# INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

# Syntactic Model Reduction for Hardware Verification

Mohamed H. Zaki Hussein

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

April 2003

# ABSTRACT

## Syntactic Model Reduction for Hardware Verification

Mohamed H. Zaki Hussein

Microelectronics systems become more and more complex, making the detection of errors extremely difficult. Model checking, as a formal hardware verification technique, can potentially catch subtle hardware design errors. It is used to automatically verify temporal properties on finite state systems. However. model checking tools suffer from the state explosion problem; when the number of states representing the program under verification grows exponentially. Model reduction and model abstraction are approaches used to alleviate the state explosion problem. The main idea of model reduction is to suppress the states of the program while model abstraction is to interpret the behavior of the system at a different abstracted level with fewer states. Applying techniques that involve both approaches can greatly reduce the design under verification. In this thesis, we propose a syntactic model reduction and abstraction techniques in which, variables not influencing the property's variables are removed and the values domains of state variables in the system model are abstracted. This is done by partitioning these values into active and

deactive values according to their dependency to the property to be verified. After the above procedures are done, the state space of the reduced program is smaller than that of the original one, while the correctness of the properties are preserved. We have developed a tool called SynAbs which accepts a subset of the Verilog hardware description language and universally quantified computation tree logic (CTL) temporal properties, and generates a reduced Verilog code which can be fed into a CTL model checker. We have successfully applied our tool on the verification of a number of simple Verilog programs.

To my parents and my sister

# ACKNOWLEDGEMENTS

I have been very fortunate to have Dr. Sofiène Tahar as my supervisor. I am deeply grateful for his strong support and encouragement through out my Master's studies. His expertise and competent advice have shaped the character of my research. The finance he has ensured has facilitated me to actively concentrate on research.

It has been a great opportunity for me to work with Dr. Yassine Mokhtari. I am greatly grateful to him also.I extensively benefited from his deep knowledge and insight in the subject. Without his invaluable guidance and help, I could not have completed this work.

I also wish to thanks the examination committee members, Dr. Soleymani, Dr. Bois and Dr. Gohari for reviewing my thesis and giving me invaluable feedback.

My colleagues from the Hardware Verification Group (HVG), at Concordia University supported me in my research work. I want to thank them for all their help, support and valuable hints.

I would like to reserve my deepest thanks to my parents and my sister for their perpetual love and encouragement. Their life time support and encouragement has provided the basic foundation of any success I will ever achieve.

Everything I have is given by God, and my gratitude would always be due to Him.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF ACRONYMS

| | |
|---|---|
| AST | Abstract State Transition |
| ASIC | Application-Specific Integrated Circuit |
| ATM | Asynchronous Transfer Mode |
| CFG | Control Flow Graph |
| CNF | Conjunctive Normal Form |
| COI | Cone Of Influence |
| CTL | Computational Tree Logic |
| DDG | Data Dependency Graph |
| FSM | Finite State Machine |
| HDL | Hardware Description Language |
| HOL | Higher-Order Logic |
| LHS | Left Hand Side |
| LTL | Linear Temporal Logic |
| PS | Path Sequence |
| PVS | Prototype Verification System |
| RHS | Right Hand Side |
| ROBDD | Reduced Ordered Binary Decision Diagram |
| RTL | Register Transfer Lever |
| SAT | SATisfiability |

| | |
|---|---|
| SMV | Symbolic Model Verifier |
| SoC | System-on-a-Chip |
| SynAbs | Syntactic Abstraction Tool |
| VIS | Verification Interacting with Synthesis |
| VLSI | Very Large Scale Integration |

# Chapter 1

# Introduction

The use of digital systems in safety-critical applications such as space exploration and medical applications requires a high confidence in these systems. The consequences of wrong behaviour could be very costly. What happened to the Arian 5 missile may be considered as an example of this. A bug in its control software led to the destruction of the missile before going in a wrong trajectory [41]. The consequence resulted in a 500 million US dollars loss. Testing and simulation prove the existence of bugs in a design, but they do not prove their absence. The only way to ensure that a design is bug free is to fully test the whole design. However, in reality today's designs are too large and complex to ensure that they are fully verified. Over the last four years, the verification portion of the hardware design process has increased significantly, from about 30% - 40% to 50% - 80%, as shown in Figure 1.1 [20]. This makes verification the most significant part of the hardware

design development process.

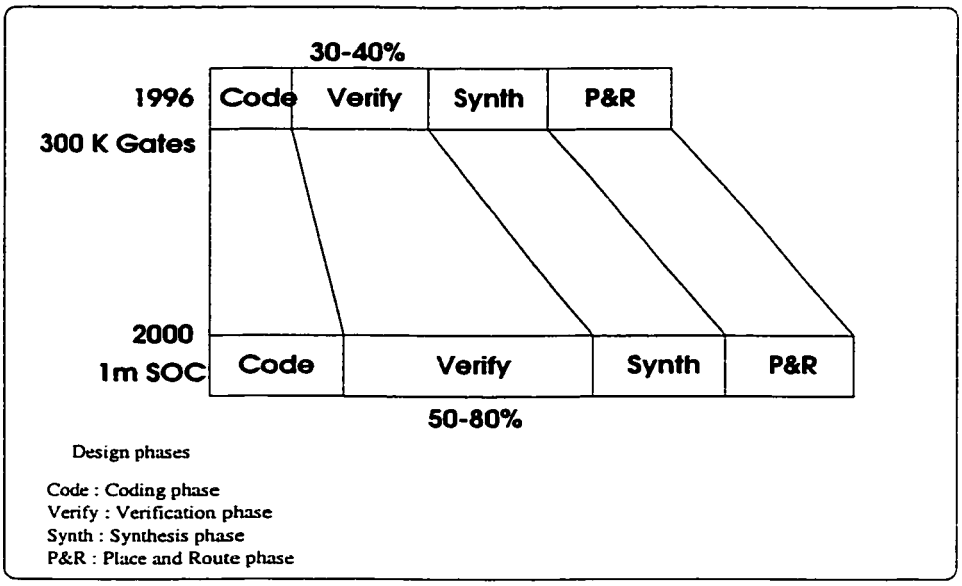


Figure 1.1: Design process [20]

A hardware design typically starts with a high-level specification, conveying the desired functionality, then the design flow goes through several steps from the specification to the final physical layout of the chip. After each step, a set of verification techniques is applied in order to verify that the new design step behaves according to its specification. Classically, the verification of the hardware designs is done in two steps. First, the design code is debugged to check for syntax violation then test-benches are written and fed to a simulator alongside the code to simulate and verify the behaviour of the design. With multi-million gates designs, detecting design errors for systems of such scale becomes extremely difficult and simulation becomes less efficient in terms of coverage and long simulation periods. It is common that even experienced engineers overlook some "corner" cases in the design phase.

Ignoring such cases may result in serious problems.

In recent years hardware verification teams had turned their attention to some formal verification techniques which can be applied alongside classical verification [35]. In such techniques, the correctness of designs is verified and proved mathematically.

## 1.1 Formal Hardware Verification

In general, formal verification consists of mathematically establishing that an implementation satisfies its specification [35]. The implementation refers to the system design which should be verified. It can represent a design description at any level of the system abstraction hierarchy. The specification usually refers to the reference model with respect to which correctness is to be determined. It can be expressed in a variety of ways, such as behavioral description, an abstract structural description, a timing diagram which reflects the behaviour of the system at different time points, a temporal logic formula, etc.

Formal verification techniques naturally group themselves into *theorem proving* methods and *automated finite state machine (FSM)* methods. The latter techniques are identified as *model checking* and *equivalence checking* of either combinational and sequential designs. In the next subsection, we overview the two major techniques, namely *theorem proving* and *model checking*.

### 1.1.1 Theorem Proving

In theorem proving, also known as proof based or syntactic methods, the designer constructs a mathematical proof, with the aid of some automated support, that a model or a structure meets its specification. With theorem proving, an implementation and its specification relationship, stated as equivalence or implication, is regarded as a theorem to be proven within the logic system, using axioms and inference rules. Specification and implementation models are expressed in some powerful formal language (first-order or higher-order logic), making theorem proving a powerful verification technique. It can provide a unified framework for various verification tasks at different hierarchical levels. A theorem prover or proof checker is a tool developed to partially automate the proof process or to check a manual proof. Theorem proving systems are being widely used for both hardware and software verification, on an industrial scale. Some of the well-known ones are HOL (Higher-Order Logic) [25] and PVS (Prototype Verification System) [15]. However, theorem proving is basically an interactive process, i.e., the task of proving complex theorems needs expertise, requires great effort and creativity on the part of the user which makes it a major difficulty for applying theorem proving on industrial designs.

### 1.1.2 Model Checking

The main FSM method is model checking [12], where the model of a design under verification is a kind of transition system describing all its possible behaviours. The

specification is a temporal logic formula property or an automaton written in a logic that is interpreted over the model by exhaustive exploration of the state space. By proving that the property holds on the transition system, we prove the correctness of the system.

The advantage of this technique is its automation. The verification is done without any required interaction between the user and the tool. In case the property does not hold, a counterexample describing the trace to the the failure point(s) is generated. Then the design can be corrected and re-verified. The general function of a model checker is described in Figure 1.2.

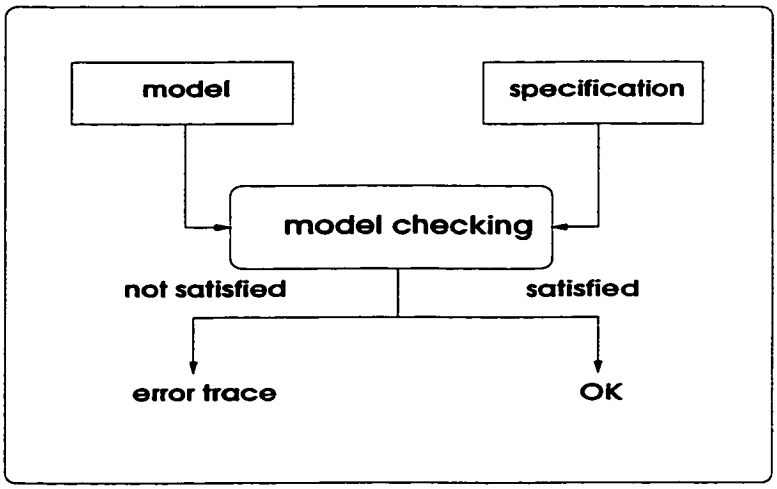


Figure 1.2: Model checking

A standard model-checking procedure is conducted in four steps as follow [29]:

1. Flatten a given design into a large but finite state-transition graph called Kripke Structure [12].

2. Specify graph properties as formulas of a linear or branching temporal logic.

3. Reduce the graph by detecting redundancies that are irrelevant to the requirement specification, for example, by abstracting data paths, or folding symmetries, or partially ordering independent events.

4. Explore the reduced graph (exhaustive state-space exploration).

The original model checking algorithms were based on the construction of the entire state transition graph (also known as reachability state graph). This method is known as explicit enumeration of the reachable state space [56]. Since model checking is based on the size of the state space of the design, state explosion is the main limiting factor of the technology. The efficiency of state exploration and model checking methods depends heavily on the size of the reachability state graph. The larger the reachability state graph, the more time and memory it takes to verify a system. But using symbolic rather than enumerative state representations [12], larger systems can be explored. In general, it is not strictly the size but the structure of the state space that determines the model-checking success or failure. In symbolic model checking [43], the structure of the system is represented implicitly, by means of Binary Decision Diagrams (BDDs) [7] . BDDs are data structures that represent Boolean functions (functions whose operands are Boolean vectors and the result is a Boolean value), and that can be used to represent sets of states and transitions between states.

Popular model checker tools include **SMV** (Symbolic Model Verifier) [37].

which performs symbolic model checking, **VIS** (Verification Interacting with Synthesis) [52], an integrated tool for **CTL** mmodel checking, simulation and synthesis of FSMs and **Spin** [30] mainly used for protocol verification.

## 1.2 Thesis Motivation

As the model to be created by the verifier becomes large, the state explosion problem (running out of memory or unacceptable verification time) arises. A number of state reduction approaches have been proposed to alleviate this problem by reducing the number of states under verification. The goal is to create a structure as small as possible, while maintaining the properties that are to be checked. Some of the known model reduction techniques include *partial order reductions*, and *abstraction techniques*.

- Partial order reduction [12, 48] is a fully automatic method aimed at reducing the size of the state space of concurrent programs, by inspecting the dependency relation that exists between the transitions of the system. It is based on the observation that execution sequences of a concurrent program can be grouped together into equivalence classes that are indistinguishable by the property to be checked. The reduced state-space generated contains at least one representative for each equivalence class. This method has aroused significant interest because of the good reductions obtained in many systems. However, for many practical systems, the reduction obtained by only partial

order reduction is still not sufficient.

- Model abstraction [12] is achieved by removing part of the design irrelevant to the property under verification, in such a way that the size of the model is reduced while the property is safely verified (property truth is preserved). In other words, the model is replaced by a non-deterministic state transition system that encapsulates the behaviour of the original model. The main requirement for the abstraction validity is that satisfaction over the abstract model implies satisfaction over the original model, i.e., if a property is not satisfied over the abstract model, it does not imply that this is the case for the original one. We call this implication *weak preservation* of the property under the abstraction.

Abstraction relies on the user to provide an appropriate abstraction algorithm to remove irrelevant information from a state space [14], and an abstraction mapping from an original state to an abstract state is required. Usually intervention from the user is required to guide the process. For larger systems, these human-involved reduction methods are infeasible and the main challenge facing the abstraction is related to its degree of automation. In order to automate the process, a series of refinement algorithms must be provided. So far, most of refinement algorithms are based on counterexamples analysis [10], where a counterexample trace is checked whether it is real or spurious (can be simulated on the original model or not). If it is not spurious, it is analysed and fed back with the abstract model to the abstraction

tool for refinement. The process stops when no real counterexample is generated. Such techniques are expensive due to the repetition of the model checking phase.

In this thesis, we present a fully automated syntactic model reduction tool, where the refinement process is based on extensive analysis of the program syntax of Verilog [55] designs prior to the verification, therefore avoiding the cost of the abstraction/verification iteration process.

Using the syntactic model reduction, one can reduce the size of the module under verification automatically, by traversing the syntactic structure of the module while preserving the correctness of the specifications. Syntactic abstraction has several advantages as mentioned below [46]:

- As the output of the abstraction tool is a syntactic description of the abstract program, other reduction methods based on BDD or partial reduction can be applied.

- It is more efficient than symbolic minimisation algorithms such as the one described in [10], where explicit transition system of the minimised system (represented symbolically) has to be constructed in order to apply the refinement algorithms. The output of a syntactic abstraction tool is a text file, which can be parsed and analysed by a model checker.

The whole approach is based on the value dependency analysis of the program variables against information extracted from the properties. Throughout the analysis, redundant data in the program (which includes values domains and program

variables) are reduced or abstracted, leaving only the part of the program needed for the verification.

The Verilog Hardware Description Language [55] was recommended as the modelling language, due to its popularity in the design and verification, and the fact that it is being supported by many model checkers. Two major factors affected the choice of the Verilog subset which is supported by our tool:

- A familiar synthesizable Verilog subset is advantageous, as it allows abstraction on designs used in real world.
- Can be accepted by popular model checkers, to avoid unnecessary translation prior to verification.

A comprehensive use of the proposed reduction tool is illustrated in Figure 1.3. Given a system model and a property, the abstraction process creates an approximation of that model. The approximation is simpler, yet it preserves enough information to make the verification process still meaningful. The approximated model obtained is then used to automatically verify the specification.

