for unit-delay VHDL. *Formal Methods in System Design*, 7(1/2):27–51, 1995.

[BG00] D. Borrione and P. Georgelin. Formal verification of VHDL using VHDL-like ACL2 models. In J. Mermet, editor, *Electronic Chips and Systems Design Languages*. Kluwer Academic Publishers, 2000.

[BGM95] E. Börger, U. Glässer, and W. Müller. A formal definition of an abstract VHDL'93 simulator by EA-Machines. In C.D. Kloos and P.T. Breuer, editors, *Formal Semantics for VHDL*, pages 107–139. Kluwer Academic Publishers, 1995.

[BJ03] R.A. Bergamaschi and Y. Jiang. State-based power analysis for systems-on-chip. In *Proceedings of the 40th Design Automation Conference (DAC'03)*, pages 638–641. ACM, 2003.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[Bry90] R.E. Bryant. Symbolic simulation - techniques and applications. In *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC'90)*, pages 517–521. IEEE Computer Society Press, 1990.

[BSK94] P.T. Breuer, L. Sánchez, and C.D. Kloos. Proving hardware designs. In *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, page 745. MIT Press, 1994.

[BT00] O. Botchkarev and S. Tripakis. Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations. In *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2000.

[BWH+03] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A.L. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003.

[CBRZ01] E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.

[CC79a] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[CC79b] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CE81] E.M. Clarke and E.A. Emerson. The design and synthesis of synchronization skeletons using temporal logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: a practical approach. In *10th ACM Symposium Principles of Programming Languages (POPL'83)*, 1983.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference*

Record of the 5th Annual ACM Symposium on Principles of Programming Languages, 1978.

[Cho99] C.-T. Chou. The mathematical foundation of symbolic trajectory evaluation. In *11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 196–207. Springer, 1999.

[CKY03] E.M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference (DAC'03)*, pages 368–371. ACM, 2003.

[CKZR02] M.J. Ciesielski, P. Kalla, Z. Zeng, and B. Rouzeyre. Taylor expansion diagrams: A compact, canonical representation with applications to symbolic verification. In *Design, Automation and Test in Europe Conference and Exposition (DATE'02*, pages 285–291. IEEE Computer Society, 2002.

[CMP+01] A. Chatelain, Y. Mathys, G. Placido, A. La Rosa, and L. Lavagno. High-level architectural co-simulation using Esterel and C. In *Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES'01)*, pages 189–194, 2001.

[Cou78] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes*. PhD thesis, Université scientifique et medicale de Grenoble, 1978.

[Cou81] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

[Cou99] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[DB93] D. Déharbe and D. Borrione. A qualitative finite subset of VHDL and semantics. Technical Report RR943, IMAG Institute, 1993.

[DB95] D. Déharbe and D. Borrione. Semantics of a verification-oriented subset of VHDL. In *Correct Hardware Design and Verification Methods (CHARME'95)*, volume 987 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 1995.

[DB97] J. Dushina and D. Borrione. Formalisation and validation of the std_logic_1164 and numeric_std VHDL packages using the nqthm theorem prover. In *2nd Workshop on Libraries, Component Modeling and Quality Assurance*, pages 23–25, 1997.

[DBB+02] J.A. Darringer, R.A. Bergamaschi, S. Bhattacharya, D. Brand, A. Herkersdorf, J.K. Morell, I.I. Nair, P. Sagmeister, and Y. Shin. Early analysis tools for System-on-a-chip design. *IBM Journal of Research and Development*, 46(6):691–707, 2002.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages (POPL'82)*, pages 207–212, 1982.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[DSC98] D. Déharbe, S. Shankar, and E.M. Clarke. Model checking VHDL with CV. In *Proceedings of the 2nd International Conference on Formal Methods in Computer-Aided Design (FM-CAD'98*, volume 1522 of *Lecture Notes in Computer Science*, pages 508–514. Springer, 1998.

[EIA93] EIA (Electronic Industries Association). *Electronic Design Interchange Format (EDIF) Version 3 0 0 Level 0 Reference Manual, Std 618*, 1993.

[Enc95] E. Encrenaz. A symbolic relation for a subset of VHDL'87 descriptions and its application to symbolic model checking. In *Correct Hardware Design and Verification Methods*

(CHARME'95), volume 987 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 1995.

[Fer01] J. Feret. Abstract interpretation-based static analysis of mobile ambients. In *Static Analysis Symposium (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 412–430. Springer, 2001.

[Fer04] J. Feret. Static analysis of digital filters. In *13th European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2004.

[FM95] M. Fuchs and M. Mendler. Functional semantics for delta-delay VHDL based on focus. In C. Delgado Kloos and P. Breuer, editors, *Formal Semantics for VHDL*, pages 9–42. Kluwer Academic Publishers, 1995.