## 1.3 Thesis Overview

The rest of this thesis is organised as follow: Chapter 2 describes the required concepts and definitions. Chapter 3 gives an overview on static analysis techniques as well as some relevant works in the literature. Chapter 4 presents two model reduction

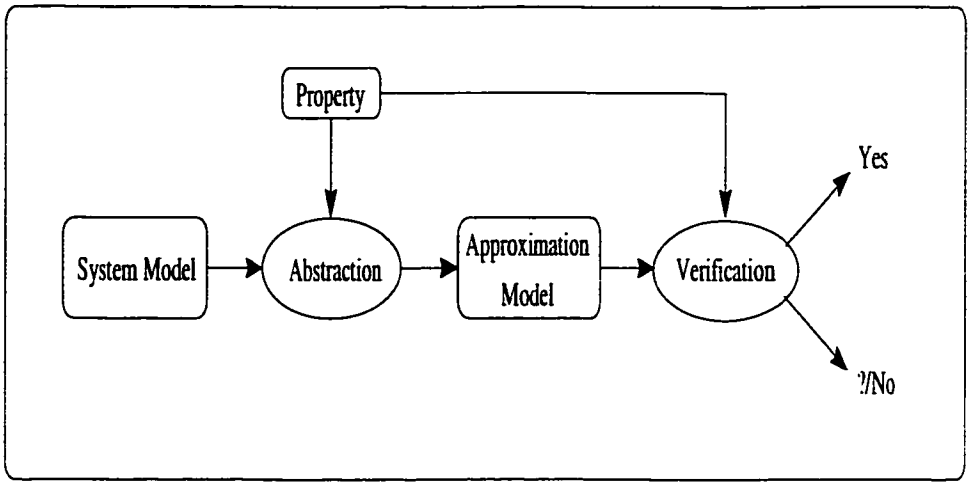


Figure 1.3: Abstraction and model checking

algorithm based on static analysis. Chapter 5 describes a tool implementing the model reduction algorithms as well as its performance evaluation. Finally, Chapter 6 concludes the thesis and highlights some future work.

# Chapter 2

# Preliminaries

In this chapter, we will start by introducing the Verilog-HDL subset to be used in this work. We will represent its syntax representation modelled by the control flow graph (CFG) and the data dependency graph (DDG). Next, we will propose a path dependency graph called *path sequence*, which is created by applying two semantic formulas on the CFG known as *reachability condition* and *state transformation*. We will show how this graph can be used to extract semantic information of the program. Finally, we will present the specification language represented by temporal logic formulas known as *ACTL*.

## 2.1 Verilog Subset

Verilog [4] is a hardware description language used to specify a circuit at low level structural elements as well as high level behavioral programming. It can be used to

specify concurrent and sequential designs at different level of abstraction. Verilog was only standarized in 1995 (see IEEE standard [55]). However, till this day, no complete standard semantic of the language exists. To verify a design by a model checker, a semantic representation of the design must be constructed (BDD or Kripke structure). Without a well defined standard semantic, different behaviors can be constructed for different model checkers. For this reason, we have chosen a synthesizable subset of Verilog supported by popular model checkers such as VIS [52], SMV [43] and FormalCheck [39].

### 2.1.1 Modelling with Verilog

In Verilog, a program can be described by a set of modules executed in parallel. A *module* (see Figure 2.1), consists of a number of threads, all executed concurrently. Threads can run continuously, like *always* procedural blocks or *continuous assignments*, or run only once, like *initial* procedural blocks. A variable in Verilog *input, output, internal variable* can be defined as a *register* or a *net*. A register corresponds to a variable whose size (in number of bits) can be explicitly declared. On the other hand, net (wire) represents the physical connections within a circuit and it simply carries a value from one part in the circuit to another. Unlike register which can be assigned within procedure, the only allowable assignment for variable of type wire is either through declaration or by using a continuous assignment statement *assign*. The right hand side expression on the continuous assignment is re-evaluated

every time one of its variables is changed. The result is the passed as new value to the assigned variable. The behaviour of a *continuous assignment* statement can be approximated as an *always* procedural block with only one statement.

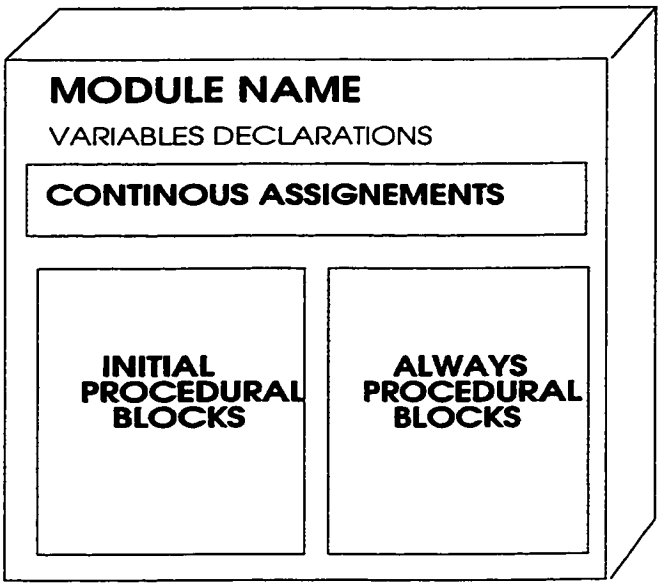


Figure 2.1: General Verilog model

Inside the procedural blocks, two types of procedural assignments can be used. The first is *blocking assignment* identified by the symbol " = ", where a right hand side expression is evaluated and the register variable is immediately updated and keeps its value unchanged until the next time step. The other type is *blocking assignment*, identified by the symbol "$\Leftarrow$", where the evaluated right hand side is not transfered to the register variable until the end of the current clock cycle. This is a preferable way to avoid any race condition that may arise inside a procedural block.

Event control is very important in the design, however it was not supported here, as the designs handled by this tools are assumed to be synchronous designs. Any guard commands can be changed to a conditional *if* statement as shown in Figure 2.2. Guard commands, like $x$ or $y$ block any execution of the thread until the condition in this guard is satisfied. Equivalently, the *if* statement in the figure skips all the statements within the thread until the condition is satisfied.

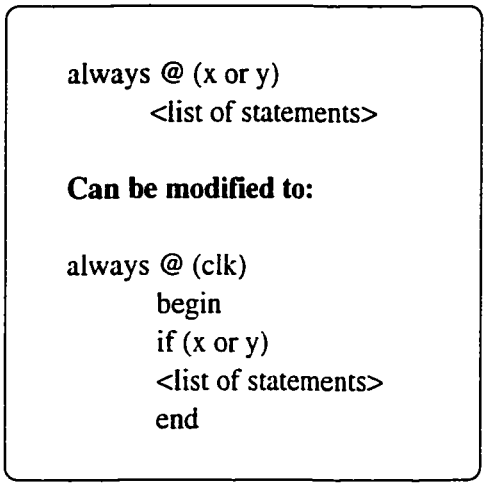


Figure 2.2: Event control in Verilog

We have identified a subset of behavioral Verilog for high-level synthesis to which we will restrict the discussion in the rest of the thesis. This subset is similar to the one used in [43]. We will assume that explicit use of time is not allowed in the behavioural specification. As a result, signal assignments cannot have explicit delay clauses. The only timing statement allowed is the *wait* statement. In the next section, we will present the syntax model of our subset of Verilog programs.

## 2.2 Program Syntax

To efficiently analyze a design written in a higher-level language such as Verilog, a graphical representation of the program syntax must be created. We follow the syntax representation proposed in [49], where a program can be modelled by two graphs; a *control flow graph* and a *data dependency graph.*

Flattening Verilog programs involves replacing all the Verilog program modules, by their threads such that the resultant program is one larger module containing all theses concurrent threads. Thereafter, a Verilog program is described as follows:

**Definition 2.2.1** *A Verilog program* P *can be specified as a* tuple $\langle V, I, A, S \rangle$, *where:*

- V $= \{v_1, \cdots, v_n\}$ *is a set of variables where variable* $v_i$ *has an associated finite domain of values i.e.* $dom(v_i)$.
- I *is an* **initial** *procedural block that specifies the initial state.*
- A *is a set of* **always** *procedural blocks where each block contains a set of statements S specified as follows:*

| | | | |
|---|---|---|---|
| $S$ | ::= | $v = e$ | *blocking assignment* |
| | \| | $v \Leftarrow e$ | *non-blocking assignment* |
| | \| | **wait** $(n)$ | *wait n cycles* |
| | \| | **if** $(B)$ $S_1$ **else** $S_2$ | *if statement* |
| | \| | **while** $(B)$ $S$ **end** | *while statement* |
| | \| | **case** $(v)$ $e_1 : S_1 \ldots e_n : S_n$ **endcase** | *case statement* |
| | \| | **begin** $S_1; \ldots ; S_n$ **end** | *block of sequential statements* |

*where n is a natural number, B is a Boolean condition and e (with subscript) is an expression.*

**Example 2.2.1** *The following program (see Figure 2.3) is a simple Verilog module. with only one* initial *and one* always *procedural blocks. The modules use the variables* in *as the input and* out *as the output. The values of* out *depends are changed depending on conditions on* in *as well as on some internal variables.*

### 2.2.1 Control Flow Graph

The control flow graph is a data structure representing the syntax of the program. It shows the sequence in which the program statements will be executed. Its size is usually proportional to the number of code lines. Control flow graphs were initially used in compilers as an intermediate model from the program text to its semantic model or from the program text to another program text.

```
module exp_1(in, out)

input in;
output [0:2] out;
reg [0:2] out;
reg [0:6] x, y , pc;

initial
      begin
      pc <= 0 ;
      x <= 0;
      y <= 0;
      out <= 0;
      end
always
      begin
      case(pc)
          'd00:
              begin
              if(x<100)
                  begin
                  pc = 0; y= y+1;
                  end
              else pc =1;
              x = x+1;
              end
      'd01:
          pc =pc+1;
      'd02:
          if(y == 100) out = in;
      endcase
      end
endmodule
```

Figure 2.3: A simple Verilog program

**Definition 2.2.2** *The control flow graph (CFG) of a Verilog program $P$ is a graph $\langle N, E, L, Initial, \omega, \varepsilon, SG \rangle$, where:*

- N *is a finite set of nodes labeled by the program counter locations $\ell_j, j \in \{1, ..., m\}$. A node in the CFG represents a location in the program and points to the (possible) next execution statement(s). By possible, we mean the case of a condition based on which an edge is chosen among different edges to be the next traversed edge.*

- E *is a finite set of edges, where each edge has labelling function $L : (E \rightarrow S)$.*

*An edge can be labelled with one assignment statement $S_a$ or it could be labelled with a test statement $S_t$.*

- *Initial* $\in N$ *is a specific node that denotes the beginning of initial block*
- $\omega \in N$ *is a specific node that denotes the beginning of the always procedural blocks.*
- $\varepsilon \in N$ *is a specific node that denotes the end of the always procedural blocks.*
- *SG is a set of subgraphs where each one of these graphs* $sg_i \in SG, i = \{1, ...., n\}$. *representing an always procedural block.* $\omega$ *and* $\varepsilon$ *nodes are connected to the* start *and* end *nodes of the subgraphs in a fork/join fashion through special parallel edges with no labelling.*

As demonstrated in Figure 2.4 [1]. The *initial* block $A$ begins at the *Initial* node and ends at the $\omega$ node, while *always* blocks $B$ and $C$ have $\omega$ as their starting node and $\varepsilon$ as their ending node.

The control flow graph is built in a standard way. For example, the node that represents *if statement* has two out-going edges labelled with the testing condition and its negation, pointing to the program locations of the *then* and *else* statements, respectively. Note that the *case* statement can also be rewritten in term of *if statement* but we have preferred to keep it in the abstract syntax since its presentation is

[1] In all the Figures in this thesis, "'Epsilon" corresponds to $\varepsilon$ and "Omega" to $\omega$, respectively

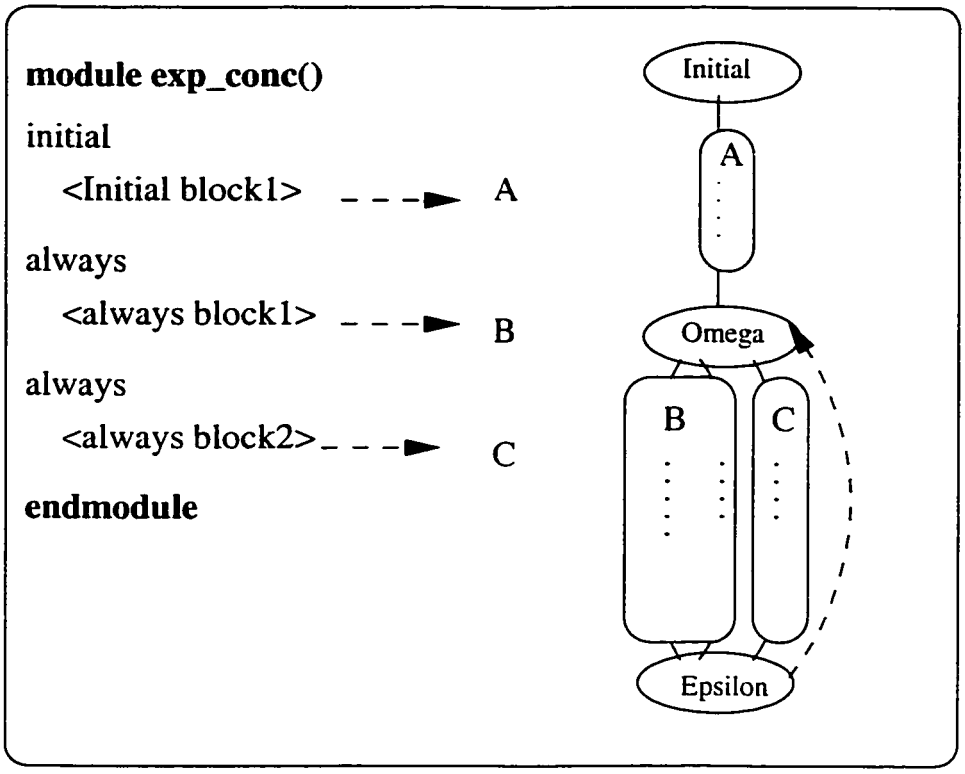


Figure 2.4: Structure of the CFG

more readable than its coding (we assume all the conditions are mutually exclusive). The CFG of the Verilog program in Example 2.2.1 is shown in Figure 2.5.

The control flow graph presented here shows only the possible execution paths for each program run. However, for the analysis, usually, a dependency representation between the program variables is needed.

### 2.2.2 Data Dependency Graph

Data dependency graphs (DDG) are graphs which represent the relationship between state variables. A DDG does not give any semantic information, only the type of dependency between variables, whether it is an assignment or control dependency.

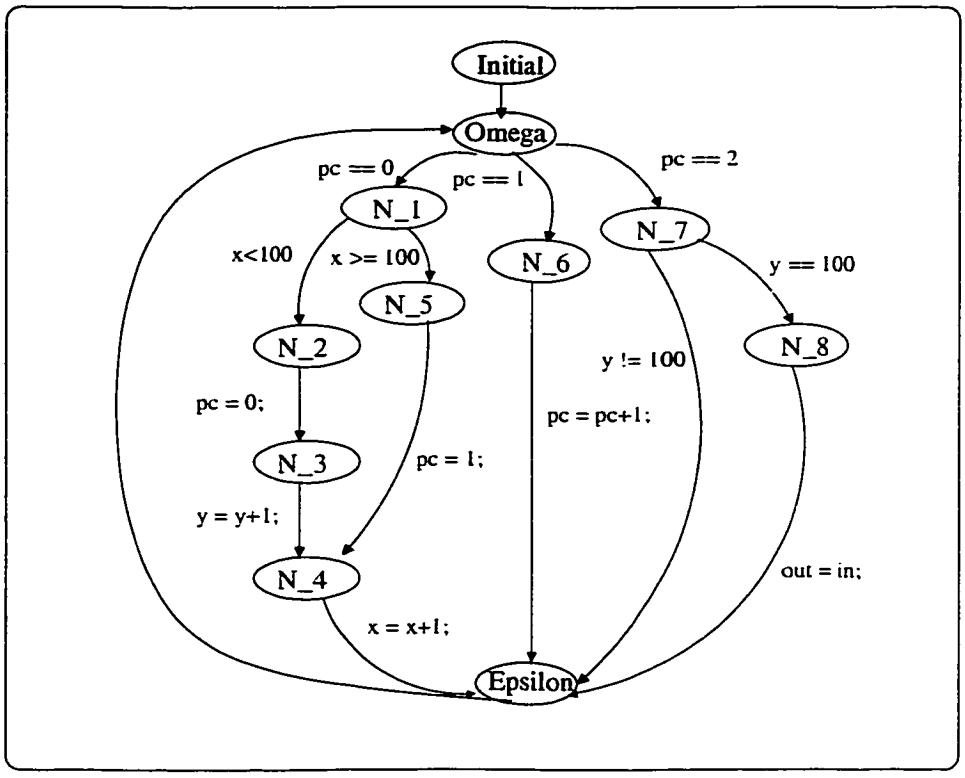


Figure 2.5: CFG of the Verilog Program in Example 2.2.1

The dependency between two variables is not always direct. It can be related through other variables. The direct dependency between variables can be divided into three categories:

- $v_1 \rightarrow v_2$: $v_1$ is used on an expression assigned to $v_2$.
- $v_1 \Upsilon v_2$: $v_1$ and $v_2$ are related by a relational operator.
- $v_1 \mapsto v_2$: $v_1$ is used on a test statement affecting the evaluation of $v_2$.

**Definition 2.2.3** *The data dependency graph DDG of a program* $P\ \langle V, I, S \rangle$ *is a directed graph* $\langle D, F \rangle$ *,where:*

- *D is a set of nodes, each labelled by a variable* $v_l \in V$.

- $F \subseteq D \times D$ *is a set of edges connecting the nodes based on one or more of the above three dependency categories. Each edge can be labelled by the types of dependency between the nodes' variables.*

The data dependency graph for the Verilog program in Example 2.2.1 is shown in Figure 2.6.

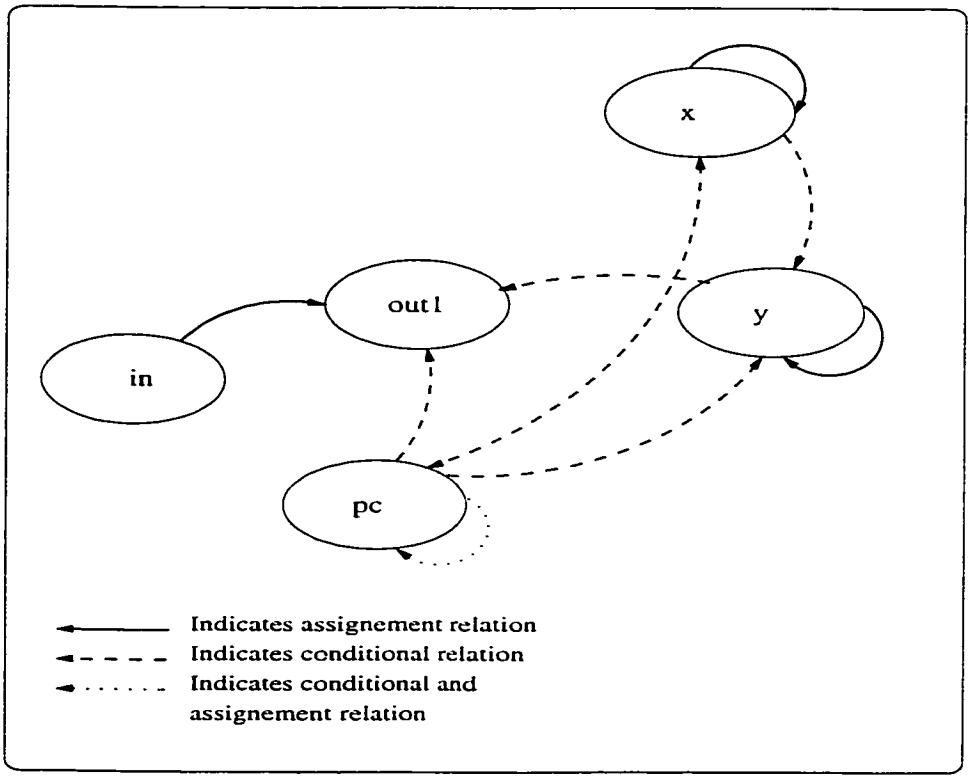


Figure 2.6: DDG of the program in Example 2.2.1

By inspecting the DDG, we can build the sets of dependency relations of any variable. For example, the set of assignment dependencies of *pc* is (*pc*), while its set of control dependencies is (*pc*, *x*); *pc* evaluation is depending on the truth

of condition statements involving the variables $pc$ and $x$. Using such graph helps identifying parts of the system that will affect the static analysis, as will be shown in the next chapter.

## 2.3 Program Semantics

The behaviour of a Verilog programs can be described using operational semantics [51]. We define a set of transition rules over configurations. A configuration has the form $\langle t, \lambda, \sigma \rangle$ where $t$ is a simulation time (a natural number), $\lambda$ is a program counter location pointing to the next statement to be executed, and $\sigma$ is A mapping from program variables to values: it denotes the final values of the program variables at the termination of a program statement.

**Definition 2.3.1** *Given an edge $\eta$ from a node $n$ labelled with location $\lambda$ and another node $n$ labelled with $\lambda'$, we define the transition rule according to the statement associated to the edge $\eta$, written $L(\eta)$ as follows:*

- $v = e \qquad \langle t, \lambda, \sigma \rangle \longrightarrow \langle t, \lambda', \sigma[e/v] \rangle$

  *where $\sigma[e/x]$ is the same as $\sigma$ except the value of the variable $v$ is now associated with the value $e$. Note that non-blocking assignment expressions can be translated into blocking assignment expressions by introducing intermediate variables. For example $a \rightarrow b$ has the same behaviour as the sequence $c = b,\ a = c$.*

- **wait** $(n)$

$$\frac{t < n}{\langle t, \lambda, \sigma \rangle \longrightarrow \langle t+1, \lambda, \sigma \rangle} \qquad \frac{t \geq n}{\langle t, \lambda, \sigma \rangle \longrightarrow \langle t, \lambda', \sigma \rangle}$$

- *a boolean condition*

$$\frac{\sigma(B) \text{ is true}}{\langle t, \lambda, \sigma \rangle \longrightarrow \langle t, succ^{+}(\lambda), \sigma \rangle} \qquad \frac{\sigma(B) \text{ is false}}{\langle t, \lambda, \sigma \rangle \longrightarrow \langle t, succ^{-}(\lambda), \sigma \rangle}$$

*where* $succ^{+}(\lambda)$ *and* $succ^{-}(\lambda)$ *are the true or false successor of* $\lambda$ *in a CFG, respectively.*

- $\lambda = \varepsilon$ *or* $\lambda =$ *Initial and* $\lambda' = \omega \qquad \langle t, \lambda, \sigma \rangle \longrightarrow \langle t+1, \lambda', \sigma \rangle$

- *Parallel Composition* $SG = [sg_1 \parallel ... \parallel sg_n]$ $\lambda = \omega$ *and* $\lambda' = \varepsilon$

$$\frac{\langle t, \lambda_i, \sigma \rangle \longrightarrow \langle t, \lambda_i', \sigma' \rangle}{\langle t, [\lambda_1 \parallel .. \parallel \lambda_i \parallel ... \parallel \lambda_n], \sigma \rangle \longrightarrow \langle t, [\lambda_1 \parallel .. \parallel \lambda_i' \parallel .. \parallel \lambda_n], \sigma' \rangle}$$

*where* $i \in 1, ..., n$.

Let an *action* $A$ be a statement of boolean expression within a procedural block $sg_i$ of parallel composition with shared variables $SG = [sg_1 \parallel ... \parallel sg_n]$ . This action can be identified as an atomic action if no other blocks $sg_j$, $j \neq i$ will be able to update this variable. Now assume that parallel blocks can share more than one variable. We will define an *atomic block*, as those blocks $sg_i$ with a sequence of atomic actions. It is considered as a critical section and it is executed without interruption. The atomic block has an exclusive access to the shared variables where no other

blocks can modify them during this period. Syntactically we will identify atomic blocks as statements enclosed in angle bracket, i.e $\langle sg \rangle$. As each process is identified as an atomic region $\langle sg \rangle$, its computation is reduced to one step computation. This reduction prevents interference from other parallel blocks.

$$\frac{\langle t, \lambda, \sigma \rangle \longrightarrow^* \langle t, \varepsilon, \sigma' \rangle}{\langle t, \langle \lambda \rangle, \sigma \rangle \longrightarrow \langle t, \varepsilon, \sigma' \rangle}$$

We have just defined the simulation semantic of the parallel composition. However, in static analysis no execution of the program will be done. Therefore, we will simplify the analysis by putting the constraints that variables can only be updated within one procedural block, while others procedural blocks can only use these variables. This allow us to treat the execution of the parallel blocks as a sequential execution, while the final state of the program will not be updated until all processes have been executed successfully [2].

### 2.3.1 Kripke Structure

The semantics of a program $P$ under verification is usually modelled by the finite state transition system known as *Kripke structure.*

**Definition 2.3.2** *The semantic of a program* $P = \langle V, I, S \rangle$ *is a Kripke structure* $\mathcal{K} = (\langle 0, Initial, \sigma_0 \rangle, S, \longrightarrow)$ *where:*

[2] Consider the parallel program $P \parallel Q$, where $P = (x = 1; y = x + 1)$ and $Q = (x = 2)$. The process Q can only be started at the beginning or at the end of the execution of P

- *$S$ is the set of configurations such that*

  $$S = \{\langle 1, \omega, \sigma_0\rangle : \langle 0, Initial, \sigma_0\rangle \longrightarrow \langle 1, \omega, \sigma_0\rangle\} \cup \{\langle n', \omega, \sigma'\rangle : (n, \omega, \sigma) \xrightarrow{*} (n', \omega, \sigma')\} \cup \{\langle 0, Initial, \sigma_0\rangle\}$$

- *$\langle 0, Initial, \sigma_0\rangle \in S$ is the initial configuration such that $\sigma_0(I)$ is true.*

*where $\xrightarrow{*}$ is the transitive closure of $\longrightarrow$, and $n$ and $n'$ are some natural numbers.*

The Kripke structure of the Verilog program in Example 2.2.1 is shown in Figure 2.7. The Kripke structure represents all the possible behaviours of the system. We can see from the figure how large such structure could be even for small examples. A binary representation (BDD) of this model is used instead of model checking. Although, such representation enlarges the limit of the design to be verified, it cannot solve completely the state explosion problem. Replacing the concrete model by an abstract one using static analysis techniques prior the verification tries to avoid this problem.

### 2.3.2 Reachability Condition and State Transformation

To determine the relation between states, we associate with every path $\pi$ in the CFG two semantics formulas [24] as follows:

**Definition 2.3.3** *A path in the CFG is a finite sequence of nodes $(n_1, \ldots, n_k)$, which begin by node $start_i$ and end with node $end_i$, where $i \in 1, 2, ..., k$ and $k$ is the number of* always *procedural blocks.*

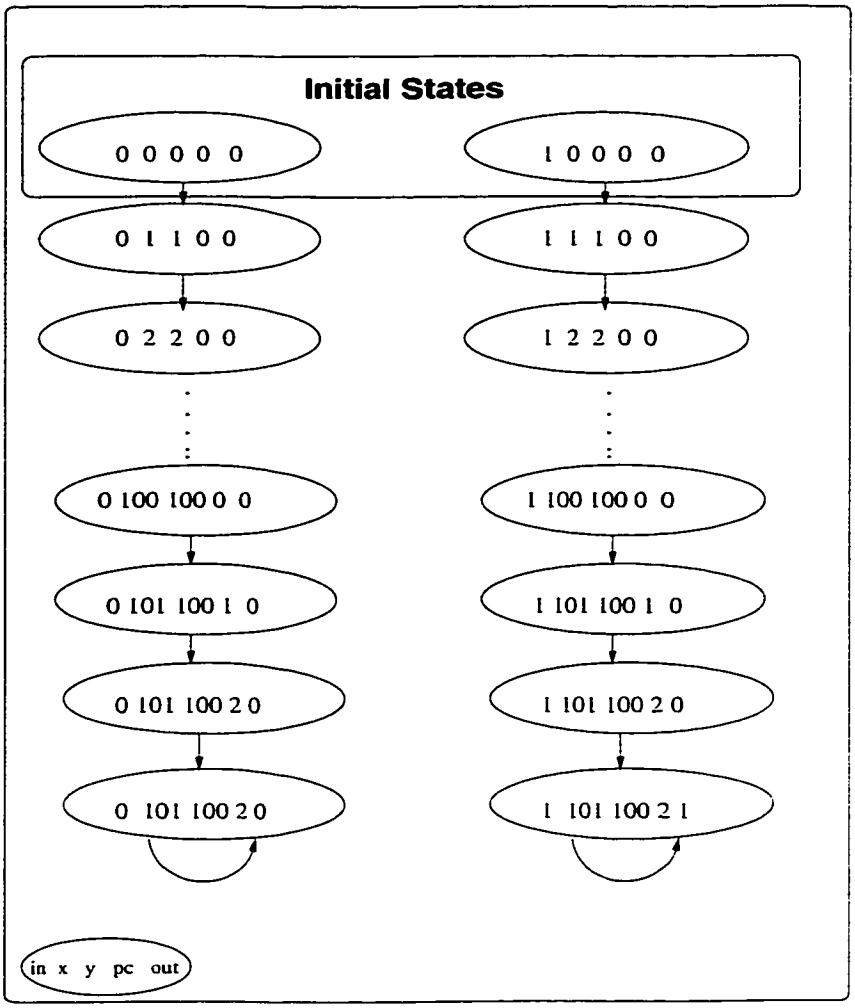


Figure 2.7: Kripke structure of the program in example 2.2.1

**Definition 2.3.4** *Let $V_\pi$ be a non empty set of variables, such that $V_\pi \subseteq V$.*

- *Reachability Condition $RC_\pi(V_\pi)$ guarantees that the path will be traversed, i.e., all the conditions on this path will be satisfied $RC_\pi(V_\pi) : C \rightarrow \{true, false\}$. where $C$ is the set of all possible states.*

- *State transformation $ST_\pi(V_\pi)$ which gives the final evaluation for $V_\pi$ under condition that $RC_\pi$ is evaluated to true.*

**Definition 2.3.5** *Let $\pi = n_1 \rightarrow \ldots \rightarrow n_m$ be a finite path in the CFG, we define $RC_\pi$ and $ST_\pi$ by induction and backwards*

- Induction basis*: $RC_\pi^m =$ **true** and $ST_\pi^m = V$. Being at the end of a path $\pi$, at time n, to traverse implies a true reachability condition and an identity state transformation.*

- Induction steps*: We define $RC_\pi^k$ and $ST_\pi^k$ according to the statements at $L(k, k+1)$*

  - v=e*: $RC_\pi^k = RC_\pi^{k+1}[e/v]$ $\quad ST_\pi^k = ST_\pi^{k+1}[e/v]$*

  - Boolean condition*: $RC_\pi^k = RC_\pi^{k+1} \wedge B$ $\quad ST_\pi^k = ST_\pi^{k+1}$ where $B$ is a Boolean function.*

*Finally $RC_\pi = RC_\pi^1$ and $ST_\pi = ST_\pi^1$.*

*Both functions $RC_\pi(V_\pi)$ and $ST_\pi(V_\pi)$ are computed syntactically by backward induction over the path $\pi$. We can represent any path in the CFG by a single transition as follows:*

$$\pi := RC_\pi(V) \wedge V' = ST_\pi(V)$$

Now, consider for example the paths $\psi_0$ and $\psi_1$ from Figure 2.5:

$\psi_0 : (Initial, \omega)$

$\psi_1 : (\omega,\ N_1),\ (N_1,\ N_2),\ (N_2,\ N_3),\ (N_3,\ N_4),\ (N_4, \epsilon)$

We want to check whether $\psi_1$ will be executed after $\psi_0$ or not. To achieve this, we apply backward computation on the paths to get the $ST_\psi$ and $RC_\psi$ at each edge. Once this is done, $RC_{\psi_0,\psi_1}$ and $ST\psi_0, \psi_1$ of both paths together are identified.