[Fos02] H. Foster. Unifying traditional and formal verification through property checking. In M. Sheeran and T. Melham, editors, *Designing Correct Circuits (DCC'02)*, 2002.

[GB03] A. Goel and R.E. Bryant. Set manipulation with boolean functional vectors for symbolic reachability analysis. In *Design, Automation and Test in Europe Conference and Exposition (DATE'03)*, pages 10816–10821. IEEE Computer Society, 2003.

[GB04] A. Goel and R. E. Bryant. Symbolic simulation, model checking and abstraction with partially ordered boolean function vectors. In *Computer-Aided Verification (CAV'04)*, 2004. Available at: http://www.cs.cmu.edu/ bryant/pubdir/.

[GBO02] P. Georgelin, D. Borrione, and P. Ostier. A framework for VHDL combining theorem proving and symbolic simulation. In *Proceedings of the 3rd International Workshop on the ACL2 Theorem Prover and its Applications (ACL2'02)*, pages 6–14, 2002.

[Gol94] D.M. Goldschlag. A formal model of several fundamental VHDL concepts. In *Proceedings of the 9th Annual Conference on Computer Assurance (COMPASS'94)*, pages 177–181, 1994.

[Goo95] K.G.W. Goossens. Reasoning about VHDL using operational and observational semantics. In *Correct Hardware Design and Verification Methods (CHARME'95)*, volume 987 of *Lecture Notes in Computer Science*, pages 311–327. Springer, 1995.

[Gor95] M.J.C. Gordon. The semantic challenge of verilog HDL. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 136–145. IEEE Computer Society Press, 1995.

[Gra91] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91)*, volume 1, pages 169–192, 1991.

[Gra92] P. Granger. Improving the results of static analyses programs by local decreasing iteration. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'92)*, volume 652 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 1992.

[Gro96] The VIS Group. Vis: A system for verification and synthesis. In *8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer, 1996.

[HDT87] S. Horwitz, A.J. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[HK01] J.M. Howe and A. King. Positive boolean functions as multiheaded clauses. In *17th International Conference in Logic Programming (ICLP'01)*, pages 120–134, 2001.

[HU03] C. Hymans and E. Upton. Static analysis of gated data dependence graphs. In *11th International Static Analysis Symposium (SAS'04)*, volume 3148 of *Lecture Notes in Computer Science*. Springer, 2003.

[Hym02a] C. Hymans. Checking safety properties of behavioral VHDL descriptions by abstract interpretation. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*. Springer, 2002.

[Hym02b] C. Hymans. Modular analysis of circuit description language by abstract interpretation: Application to the automatic extraction of circuit shapes. In *Designing Correct Circuits (DCC'02)*. ETAPS 2002, April 2002.

[Hym03] C. Hymans. Design and implementation of an abstract interpreter for VHDL. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *Lecture Notes in Computer Science*, pages 263–269. Springer, 2003.

[Hym04] C. Hymans. Verification of an error correcting code by abstract interpretation. Technical Report 2004-11, Laboratoire d'informatique, École polytechnique, Palaiseau, 2004.

[IEE87] IEEE (Institute of Electrical and Electronic Engineers). *VHDL Language Reference Manual, Std 1076*, 1987.

[IEE93] IEEE (Institute of Electrical and Electronic Engineers). *Multivalue Logic System for VHDL Model Interoperability, Std 1164*, 1993.

[IEE01] IEEE (Institute of Electrical and Electronic Engineers). *Verilog Hardware Description Language, Std 1364*, 2001.

[IEE02] IEEE (Institute of Electrical and Electronic Engineers). *VHDL Language Reference Manual, Std 1076*, 2002.

[Int03] International SEMATECH. *International Technology Roadmap for Semiconductors*, 2003. Available at: http://public.itrs.net.

[Jai97] A. Jain. *Formal hardware verification by symbolic trajectory evaluation*. PhD thesis, Carnegie Mellon University, 1997.

[Jon03] S.W. Jones. Exponential trends in the integrated circuit industry, 2003. Available at: http://www.icknowledge.com.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6, 1976.

[Kil73] G. A. Kildall. A unified approach to global program optimization. In *1st ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[KKD01] A. Kölbl, J.H. Kukula, and R.F. Damiano. Symbolic RTL simulation. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 47–52. ACM, 2001.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. North-Holland, 1952.

[LBPV94] J. Lohse, J. Bormann, M. Payer, and G. Venzl. VHDL-translation for BDD-based formal verification. Technical report, Siemens, 1994.

[LDG+02] X. Leroy, D. Doliguez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.06, documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique (INRIA), 2002.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *12th Annual ACM Symposium on Principles on Programming Languages (POPL'85)*, 1985.