Let $V$ be the ordered set of the program variables $in, x, y, pc, out$, based on the above definition, we get:

- $RC_{\psi_0}^{Initial}\{V\} = true = RC_{\psi_0}, ST_{\psi_0}^{Initial}\{V\} = in, 0, 0, 0, 0, 0\} = ST_{\psi_0}$
- $RC_{\psi_1}^{\epsilon}\{V\} = true, ST_{\psi_1}^{\epsilon}\{V\} = \{in, x, y, pc, out\}$
- $RC_{\psi_1}^{N_4}\{V\} = true,\ ST_{\psi_1}^{N_4}\{V\} = \{in, x+1, y, pc, out\}$
- $RC_{\psi_1}^{N_3}\{V\} = true,\ ST_{\psi_1}^{N_3}\{V\} = \{in, x+1, y+1, pc, out\}$
- $RC_{\psi_1}^{N_2}\{V\} = true,\ ST_{\psi_1}^{N_2}\{V\} = \{in, x+1, y+1, 0, out\}$
- $RC_{\psi_1}^{N_1}\{V\} = true \wedge (x < 100) = (x < 99),\ ST_{\psi_1}^{N_1}\{V\} = \{in, x+1, y+1, 0, out\}$
- $RC_{\psi_1}^{\omega}\{V\} = (x < 99) \wedge (pc == 0) = (x < 99) = RC_{\psi_1},\ ST_{\psi_1}^{\omega}\{V\} = \{in, x+1, y+1, 0, out\} = ST_{\psi_1}$

We can see from the above that $\psi_1$ will be executed after $\psi_0$. Using the methodology in the above example, we will build what we call a *path sequence*, which represents the path dependency between the control flow graph paths.

## 2.4 Path Sequence

Unfortunately, it is not always possible to know statically whether a path of interest will be traversed during the execution. Let us consider the following example:

**Example 2.4.1** *Consider the program in Figure 2.8 and assume that a property to verify includes the expression "out == in". We need to check if this statement can be evaluated to* 0. *As* in *is assigned throughout the program more than one value, it is not clear, by just analyzing the CFG, which value(s) will reach this statement.*

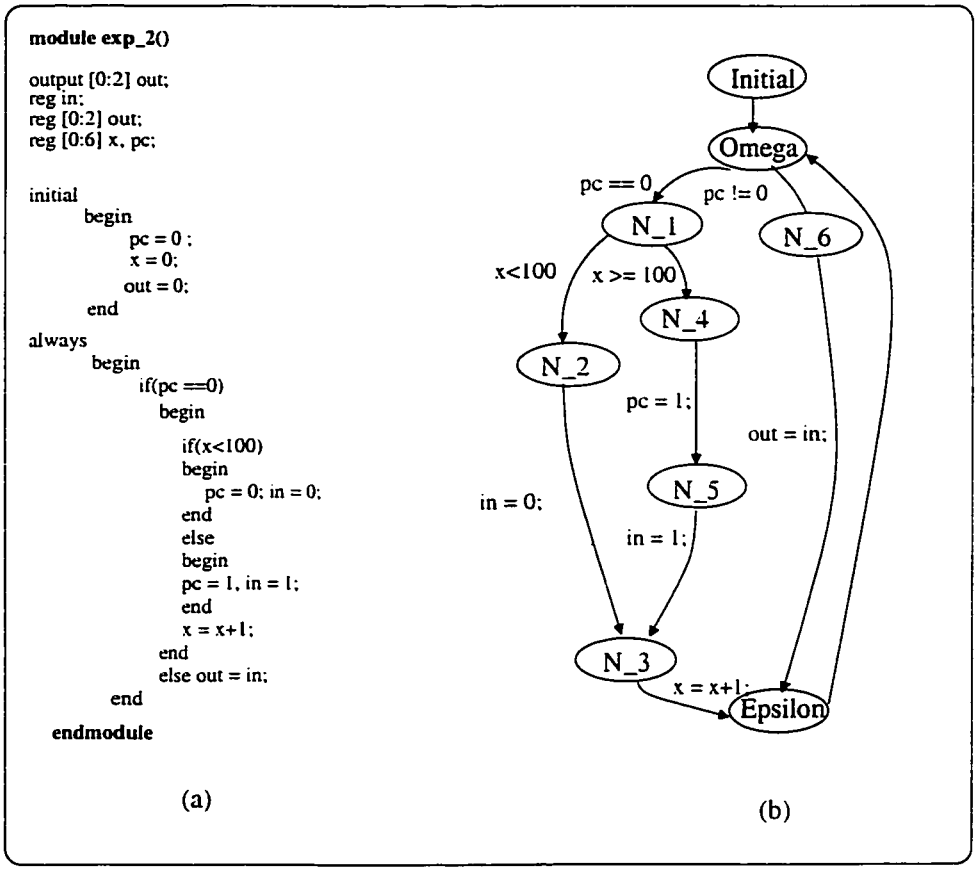


Figure 2.8: Verilog program and CFG of Example 2.4.1

We will refine the analysis by introducing the dependency between the CFG paths called *path sequence*. The path sequence provides a static check to ensure that some program state will be reachable. We will construct the path sequence by using the semantic functions, the reachability condition and the state transformation introduced in section 2.3.2.

**Definition 2.4.1** *The path sequence is a tuple* $\langle P_0, P, R \rangle$ *where*

- $P_0 \in P$ *is an initial path*
- $P$ *is a set of paths in CFG.*
- $R \subseteq P \times P$ *such that* $(\pi_1, \pi_2) \in R$ *iff* $RC_{\pi_1} \wedge RC_{\pi_2}[ST_{\pi_1}]$ *is not false.*

Verilog modules usually contain more than one *always* procedural block, executed concurrently. Therefore, a path sequence is created for each block. We can define the set of all the path sequences created for a program as follow:

**Definition 2.4.2** $PS_p$ *of a program* $p$ $\langle V, I, S \rangle$ *is a set of path sequences* $ps_j$, *where* $j > 0$ *is an identity for each* always *procedural block.*

Building a path sequence is recursive. Each node is a simple data structure, which keeps track of its successor and follower nodes. Starting from the initial (path) node, we identify its successor paths by adding all the paths with the $RC_\pi \neq true$ to the structure. The procedure is recursively applied on each (path) node on the data structure. However, in some cases, it is not possible to evaluate boolean conditions

using this technique; due to feedbacks and loops. We follow the routine used in classic static analysis techniques [28] which assumes that a condition is true as long as no available information that prove the inverse.

The procedure is approximate but safe; we are sure that no desired behaviour will be removed, but accuracy is affected as spurious behaviours can be created. Refinement is applied in an iterative way,; by traversing the path sequence more than once, in order to remove redundant links between node.

The path sequence of the Verilog program in Example 2.4.1 is shown in Figure 2.9. Remember that we needed to check whether $out == in$ will be evaluated to 0 or not. By examining the path sequence, we find out that the only value of $in$ that can be reached is 1, hence we conclude that *out* will never be evaluated to 0.

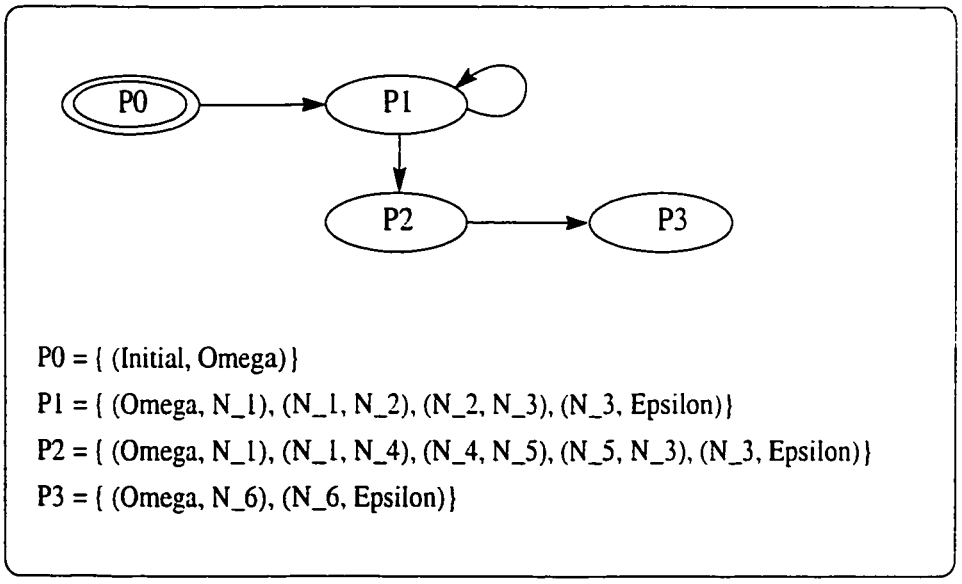


Figure 2.9: Path Sequence of the Verilog program in Example 2.4.1

Figure 2.10 shows the path sequence of the Example 2.2.1. For example there is a transition between $P_3$ and $P_4$ because $RC_{\pi_4}[ST_{\pi_3}] \wedge RC_{\pi_3}$ is $pc = 1 \wedge pc + 1 = 2 \wedge y \neq 100$ which could be true.

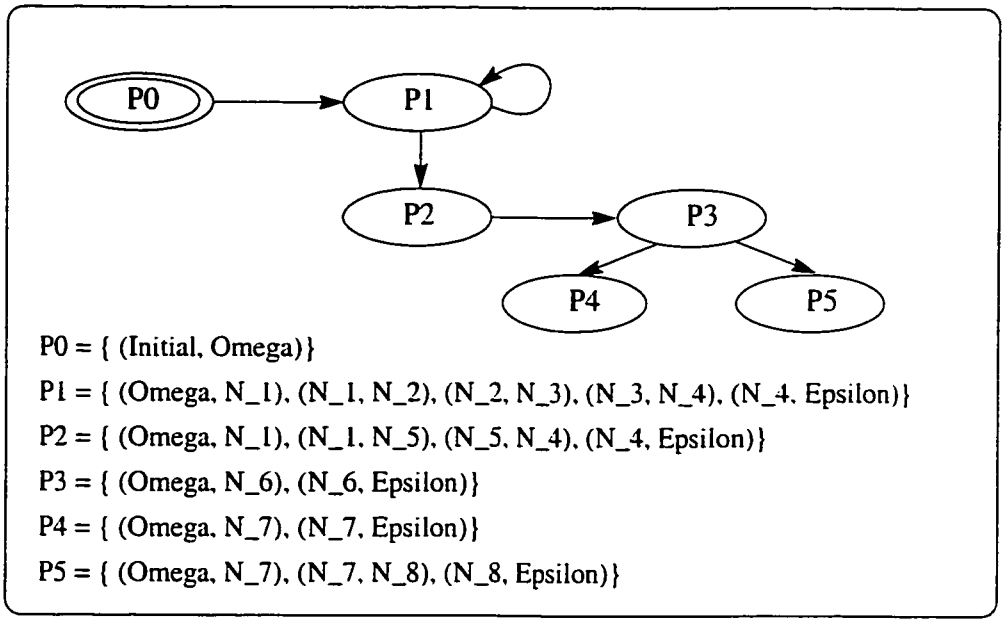


Figure 2.10: Path Sequence of the Verilog program in Example 2.2.1

From the above examples, we notice that the path sequence can analyse the dependency between the program variables in a way better than the approaches based only on the data and control dependencies. The idea of path sequence can also be used in order to remove redundant path; paths which will never be traversed, or detect design violation, as well as other analysis goals.

## 2.5 ACTL Specification Language

*ACTL* is a subset of *Computational Tree Logic (CTL)* [57], where only *universal* path quantifier, namely **A** is allowed. *ACTL* formulas are specified and evaluated over the semantic model of the system; usually modelled as a Kripke structure or a BDD representation. Beside Boolean connectives, *ACTL* provides linear time operators and path quantifier. The linear time operators allow expressing properties of a particular behaviour of the system given by a series of events in time. Path quantifiers, used with time operators, allow to take account the existence of multiple possible future scenarios starting at a given state at a point in time.

The main temporal operators describing properties of a path through the tree are [57]:

- **X** p ("next time") requires that a property $p$ holds in the second state of the path.
- **F** p ("eventually or in the future") asserts that property $p$ will hold at some states on the path.
- **G** p ("always or globally") specifies that property $p$ holds at every state on the path.
- p **U** q ("until") holds if there is a state on the path where property $q$ holds, and at every preceding state on the path, property holds $p$.

- p **V** q ("release") is the global dual of **U**. It requires that property $q$ holds along the path up to and including the first state where property $p$ holds. However, the first property is not required to hold eventually.

Based on the path quantifiers and temporal operators, we can define state formulas and path formulas as follows.[3] is an example of a state formula.

**Definition 2.5.1** *Let $AP$ be the set of atomic propositions. The* ACTL *is the set of state formulas on $AP$ inductively defined as follow:*

- *Any Boolean formula over atoms from $AP$ using the connectives $\vee$, $\wedge$ and $\neg$ is a pure state formula.*
- *If $\phi$ and $\varphi$ are* state formulas, *then $\phi \wedge \varphi$ and $\phi \vee \varphi$ are state formulas.*
- *If $\phi$ and $\varphi$ are* state formulas, *then $\mathbf{X}\phi$, $\mathbf{F}\phi$, $\phi\mathbf{U}\varphi$ $\phi\mathbf{V}\varphi$ are path formulas.*
- *If $\phi$ is a path formula, then $\mathbf{A}(\phi)$ is a state formula.*

**Definition 2.5.2** *A path $\pi$ of a Kripke structure $M$ is a finite sequence of states $\pi = [c_0, c_1, ...., c_i]$ such that $i \geq 0$. Given an integer $i \geq 0$ and a path $\pi$, we denote by $\pi_i$ the $i$-th state of $\pi$.*

**Definition 2.5.3** *Let $c$ and $\pi$ be a generic state and a path in the Kripke structure model $M$, respectively. The satisfaction relation $\models$ for state and path formulas is defined as follow :*

[3] "state formulas" reason about things that are true at one point in time, and "path formulas" about things that are true for a sequence of steps. "p **U q" is an example of a path formula while " X x"**

- $c \models p$ *iff* $p \in L(c)$ *where* $L(c)$ *is the labelling function of state* $c$
- $c \models \neg\varphi$ *iff* $\neg c \models \varphi$
- $c \models \varphi \wedge \psi$ *iff* $c \models \varphi$ *and* $c \models \psi$.
- $c \models \varphi \vee \psi$ *iff* $c \models \varphi$ *or* $c \models \psi$.
- $c \models \mathbf{AX}\varphi$ *iff for every path* $\pi$ *starting at the state* $c$, $\pi_1 \models \varphi$.
- $c \models \mathbf{A}(\varphi\ \mathbf{U}\ \psi)$ *iff for every path* $\pi$ *starting at the state* $c$, *there exists* $i \geq 0$ *such that* $\pi_i \models \psi$ *and* $\pi_j \models \varphi$ *holds for all* $j < i$.
- $c \models \mathbf{A}(\varphi\ \vee\ \psi)$ *iff for every path* $\pi$ *starting at the state* $c$, *and every* $j \geq 0$, *if for every* $0 \leq i < j, \pi_i \not\models \varphi$ *then* $\pi_j \models \psi$.

## 2.6 Summary

In this chapter, we have introduced the synthesizable subset of the Verilog to be considered in this thesis. We have shown its syntax representation by control and data flow graphs. We also presented the kripke structure as the semantic model of Verilog programs. In order to overcome the complexity of building this Kripke structure, we proposed an abstract state transition system. we called *path sequence*. Finally, we presented the ACTL the specification language to be used for properties description.

# Chapter 3

# Static Analysis

## 3.1 Introduction

Static analysis [13] is one of the program analysis techniques that was initially used in software debugging and testing. Recently, researchers started to investigate its use in the verification and abstraction of software and hardware designs. Static analysis means the analysis of the program semantics to extract information about the system at compile time such that properties can be proven about the system without the need to actually run the program. Widely used static analysis techniques for hardware verification includes *dataflow analysis* [28], where the property is checked by solving a dataflow problem on a graphical representation of the program, *abstraction* [14] of the model of the system under verification and *slicing* [58] to collect the semantics at a specific program point.

In this chapter, we will give an introduction to static analysis techniques that influenced the model reduction algorithms implemented in our tool. We will begin by introducing dataflow analysis and program slicing, which is its application on program verification. We will then give a brief overview about model abstraction and its use in model checking.

## 3.2 Data Flow Analysis

The basis of data flow analysis is the control flow graph (CFG) [28]. Dataflow analysis lets information be propagated forwards (or backwards) through the CFG. A program point might have a number of predecessors (or successors); the information of these different predecessors (successors) must be combined. The combination operation is unusually a union or an intersection.

Data flow analysis encompasses a wide range of static analysis techniques, which are used in code generation, optimisation and debugging. In data flow analysis, one is interested in how data flows through a program to obtain a particular information. John and Nielson [59] classified the different analysis techniques according to the behavioral properties on which they depend. The following are some of the techniques of our interest:

**Set of values, or environment that can occur at a program point**

- *Constant propagation.* By using forward reachability analysis on the CFG, it computes, for every point in the program, which variables have a constant

value that can be determined statically. This allows to simplify expressions with their values. Having constants in code allows easier analysis afterwards.

### Computational past

- *Use-definition chains* associates with a reference to variables $x$ the set of all assignments $x := ...$ that assigns values to $x$ that can reach the reference following the possible flow of program control.

- *Available expressions.* By using forward reachability analysis on the CFG, it detects which previously evaluated expressions are still valid, (no variables used in the expressions have been re-assigned since the expression was evaluated) it computes, for every point in the program, which expressions are available, i.e. whose values have already been computed; their value can be available in some register, and can be copied from there.

### Computational future

- *Live variables.* By using backward reachability analysis on the CFG, it computes, for every point in the program, which variables are alive, i.e. whose value in the point is used by the following code; assignments to variables that are not alive can be safely removed from the program.

- Very busy expressions: By using backward reachability analysis on the CFG: it computes, for every point in the program, which expressions appear in all the computations starting from there.

### 3.2.1 Slicing

One of the direct application of data flow analysis is program slicing [58]. Program slicing is a technique for simplifying programs by focusing on selected aspects of semantics and deletes those parts of the program, which can be determined to have no effect upon the semantics of interest [58]. Slicing is an automated technique for program simplification, which can aid debugging and maintenance.

There are two main properties desirable in slicing algorithms: *soundness* and *completeness*. In order to be *sound*, an algorithm must never delete a statement from the original program, which could have an effect upon the slicing criterion. It is this property that enables us to analyse a slice that contains all the statements relevant to the criterion. Soundness alone, however, is not enough: a trivial slicing algorithm could refuse to delete any statement from a program, thereby achieving soundness by doing nothing. In order to be *complete*, a slicing algorithm must remove all statements which cannot affect the slicing criterion. In general, completeness is unachievable (for theoretical reasons dealing with undecidability [58]). Therefore, the goal of slicing algorithms is to delete as many statements as possible, without giving up soundness. The closer an algorithm approximates completeness, the more precise the slices it constructs will be.

Slicing algorithms can be divided into two basic categories: *Static slicing* and *dynamic slicing*. Each has its own algorithms such as forward and backward reachability analysis and techniques for different applications like *interprocedural* [31],

*intraprocedural* [60] and *concurrent programs* [44]. Researchers have even tried to come up with *hybrid* ideas between the two categories [26].

Usually, analysis is done on program dependence graph [23] where control and data dependence edges are added. Although such representation shows relation between statements, it does not show any order of path execution that is important for nonterminating program analysis. For threaded programs [38], interference dependence are explicitly represented. Interference occurs if a variable is defined in one thread and referenced in another parallel executing thread.

### Slicing Techniques

1. **Static Slicing**

   The static slice [58] is the simplest form of slicing. A slice is constructed by deleting those parts of the program that are irrelevant to the values stored in the chosen set of variables at the chosen point. The point of interest is usually identified by annotating the program with line numbers which identify each primitive statement and each branch node. Having picked a slicing criterion, we can construct one of two forms of slices: a backward slice or a forward slice. A backward slice contains the statements of the program which can have some effect on the slicing criterion, whereas a forward slice contains those statements of the program which are affected by the slicing criterion. The slicing criterion contains no information about the execution of the program; thus the slice

preserves the effect of the code for every possible execution pattern, i.e., all conditions on all paths are assumed satisfied.

To see how static slicing works, consider the simple example fragment of *C* code in Figure 3.1(a). Suppose we only care about the effect of this fragment of code has on the variable $z$ at the end of the program. We can construct a backward slice on $z$ at the end of the program to focus the attention on this aspect of the fragment. The slice on $z$ at the end of the program is shown in Figure 3.1(b). It is easy to see that the line $r = x$; cannot affect the final value of $z$ and this is why it is not included in the slice. The assignment $z =$ *y-2*; also has no effect upon the final value of $z$, and is not included in the slice too.

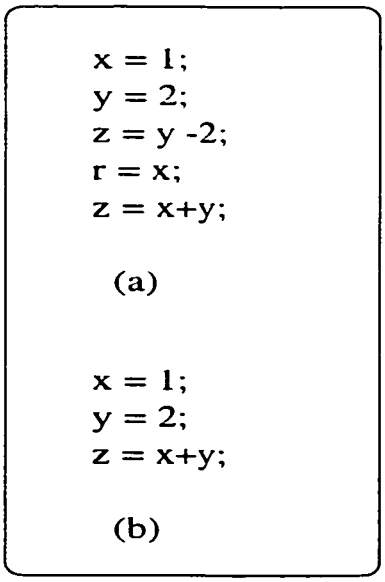


Figure 3.1: Static slicing example

## 2. Dynamic Slicing

In dynamic slicing [2], only the dependence in a specific execution is taken into account. A dynamic slicing criterion specifies the input. The difference between static slicing and dynamic slicing is that dynamic slicing assumes a fixed point for a program, whereas static slicing does not make assumptions regarding the input. One way to construct a dynamic slice is simply to trace the execution of the program for the input we are interested in and then to remove from the corresponding static slice those statements which were not actually executed. This technique is obviously sound, because an unexecuted statement cannot affect the slicing criterion. Suppose we want to slice the program shown in Figure 3.2(a) according to the criteria $(y,\ n = 2)$; which means the variable of interest is $y$ and the initial value for $n$ is 2. By executing the program once, we can find that the statement $x = 18$ will not be reached. so it can be removed (see Figure 3.2(b)).

3. **Hybrid Slicing**

Static slicing lacks accuracy since it is based on all executions that reach a given program point rather than specific execution under program point. Dynamic slices are precise but require a large amount of run time overhead due to the tracing information that is collected during the program execution. The accuracy of the analysis can be improved by incorporating dynamic information (breakpoints) into the static slicing [26]. This is done by more accurately estimating the potential paths taken by the program, and eliminating the other

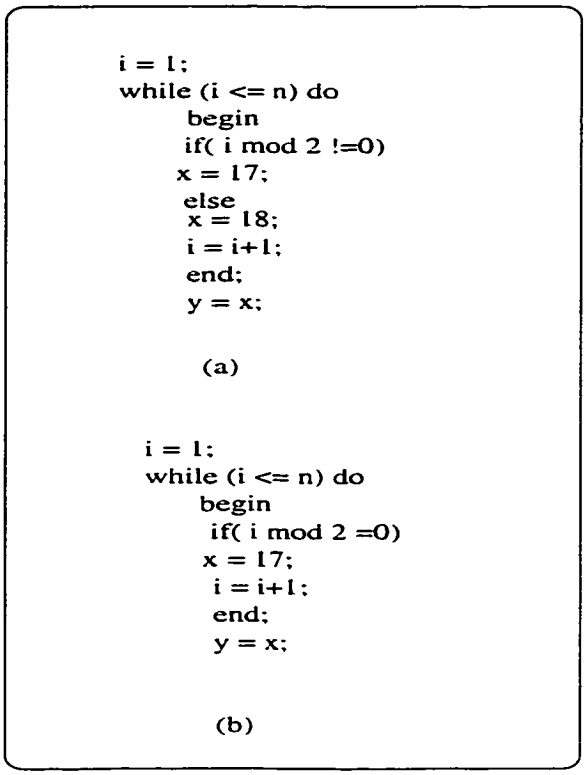


Figure 3.2: Dynamic slicing example

paths thus reducing the run-time overhead. The hybrid slicing, hence, is more precise than the static analysis and less costly than the dynamic slice.

### 3.2.2 Related Work

There are a few works dealing with slicing application for *hardware* programming languages. For instance, the work presented in [34] applies program slicing to VHDL programs [54]. In order to analyse the programs, along the data and control dependencies, a new dependency, called signal dependence, is defined for inter-process dependency. A slice follows from these dependencies (though the details of the slice are not defined). Few details are presented to support the sufficiency of the model

or to demonstrate its correctness.

Clarke *et al.* [9] extends program slicing to VHDL in the context of a verification tool that can reduce the VHDL code to be analysed. This is a direct application for static slicing of sequential system with the introduction of a new dependence graph, called interference graph, which represents the dependencies that may arise in concurrent procedure. The analysis of the program is done on the system dependency graph which is formed of data, control and interference dependencies. The approach is preliminary and there is not much discussion about the semantic mapping of VHDL.

Other related work includes the work done by Baresi *et al.* [53], where they used data flow analysis techniques in order to identify deadlocks conditions within VHDL specifications. Hsieh *et al.* [32] , applied data flow analysis techniques in their model abstraction algorithm, in order to abstract VHDL semantic.

## 3.3 Abstraction

Model abstraction is one of the most important techniques for improving the state explosion problem. The majority of the abstraction techniques are based on Abstract Interpretation. Abstract interpretation [14] describes a general framework for static analysis. It is semantic based, and hence supports correctness proofs of program analysis. Abstract interpretation performs program's computation using abstract values instead of concrete values. One reason for using abstract values instead

of concrete ones is to transform uncomputable problems to computable ones or computable problems into more efficiently computable ones. Another reason is to deduce facts that hold on all possible paths and different inputs. In [59], John and Nielson used the following analogy to describe abstraction:

*Abstract interpretation is to formal semantics as numerical analysis is to mathematical analysis*

The main idea of model abstraction is to find a map between the actual set of values of state variables and a small set of abstract values such that a simulation relation (a mathematical relation) exists between the original transition system and the newly created one. In his overview paper, Dams [17] tried to define abstraction in the model checking context in terms of theory, methodology, techniques and tools.

- **Abstraction theory** formalises the conservative approximation of the semantics of hardware systems. The semantic is a kind of mathematical model of the system representing all its possible behaviour under certain environment (Initial Conditions for example). By approximation, we mean the observation of the semantics at some level of abstraction ignoring irrelevant details to the property to be checked. In model checking, the concrete system under verification is transformed to a more abstract and smaller one . Both systems are connected by an abstraction relation which is *safe* with respect to a given property $\varphi$, namely it preserves the property. This means that if the property holds for the abstract system, it holds for the concrete one as well.

- **Abstraction methodologies** deal with abstraction process: They answer questions like *How to get a correct abstract system (regarding the property) using the information of the concrete system and the property to be checked?* Ensuring such a correct abstract system requires the formalisation of relation between the semantic (state transition systems) model of the concrete and abstract systems. Methodologies, commonly used, are iterative process where the abstract system is refined until its behaviour simulates the original system. Refinement can be done through reachability analysis on the program representation [61] or by inspection of the counterexample produced by the model checker. Refinement in this case is done by analysis of the data extracted from spurious counterexamples.

- **Abstraction techniques** were developed to build the abstract model by analysing the concrete model and the specification. Some techniques were based on analysing the syntactic format of the program [61], others by analysing the semantic model [47]. Some techniques were oriented towards reducing the set of variables used, like cone of influence [40] or variable hiding [3], others on reducing the values domains of the variables by replacing the concrete domain by an abstract domain like data abstraction [11] or by replacing the statements in the program by Boolean expressions like predicate abstraction [46, 18, 19].

- **Abstraction tools** designed to implement these different techniques. For example VIS [52], SMV [43], Bandera [21], Autoabs [45]. Usually the abstraction

tool accepts as input the model written in a description language (Smv [43], Blif-mv [52], Verilog ) and the property written in a specification language, like temporal logic formulas (e.g., CTL or LTL). Based on the model and the property analysis, the tool performs the abstraction and generate at the back end the abstract model which can be fed to a model checker.

### 3.3.1 Property preservation

The main challenge facing abstraction is the trade off between the degree of automation and the effectiveness of the abstraction. Interactive abstraction can lead often to powerful abstraction hence smaller state transition systems than automatic abstraction algorithms which are more general. There are two types of property preservation: *Weak preservation* and *Strong preservation* [6, 16].

**Weak preservation**: When a set of properties true in the abstract system has corresponding properties in the concrete system, that are also true.

An important issue of weakly preserving abstractions is *over approximation.* Over approximation of the behaviour of the system occurs when more behaviours are added than present in the concrete system. In such case, when property is true for these behaviour, we ensure that it is true for a subset of these behaviours. The problem encountered in dealing with over approximation (accuracy problem) are analogous to the problems encountered when attempting to make static analysis precise. Usually, behaviours that invalidate a property are displayed in the form of

counterexamples. By inspecting these counterexamples, we can discover that this is a spurious errors (spurious counterexample) and refinement algorithms work on readjusting the abstraction to eliminate such spurious results. However, sometimes we can apply fairness constraints to force some paths to be traversed, in this way we can avoid wrong behaviours.