[Mau94] L. Mauborgne. Abstract interpretation using TDGs. In B. Le Charlier, editor, *Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 363–379. Springer-Verlag, 1994.

[MBG94] W. Müller, E. Börger, and U. Glässer. The semantics of behavioral VHDL'93 descriptions. In *Proceedings of the European Design Automation Conference (EURO-DAC'94)*, pages 500–505. IEEE Computer Society Press, 1994.

[MC03] Y. Mathys and A. Châtelain. Verification strategy for integration 3G baseband SoC. In *Proceedings of the 40th Design Automation Conference (DAC'03)*, pages 7–10. ACM, 2003.

[MC04] N. Modi and J. Cortadella. Boolean decomposition using two-literal divisors. In *17th International Conference on VLSI Design (VLSI Design 2004)*, pages 765–768, 2004.

[MED02] MEDEA+ (Microelectronics Development for European Applications). *Design Automation Solutions for Europe*, 2002. Available at: http://www.medeaplus.org.

[MED04] MEDEA+ (Microelectronics Development for European Applications). *The Future of the European Microelectronics Industry*, 2004. Available at: http://www.medeaplus.org.

[Mil90] R. Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 1201–1242. Elsevier and MIT Press, 1990.

[Min01] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Programs as Data Objects (PADO'02)*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer-Verlag, 2001.

[Min02] A. Miné. A few graph-based relational numerical abstract domains. In *Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 117–132. Springer-Verlag, 2002.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*. ACM, 2001.

[MP03] P. Magarshack and P.G. Paulin. System-on-chip beyond the nanometer wall. In *Proceedings of the 40th Design Automation Conference (DAC'03)*, pages 419–424. ACM, 2003.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[RCK04] E. Rodríguez-Carbonell and D. Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *11th Static Analysis Symposium (SAS'04)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[Riv04] X. Rival. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 1–13. ACM, 2004.

[RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[Rus95a] D.M. Russinoff. A formalization of a subset of VHDL in the Boyer-Moore logic. *Formal Methods in System Design*, 7(1/2):7–25, 1995.

[Rus95b] D.M. Russinoff. Specification and verification of gate-level VHDL models of synchronous and asynchronous circuits. Technical Report 191608, NASA Contractor, 1995.

[SB95] C.-J.H. Seger and R.E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.

[SKH02] A. Simon, A. King, and J.M. Howe. Two variables per linear inequality as an abstract domain. In *Logic Based Program Synthesis and Tranformation (LOPSTR'02)*, pages 71–89, 2002.

[SM01] R. Sharp and A. Mycroft. A higher-level language for hardware synthesis. In *Correct Hardware Design and Verification Methods (CHARME'01)*, volume 2144 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2001.

[SSM04] S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using gröbner bases. In *Proceedings of the 31st Symposium on Principles of Programming Languages (POPL'04)*, pages 318–329. ACM, 2004.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *3rd International Conference On Formal Methods in Computer Aided Design (FMCAD'00)*, volume 1954 of *Lecture Notes in Computer Science*. Springer, 2000.

[SV00] D. Stewart and M. VanInwegen. Three notes on the interpretation of verilog. Technical Report 485, University of Cambridge Computer Lab, 2000.

[Tas93] J.P. Van Tassel. FEMTO-VHDL: The semantics of a subset of VHDL and its embedding in the hol proof assistant. Technical

Report 317, University of Cambridge Computer Laboratory, 1993.

[TE01] K. Thirunarayan and R.L. Ewing. Structural operational semantics for a portable subset of behavioral VHDL-93. *Formal Methods in System Design*, 18(1):69–88, 2001.

[Uni92] University of California, Berkeley. *Berkeley Logic Interchange Format(BLIF)*, 1992.

[vdWLGH99] P. van der Wolf, P. Lieverse, M. Goel, and D. La Hei. An MPEG-2 decoder case study as a driver for a system level design methodology. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES'99)*, 1999.

[VW86] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *1st Annual Symposium on Logic in Computer Science (LICS'86)*, 1986.

[WDB00] C. Wilson, D.L. Dill, and R.E. Bryant. Symbolic simulation with approximate values. In *Formal Methods in Computer-Aided Design (FMCAD'00)*, volume 1954 of *Lecture Notes in Computer Science*, pages 470–485. Springer, 2000.

[WM95] R. Wilhelm and D. Maurer. *Compiler design*. Addison-Wesley, 1995.

[Yi01] K. Yi. Yet another ensemble of abstract interpreter, higher-order data-flow equations, and model checking. Technical Memorandum 2001-10, Research on Program Analysis System, National Creative Research Center, Korea Advanced Institute of Science and Technology, 2001.