***Strong preservation.*** When a set of properties with truth values either true or false in the abstract system has corresponding properties in the concrete system with the same truth values, i.e., the abstract program satisfies a property if and only if the concrete program satisfies it. *Under approximation* abstraction techniques allow strong preservation of the properties. In under approximation abstraction, the model is reduced by removing behaviours when going from the concrete to the abstract system. Examples of such technique are *variables hiding* [3] and *the cone of influence* reduction [12].

Although strong preservation is more useful in practice, it does not allow efficient abstraction of the system. On the contrary, weak preservation allow more aggressive abstraction hence allows verification of larger state space systems, therefore it is more popular for verification [16].

From the above definitions, it is possible that an abstraction is exact with respect to a given specification but conservative with respect to a different specification.

One might argue about the difference between model reduction and model

abstraction in the model checking framework. In model reduction techniques, parts of the system not affecting the property are removed, and the state transition system of the reduced program is a compressed version of the original one while in model abstraction, parts of the system not affecting the property are replaced by a smaller representation.

### 3.3.2 Related Work

Many abstraction techniques have been implemented to help the verification of hardware and software designs. The degree of automation, the degree of abstraction and property preservation are major factors that affect the implementation and the usage of the abstraction. Many surveys, discussing those issues, can be found in the literature [13, 16, 17, 47].

Among abstraction techniques are *syntax directed* abstractions where static analysis is applied and the reduced model is created without the need to build its full transition system.

In [46], Namjoshi and Kurshan extended and automated a syntax abstraction approach, called *predicate abstraction*, which translates a variable with large value domain into a set of predicates. The produced output model is a reduced program text. The algorithm proposed was implemented, in a tool called **AutoAbs** [45]. Although, the technique was claimed successful in verifying mid-size designs, the disadvantage of the implementation is that error traces are given in terms of Boolean

values for each predicate and are therefore, hard to analyse and use for the refinement of the original model. A similar implementation of Predicate abstraction was applied on VHDL programs as proposed in [5].

Karen [61] in her thesis, proposed two forms of syntactic abstraction: the first called *path reduction* is based on suppressing the control flow graph paths that does not affect the property; the second called *dead-variable reduction*, where she excludes some of the successors of states where the variables of interest are not used. However, the abstraction we will propose in this thesis is more aggressive in the sense that not only the paths not affecting the variables of interest are removed, but also the domain of variables of interest abstracted.

Similar work of the state suppression (dead-variable reduction) was proposed in the dataflow analysis context; *live variable analysis* [42] where a variable is live if it is used before it is redefined. Fixpoint analysis are applied on the CFG to compute live variables. Such reduction is based on property independent analysis which make it orthogonal to other syntactic abstraction techniques.

In [3], the authors proposed a syntactic approach based on variable hiding where variables only used on assignment expressions of the variables of interest, are replaced by their domains. However, this approach is restricted to a limited subset of the program variables.

In [33], Hsieh and Levitan proposed two abstraction algorithms based on semantic extraction of VHDL code from CFG [32]. The first is called, *key value*

*extraction* where variable domain is abstracted to constants or abstract data types. Unfortunately, such abstraction becomes complex with the increase of assignment dependency between variables as variables inherent key values from each other. The other type is a kind of predicate abstraction for conditional expression relating program variables. The problem with this abstraction is the large number of Boolean variables which might be generated and evaluated. For each assignment, the set of related Boolean variables must be evaluated, while in our proposed values abstraction only one non-deterministic variable will be used to choose between values which reduce significally the number of added variables.

Our proposed reductions are also related to works like localisation reduction [40]. However, our approach is an extension of these approaches because we analyse the dependency between the *values* of variables in addition to the dependency between variables, thus the dependency relation is more accurate.

## 3.4 Summary

In this chapter, we overviewed two static analysis techniques: Data flow analysis and abstraction, which inspired the reduction algorithms proposed in the next chapter. Applying forward reachability analysis on the CFG of a given program, we can build the path sequence proposed in Chapter 2, while applying backward analysis on the CFG, we can collect the reachable values for a specific variable, starting at a specific program point. Using the concept of abstraction, we can, hence, reduce the domain

of the variable according to a certain criteria.

# Chapter 4

# Model Reduction Techniques

Data abstraction was proposed as a way to reduce the size of the design prior to verification. The basic idea is to specify a mapping from concrete variables of the model to fewer abstract variables [12]. The mapping consists of a list of concrete variables to be eliminated, and a list of variables with abstract values domain. The mapping depends on the property to be verified, hence the abstraction is repeated each time a property is defined.

In this chapter, we will introduce two reduction algorithms, based on data abstraction. The first one is *Cone of influence* (COI) [40], which is a variable reduction algorithm where variables not affecting the property are removed. Cone of influence has been implemented successfully in many tools [37, 39]. Making use of the observation that not all the values domain of the remaining variables affect the property, we will introduce a values abstraction algorithm proposed first in

[27], where the values domain of the variables are partitioned into two categories depending on the property. Values affecting the property are kept while the others are abstracted.

## 4.1 Cone of Influence Reduction

The basic idea of Cone of Influence reduction is to construct a dependency graph of the program variables, rooted at the variables in the specification. The set of variables in the graph is called the $COI$ of the specification. The variables not included in the COI set cannot influence the validity of the specification and can therefore be removed from the model.

Let's consider the example in Figure 4.1. We want to verify a property about variable $a$. If we examine its DDG in Figure 4.2, we find that $a$ only depends on $c$ which depends on $f$. Other variables are not related to neither $a$ nor $c$ and $f$. Removing these variables will not affect the property under verification. We say that the *cone of influence* of $a$ includes only the variables $c$ and $f$.

**Definition 4.1.1** *Let $V$ is the set of program variables and $\Upsilon$ is a relation between the variables. The Cone of influence of a variable $v_i$ is $\Delta = \{v_1, ...v_i..., v_k\}$, $\Delta \subseteq V$ such that for any two variables $v_n \in V$ and $v_m \in V$, if $v_n \in \Delta \bigwedge (v_n, v_m) \in \Upsilon$, then $v_m \in \Delta$*

In COI reduction, the model is reduced by removing behaviours when going

```
Module coi_exp (c,d,a)

input c,d;
output a;
reg a,b,e,f;

initial

      begin
      a = 0;
      b = 0;
      e = 1;
      f = 1;
      end;

always
      begin
      if(c == f)
            a = 1;
      else
      if(d == e)
            b = 1;
      f = f+1;
      end
endmodule
```

Figure 4.1: COI example

from the concrete to the abstract system. Since no new behaviours are added by non-deterministic statements, as in the case of over approximation techniques, strong preservation of the properties is achieved. Suppose the properties over the system models are specified by the computational tree logic CTL.

**Theorem 4.1.1** *Let* f *be a CTL formula with atomic proposition in* $\Delta$*,and* P and $\hat{P}$ *be the concrete and COI abstract models respectively, then*

$$P \models f \leftrightarrow \hat{P} \models f$$

As can be inferred from the theorem above, a bisimulation relation exists between the concrete and abstract program. The cone of influence set for a certain

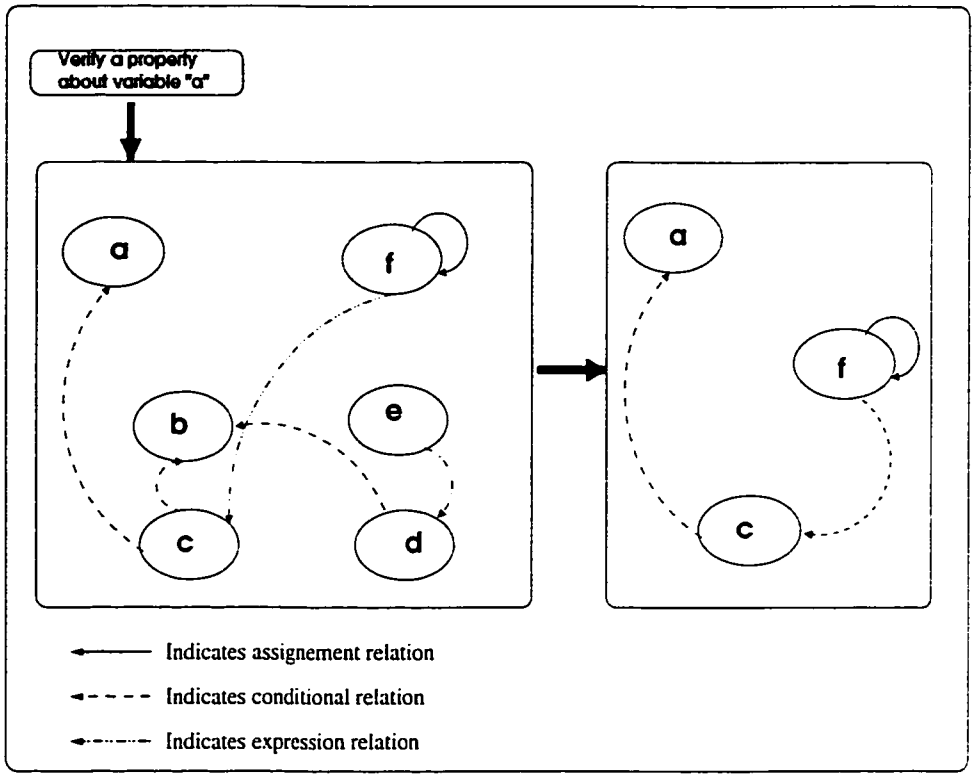


Figure 4.2: DDG of example in Figure 4.1

variable $v$ ($COI_v$) is built by fixpoint iteration using forward reachability analysis on the control flow graph; each iteration adds new variables (related to the ones added before) to the list. The iteration stops when there are no more variables to be added. After the COI set is created, only edges with variables included in the set are kept, other edges are removed and a new reduced control flow graph is created. If we apply the cone of influence algorithm for the Verilog program in Example 2.2.1, and the property to be verified is about the variable $pc$, then the cone of influence of this variable is $COI_{pc} = \{x, pc\}$. The reduced data dependency graph (DDG) and control flow graph (CFG) of this program are shown in figures 4.3, 4.4 respectively.

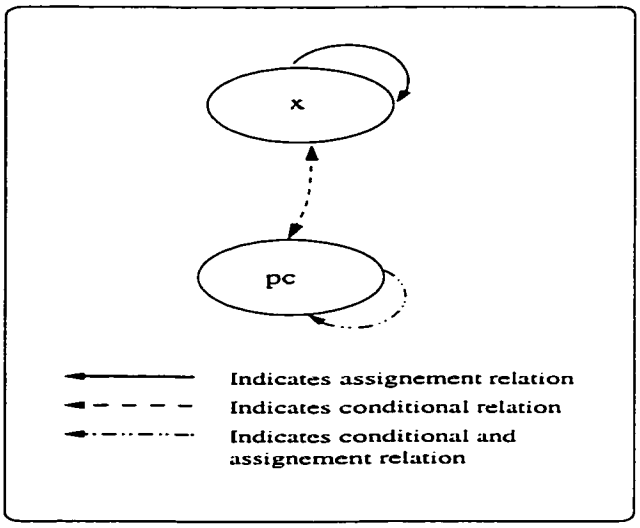


Figure 4.3: Reduced DDG of the program in Example 2.2.1

## 4.2 Values Abstraction

Only part of the value domain of the variables usually affects the verification process. For example, if we have a property to check whether a variable will reach a certain value, such as: $AF(A > 10 \Rightarrow B = 1)$. This property states that eventually on all paths, $B$ will be set if $A$ is greater than 10. We can understand from this statement that the model checker, in order to verify this property, will look for states where the values of $A$ are greater than 10 and check that $B$ will have the value one. States with $A \leq 10$ are, however, of no interest. If they are abstracted, the property will not be affected and the state space to explore will be smaller, thus avoiding a possible state space explosion problem.

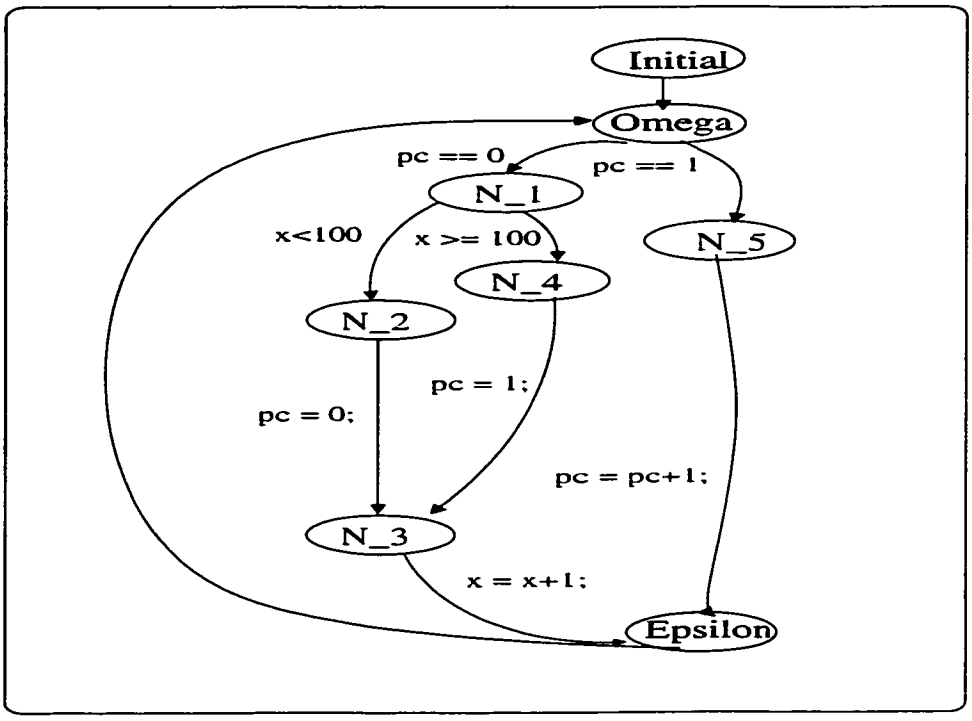


Figure 4.4: Reduced CFG of the program in Example 2.2.1

In the following, we will introduce the definitions of the key notions used for the values abstraction algorithm.

**Definition 4.2.1** *A* test node *is a node with one of its outgoing edges is an assignment edge and is labelled by a variable under verification, i.e, included in the property logical expressions. If this latter assignment satisfies the truth of the logical expressions then the test node is a* key node $\phi$.

**Definition 4.2.2** *An* active path $\tau$ *is a CFG path* $\pi$ *which begins by node* $\omega$ *and ends with a key node. As more than one path can ends with the same key node* $\phi$, *the set of active paths having* $\phi$ *as the end node is* $\Pi_\phi = \{\tau_1, ..., \tau_n\}$.

**Definition 4.2.3** *Let* $\pi$ *be a finite path in the CFG and* $b \in dom(v)$ *a value, we say*

*that a value b is active iff $RC_\pi(v)$ is true or $ST_\pi(v) = b$. Similarly, we say that a value b is deactive iff $RC_\pi(v)$ is false or $ST_\pi(v) \neq b$. $RC_\tau(v)$ and $ST_\tau(v)$ are the Reachability Condition and the State Transformation of the path $\tau$, respectively.*

*Intuitively, the active domain,* ACTIVE $(v)$*, will contain the values affecting directly the property and will remain unabstracted, while the deactive domain,* DEACTIVE $(v)$*, can be abstracted by using a representative value and therefore will contain one single value [49].*

- *if* ACTIVE $(v) = \emptyset$*, then $v$ is a variable not affecting the target statement and* DEACTIVE $(v) = \{d\}$*, where $d \in dom(v)$ and $d$ is the abstract value.*
- *if* ACTIVE $(v) \neq \emptyset$ *and* DEACTIVE $(v) \neq \emptyset$*, then $v$ has values affecting the target statement, and these values will not be abstracted. They will be stored in* ACTIVE $(v)$*, while the rest of the values will be abstracted.*
- *if* DEACTIVE $(v) = \emptyset$*, then the whole domain of $v$ will affect the target assignment*

According to the the new active and deactive domains, we can describe the abstract program as follow:

**Definition 4.2.4** *The abstract program of $P$, written $\widehat{P} = (\widehat{V}, \widehat{I}, \widehat{A}, \widehat{S})$, is defined as:*

- $\widehat{V} = V$ *and for each* $v \in \widehat{V}$*,* $dom(v) =$ACTIVE $(v) \cup$ DEACTIVE $(v)$.

- $\widehat{I}(\widehat{V})$ *is equal to* $I(V)$[DEACTIVE $(v)/d$] *for each* $v \in V$ *and* $d \notin$ACTIVE $(v)$

- $\widehat{A}(\widehat{V})$ *is equal to* $A(V)$ *with its set of statements* $\widehat{S}$ *next.*

- $\widehat{S}$ *is similar to* $S$*, where we only replace assignment with non-deterministically assignment* $v =$ACTIVE $(v) \cup$ DEACTIVE $(v)$.

The mapping between the initial and abstract domains can be described as follows [49]:

**Definition 4.2.5** *Let* $B \subseteq \sigma \times \widehat{\sigma}$*, where* $\sigma$ *and* $\widehat{\sigma}$ *are the concrete and abstract configurations respectively:*

$$\forall\, i : ((d_1, \ldots, d_n), (\widehat{d_1}, \ldots, \widehat{d_n})) \in B$$
$$\Leftrightarrow ((\widehat{d_i} = d_i) \wedge (\widehat{d_i} \in ACTIVE(v_i))) \vee ((\widehat{d_i} \neq d_i) \wedge (\widehat{d_i} \in DEACTIVE(v_i)))$$

Before using this abstraction, it is important to check its conservativeness, .i.e. a weak preservation exists between the concrete and abstract models.

**Theorem 4.2.1** *[49]: B is a simulation relation between the models* $\mathcal{K}(P)$ *and* $\mathcal{K}(\widehat{P})$*, namely* $\mathcal{K}(P) \preceq \mathcal{K}(\widehat{P})$.

**Theorem 4.2.2** *[11] Suppose* $M \preceq M'$*, where* $M$ *and* $M'$ *are two models. Then for every ACTL formula* $\varphi$*,* $M' \models \varphi \Rightarrow M \models \varphi$

**Example 4.2.1** *We consider Example 2.2.1 and assume that the property to verify is* $AF(out == 01)$*. According to this property, the active value for* out *is* 1*. By*

*inspecting the CFG in Figure 2.5, we identify node* $N_8$ *as a* key node *and The* key path *is* $P_5 = \omega \rightarrow N_7 \rightarrow N_8 \rightarrow \varepsilon$, *where* $RC_{P_5} = pc = 2 \wedge y = 100$ *and* $ST_{P_5} = out = in$. *The next step is to abstract the domain of the variables* pc, y *and* in, *which are the variables affecting the property on the path* $P_5$.

*From* $RC_{P_5}$, *we know that the values 2 and 100 are active value of the variable pc and y, respectively. By reachability analysis on the path sequence (shown in Figure 2.10) and starting from* $P_5$, *we find that* in *will not be abstracted as it is not assigned anywhere. The values domain of* pc *are* $0, 1, 2$, *with* ACTIVE $(pc) = \{2\}$, DEACTIVE $(pc) = \{1\}$. *The abstract values domains of the variable* y *and its dependency variable* $x$ *are be partitioned respectively as* ACTIVE $(y) = \{100\}$, DEACTIVE $(y) = \{0\}$,*and* ACTIVE $(x) = \{100\}$ *and* DEACTIVE $(x) = \{0\}$. *For the other variables, if any, their domains haven't changed and considered as active values. The abstract CFG is shown in Figure 4.5.*[1]

## 4.3 Summary

In this chapter, we presented two syntactic reduction algorithms which can be applied automatically to Verilog designs, in order to reduce them prior to verification. The first algorithm is based on removing redundant variables according to the

[1] *As this abstraction is mainly applied on synthesizable hardware description languages, the non-deterministic assignments are not permitted. In order to deal with this problem, uninitialised inputs are inserted such that for every non-deterministic assignment, one of these inputs determines which value to choose next. In Figure 4.5,* $j1$, $j2$ *and* $j3$ *are the inserted new inputs. For example, when* $j2$ *is equal to 0,* $y$ *is equal to 100 else* $y$ *is equal to 99.*

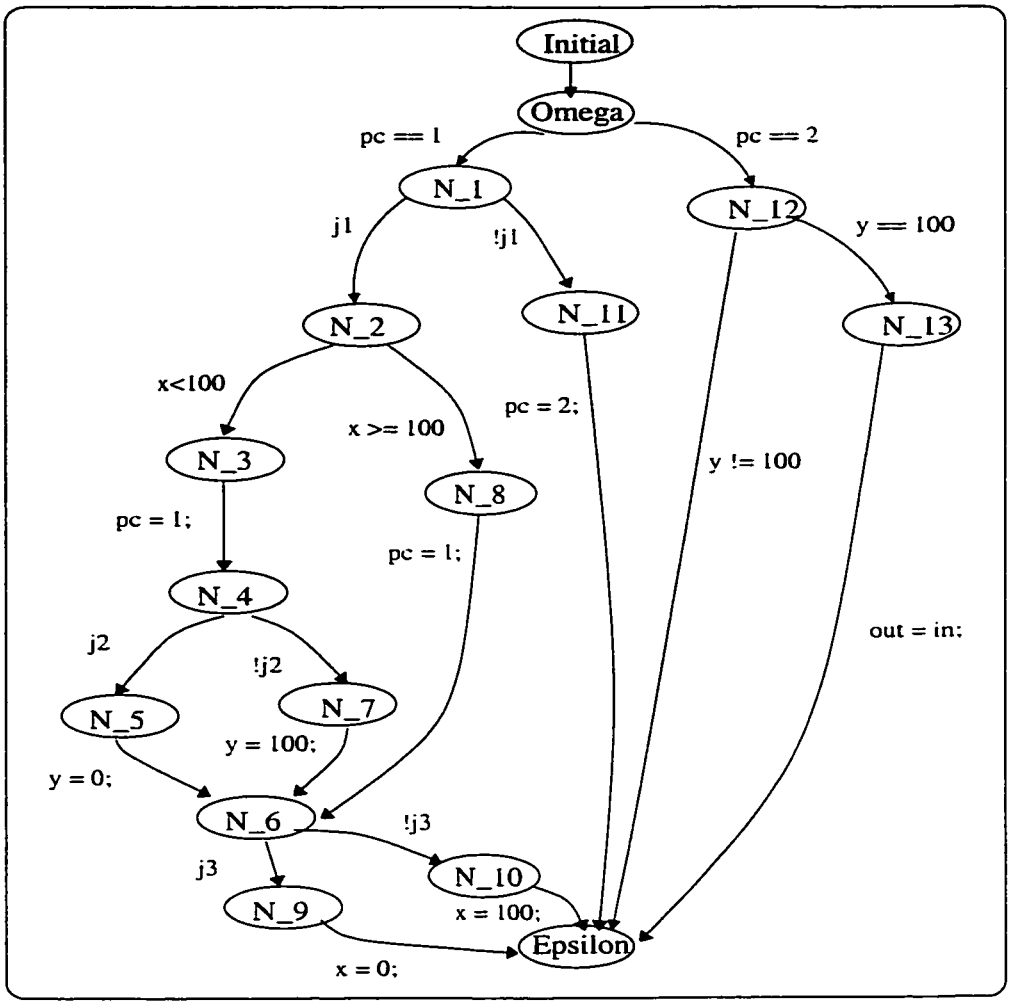


Figure 4.5: Abstract CFG for the Verilog program in Example 2.2.1

property variables, while in the other one, the value domains of the variables are abstracted according to the truth of the property expressions.

We have seen through in this chapter, how the approaches succeeded to reduce significantly the number of program variables, as well as the domain of the variables. The output model size is, hence, smaller than the original one, which helps alleviate the state explosion problem. For example, the variable $y$, of the program in Example 2.2.1 has its domain reduced from more than 100 values to only 2 values.

In the next chapter, we will present the implementation of these reduction algorithms in a tool called **SynAbs**. We will describe its architecture as well, and evaluate its performance.

# Chapter 5

# Implementation and Experimental Results

In the previous chapter, we presented a model reduction approach based on cone of influence reduction and values abstraction. The algorithm works in reducing the domain of the program variables in order to obtain a smaller design which can be accepted by model checker tools. The initial idea for implementing this algorithm was based on using the SAT solver SATO [62] [1] as a way to find the truth value of the variables leading to the creation of abstract domain, hence abstract the program [50]. This proposal has the following disadvantages:

- A translation is required from the reachability formulas to the SATO format as propositional formulas in conjunctive normal forms (CNF), then an analyser

[1] In the satisfiability problem, the question is: given the expression, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression true?

is used to apply the result out of the SATO solver to the original program in order to generate the reduced program.

- Due to the non-halting property of Verilog programs, it is not accurate to determine the values domain of variables by the SATO solver.

The algorithm implementation proposed by Peng et *al.*, in [49] is hence, not automatic and can be applied only for small programs with no feedback paths. In addition, he did not give a solution for the evaluation of ST formulas. Our goal was to develop the algorithm such that it can be fully automatic and deals efficiently with non-halting programs.

By representing the program as an abstract state transition system , i.e., path sequence, we were able to automate the algorithm by avoiding interacting with another tool, and support a synthesiable subset of Verilog. In this chapter, we will introduce **SynAbs** (Syntactic Abstraction) tool, which implements the abstraction algorithms described in Chapter 4.

## 5.1 SynAbs Tool Specification

**SynAbs** (see Figure 5.1) accepts at the input two text files: a Verilog program that includes the design to be verified and an ACTL specification . The abstraction algorithms analyse the input program, based on the property provided by the user, a reduced Verilog code is output. The process is fully automated, i.e., no interaction

is required by the user. The reduced Verilog program file, can be fed to a model checker such as SMV [43] and VIS [52] for verification.

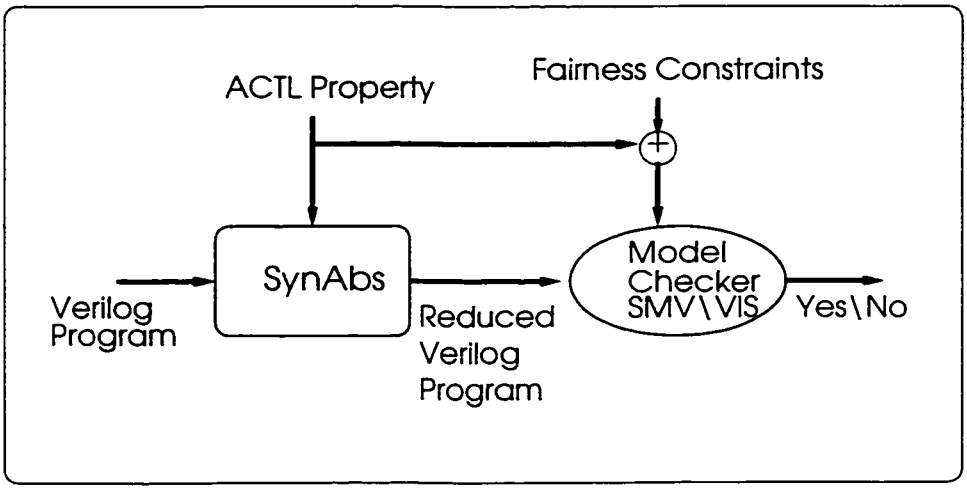


Figure 5.1: SynAbs tool usage

The full syntax of the Verilog subset supported by **SynAbs** can be found in Appendix A.1a. The syntax of the ACTL supported by **SynAbs** is given in Appendix A.1b. Properties are written the same way they are written in the SMV language.

Fairness constraints are usually added to the property when verifying the abstract design, in order to avoid false negatives (when an error in the abstract design does not exist in the original one).

In order to facilitate the analysis of the programs under verification, several options were included in **SynAbs**. The user can choose whether to turn off one of the abstractions. The internal representations (CFG, DFG, PS) of the programs can be displayed by a graph visuallization tool *Graphviz* [22]. Graphical representation

always helps understanding the behaviour of the designs better.

## 5.2 SynAbs Tool Structure

SynAbs was designed as different modules interacting together. The advantage of this modularity is the simplicity of updating its architecture. e.g., by modifying the Verilog subset or adding more syntactic abstraction techniques like those discussed in Chapter 3, the related work section.

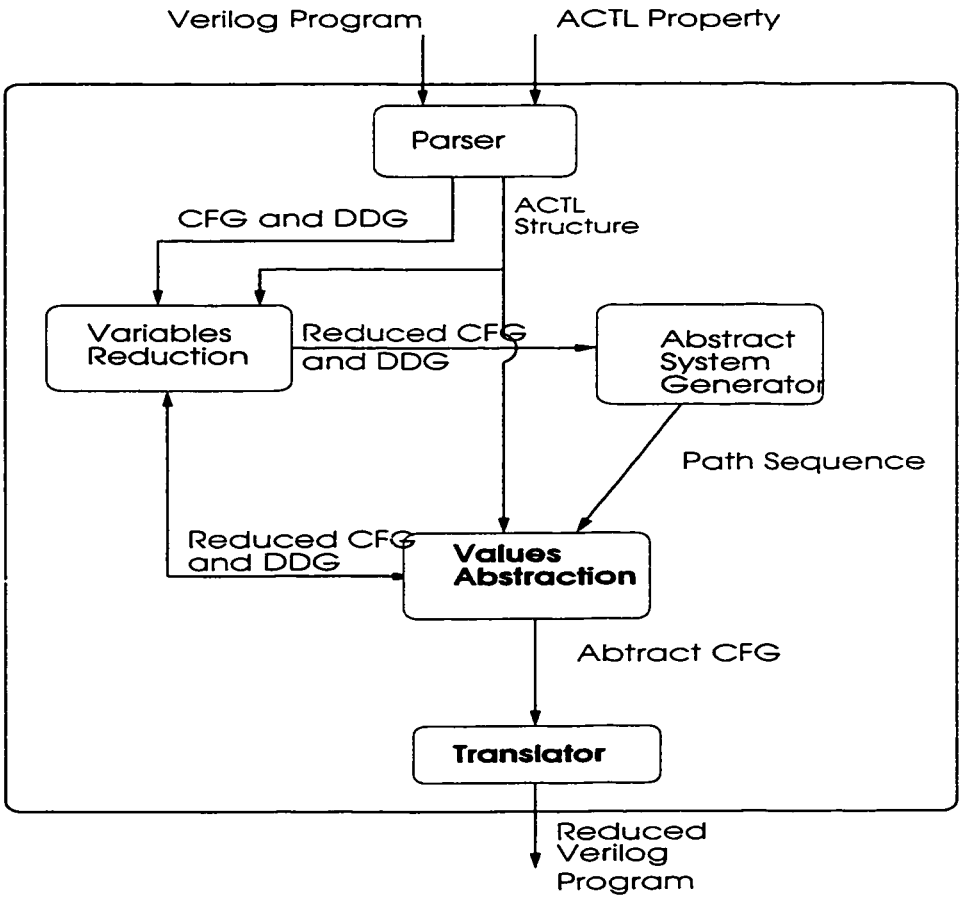


Figure 5.2: SynAbs Tool Architecture

The structure of **SynAbs**, as shown in Figure 5.2, is composed of the following modules:

**Parser**. Upon reading the Verilog and the ACTL files, the parser checks the syntactic correctness of the codes, builds the parse tree representing the internal format of these files. The parse tree of the Verilog file is then translated into linked structures representing the control flow and the data dependency graphs (CFG and DDG) of the program as well as the ACTL data structure.

**Variable Reduction**. Using the set of variables in the property, the Variable Reduction module removes irrelevant variables to the property and generates a COI reduced CFG and DDG, which are input to the Path Sequence Generator.

**Path Sequence Generator**. It performs reachability analysis on the CFG to analyse the semantics and generates the corresponding PS. This PS will be provided in case the Values Abstraction module needs to gather the domain of a certain variable.

**Values Abstraction**. This module analyses the CFG and the PS to build the sets of active and deactive domain for each variable according to data extracted from the ACTL property's expression. Once the sets are filled, the Values Abstraction module creates the abstract CFG. After the abstraction, some redundant variables will be removed. This module also generates new un-initialised inputs which will be used to implement the non-deterministic choice.

**Verilog Generator** At the back end of the tool, a translator generates the

reduced synthesiable Verilog which can be fed to model checker tools for verification.

## 5.3 SynAbs Tool Implementation

SynAbs was implemented in ANSI C [36]. The parser used in the tool is a modification of the one used by the Verilog to blif-mv language translator $VL2MV$ [8] used in the $VIS$ tool. Major modifications done was to generate the CFG and DDG as well as to support the parsing of the ACTL syntax. Some libraries form the $VL2MV$ tool were also used throughout the **SynAbs** implementation, including libraries to implement the lists and the hash tables. The software tool [2] is open source and composed of many files interacting together through header files and extern variables. As an option, the internal representation of the Verilog program can be displayed in the DOT format which is accepted by the graphical visualisation tool Graphviz, from ATT labs, to display the graph.

## 5.4 Performance Evaluation

In order to evaluate the performance of the tool, we considered some examples using the SMV model checker before and after being abstracted by **SynAbs**. Preliminary results were satisfactory.

The program used here for performance evaluation is an extended version of

[2]website: http://hvg.ece.concordia.ca/Tools/SynAbs

the program in example 2.2.1, with 32-bits registers. The program is formed of one module, which has two inputs *reset* and *in*, and two outputs *out1 and out2*. The values of the outputs are changed depending on conditions on the inputs and the internal variables. The advantages of using this example is that it includes many of the important Verilog constructs supported by our tool. Following are three example properties to be verified:

- **Prop 1**: $AG(\neg reset \rightarrow AF(out1 == 01))$.
- **Prop 2**: $AG(x == 1001 \rightarrow AF(out1 == 01))$.
- **Prop 3**: $AG(x == 1001 \rightarrow AF(out2 == 999))$.

A comparison between the verification results with SMV, before and after **SynAbs** reduction, is shown in Table 5.1. The original and abstracted Verilog codes can be found in Appendix A.2.

Table 5.1: Verification Results of Sample Examples in SMV

| | | **Model Checking** | | | | | |
|---|---|---|---|---|---|---|---|
| | | With SMV reduction | | | With **SynAbs** reduction | | |
| Example | Property | Status | BDDs | Time | Status | BDD | Time |
| E32 | (1) | Verified | 164712 | 11.87 sec | Verified | 2870 | 0.17 sec |
| | (2) | Verified | 638328 | 281.98 sec | Verified | 2870 | 0.17 sec |
| | (3) | Verified | 542553 | 175.59 sec | Verified | 1191 | 0.17 sec |

From the results in table 5.4, we can see how much saving in verification time and minimization of the reachable BDD nodes was achieved by reducing the

Verilog program prior to Verification. Main factor of this achievement was due to the reduction of large vector size variables such as $x$ and $y$, from 32 bit wide to only 10 bit or less.

# Chapter 6

# Conclusion and Future work

The state space explosion is the major problem in applying model checking to real life hardware designs. In this thesis, we proposed a model reduction tool called **SynAbs** for model checking of Verilog RTL designs. The main advantage of **SynAbs** is the automation, no interaction is required from the user.

The algorithms implemented in **SynAbs** are based on syntactic analysis of programs,where no explicit representation of the transition system is created and the reduced output model is a program text. Hence, other BBD based abstractions can be applied later, in order to enhance the abstraction phase.

The original goal of this thesis was to implement the reduction algorithms proposed in [49]. However, because of the drawbacks in using a SAT Solver, we proposed the use of an abstract state transition system (path sequence), on which reachability analysis can be applied efficiently. The main idea of the *path sequence* is

to represent all the states in the program by the control flow graph paths generating these states. This representation is conservative and is very small compared to explicit and Boolean state representations.

Our contribution can be summarised in two main points.

- The extension of the values abstraction algorithm [49] by increasing its precision and automating it. This was done by introducing a path dependency graph called path sequence.

- The construction of a model abstraction tool, implementing the abstraction algorithms.

SynAbs is a prototype tool and it has its own limitations.

- Some constructs like the *for statement* and *arrays* are not supported for the moment. These can be supported by modifying the parser of the tool with loop unrolling to simplify the *for* statements and variable renaming for array manipulation.

- The tool does not support multi-modules designs. We are planning to add a *flattening block* to eliminate such constraint.

- A major future direction is to modify the values abstraction algorithms in order to abstract the domain of the variables involved in logical comparisons. e.g., if the expression $v_1 > v_2$ is found, the tool skips the abstraction of both variables.

- The reduction algorithms can also be applied, in a restricted manner, to programs with infinite domains variables, like those used in C program. Consider for example the infinite domain variable $x$ used in a while loop such as "$while(x > 0)do$". This will lead to the nontermination problem and the algorithm will fail to abstract the variable $x$ values domain.

In addition to overcoming the limitations mentioned above, other future directions include applying the abstraction for real designs such as commercial ATM switches, investigate the usefulness of supporting new Hardware Description languages such as System Verilog [1]. One interesting point here is that System Verilog has the advantage of using *enumeration types*; which might reduce the cost of introducing uninitialised inputs for non-deterministic assignments in the abstraction.

Finally, in order to overcome the false negative problem in the over approximation technique, we have used fair constraints in conjunction with the property. Future enhancement in this direction is to investigate the use of counter-examples in the refinement procedure.

# Appendix A

# Syntax

The following is the Verilog and ACTL subsets suported by **SynAbs**. The grammar is defined in Backus-Normal Form (NBF). Capitalized words represemt terminal tokens.Other words represent non-terminals. " ::=" is read as "is defined as". "|" is read as "or"

## A.1 *Verilog Subset*

```
program
       ::= module

module
      ::= MODULE id port_list ; module_items ENDMODULE

port_list
         ::= port | port_list , port

port
    ::= IDENTIFIER
```

```
module_items
            ::=  module_items module_item

module_item
           ::= input_declaration
           |output_declaration
           |inout_declaration
           |net_declaration
           |reg_declaration
           |continuous_assign
           |initial_statement
           |always_statement


input_declaration
                 ::= INPUT range_opt variable_list ';'

output_declaration
                  ::= OUTPUT range_opt variable_list ';'

inout_declaration
                 ::= INOUT range_opt variable_list ';'

reg_declaration
                ::= REG range_opt register_variable_list ';'
        ;

register_variable_list
                      ::= register_variable
                      | register_variable_list ',' register_variable
        ;

register_variable
                 ::= IDENTIFIER
        ;

variable_list
             ::=  IDENTIFIER
             | variable_list ',' IDENTIFIER
        ;

range_opt
```

```
        ::=
        | range
        ;

range
     ::= '[' expression ':' expression ']'
        ;

continuous_assign
                ::= ASSIGN assignment_list ';'
        ;

initial_statement
                ::= INITIAL statement
        ;

always_statement
               ::= ALWAYS statement
        ;

statement
         ::= ';'
         | assignment ';'
         | IF '(' expression ')' statement
         | IF '(' expression ')' statement ELSE statement
         | CASE '(' expression ')' case_item_list ENDCASE
         | seq_block
         | WAIT '(' expression ')' statement
         | WHILE '(' expression ')' statement
        ;

case_item_list
             :: case_item
             | case_item_list case_item
             ;

case_item
         ::= expression_list ':' statement
         | DEFAULT ':' statement
         | DEFAULT statement
         ;
```

```
seq_block
        ::= BEGIN statement_list END
        ;

statement_list
             ::=
             | statement_list statement
             ;

assignment_list
              ::= assignment
              | assignment_list ',' assignment
              ;

assignment
         ::= IDENTIFIER '=' expression
         |  IDENTIFIER NBASSIGN expression
         ;

expression_list
              ::= expression
              | expression_list ',' expression
              ;

expression
         ::= IDENTIFIER
         |INUMBER
         | FALSE
         | TRUE
         |UNARY_OPERATOR expression
         | '(' expression ')'
         | expression '+' expression
         | expression '-' expression
         | expression '*' expression
         | expression '/' expression
         |  expression '%' expression
         | expression LOGEQUALITY expression
         | expression LOGINEQUALITY expression
         | expression LOGAND expression
         | expression LOGOR expression
         | expression LOGXNOR expression
         | expression LEQ expression
```

```
          | expression GEQ expression
          | expression '<' expression
          | expression '>' expression
          | expression '&' expression
          | expression '|' expression
          | expression '^' expression
      ;
```

## A.2 *ACTL Subset*

```
specification
             ::=     PROP  expression
;

expression
          ::= IDENTIFIER
          |INUMBER
          | FALSE
          | TRUE
          |UNARY_OPERATOR expression
          | '(' expression ')'
          | expression '+' expression
          | expression '-' expression
          | expression '*' expression
          | expression '/' expression
          |  expression '%' expression
          | expression LOGEQUALITY expression
          | expression LOGINEQUALITY expression
          | expression LOGAND expression
          | expression LOGOR expression
          | expression LOGXNOR expression
          | expression LEQ expression
          | expression GEQ expression
          | expression '<' expression
          | expression '>' expression
          | expression '&' expression
          | expression '|' expression
          | expression '^' expression
          | AX expression
          | AF expression
```

```
    | AG expression
    | AA '[' expression UNTIL expression ']'
    | AA '[' expression RELEASES expression ']'
    | expression IMPLIES expression
    | expression IFF expression

;
```

# Appendix B

# Verilog Examples

## B.1 Example 1

### B.1.1 Original Verilog Code

```
module main(reset,in,out1,out2);

input reset,in;
output [0:32] out1,out2;
reg [0:32] x,y,pc,out1,out2;

initial
begin
pc <= 'd00;
x <= 'd00;
y <= 'd00;
out1 <= 'd00;
if(in) out2 = 'd00;
else out2 = 'd01;
end

always
begin
if(!reset)
begin
case (pc)
'd00:
begin
if(x<1000)
begin
```

```
pc = 'd00;
y = y+1;
end
else pc = 1;
x = x+1;
end
'd01:
begin
out2 = y;
pc = pc+1;
end
'd02 :
begin
if(y==1000) out1 = 'd01;
end
endcase
end
else
begin
out1 = 'd00;
out2 = 'd00;
end
end

endmodule
```

### B.1.2 Abstract Verilog Code for Properties 1 and 2

```
module main(reset,i1,i2,i3,out1);

input reset,i1,i2,i3;
output out1;
reg out1;
reg [0:9] x,y;
reg [0:1] pc;

initial
begin
pc <= 'd01;
x <= 999;
y <= 999;
```

```
out1 <= 'd00;
end

always
begin
if(!reset)
begin
case (pc)
'd01:
begin
if(i3)
begin
if(x<1000)
begin
pc = 0;
if(i1) y = 999;
else y =1000;
          //y = ND(999,1000);
end
else pc = 1;
if(i2) x = 999;
else x =1001;
        //x = ND(999,1001);
end
else
pc = 'd02;
end
'd02 :
begin
if(y==1000) out1 = 'd01;
end
endcase
end
else
begin
out1 = 'd00;
end
end

endmodule
```

### B.1.3 Abstract Verilog Code for Property 3

```
module main(reset,i1,i2,out2);

input reset,i1,i2;
output [0:9] out2;
reg [0:9] out2;
reg [0:9] x;
reg [0:1] pc;

initial
begin
pc <= 'd00;
x <= 999;
out2 <= 'd00;
end

always
begin
if(!reset)
begin
case (pc)
'd00:
begin
if(x<1000)
pc = 0;
else pc = 1;
if(i2) x = 999;
else x =1001;
//x = ND(999,1001);
end
'd01:
begin
if(i1)
        out2 = 999;
else out2 =1000;
pc = 1;
end

endcase
end
else
```

```
begin
out2 = 'd00;
end
end

endmodule
```

# Bibliography

[1] Accellera. *The SystemVerilog Language Reference Manual (LRM)*, 2002.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, New York, USA, June 1990.

[3] R. Bharadwaj and C. L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering: An International Journal*, 6(1), January 1999.

[4] J. Bhasker. *A Verilog HDL primer*. Star Galaxy Publishing, 1999.

[5] M. Bourahla and M. Benmohamed. Predicate abstraction and refinement for model checking VHDL state machines. In *Seventh International Workshop on Formal Methods for Industrial Critical Systems*, University of Malaga, Spain, July 2002.

[6] G. Brat, K. Havelund, S.J. Park, and W. Visser. Model checking programs. In *Proceedings of International Conference on Automated Software Engineering*, Grenoble, France, September 2000.

[7] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computers*, C-35:677–691, august 1986.

[8] S. Cheng. Compiling Verilog into automata. Master's thesis, Technical Report UCB/ERL M94/37, Electronics Research Lab, University of California, Berkeley, USA, May 1994.

[9] E. Clarke, M. Fujita, S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Conference on Correct Hardware Design and Verification Methods*, LNCS, pages 298–312, Germany, 1999.

[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counter-example guided abstraction refinement. In *Procceding of Computer Aided Verification'00*, volume 1855 of *LNCS*, pages 154–169, Chicago, IL, USA, July 2000.

[11] E. M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[12] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[13] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics, 10 Years Back, 10 Years Ahead*, LNCS, pages 138–156. Springer-Verlag, 2001.

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California. USA, 1977.

[15] J. Crow, S. Owre, J. Rushby, N.Shankar, and M. Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*. Boca Raton, Florida, USA, April 1995.

[16] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands. July 1996.

[17] D. Dams. abstraction in software model checking: Principles and practice. In *Model Checking of Software, In proceedings of the ninth International SPIN Workshop*, LNCS, Grenoble, France, April 2002. Springer.

[18] S. Das and D.L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*. Springer-Verlag, November 2002.

[19] S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *Proceedings of Computer Aided Verification*, pages 160–171, Italy, 1999.

[20] D. Dempster and M. Stuart. *Verification methodology manual, techniques for verifying HDL designs.* TransEDA, 2000.

[21] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, H. Zheng, and W Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software Engineering*, pages 177–187, Toronto, Ontario, Canada, May 2001.

[22] S.North et al. E.Gansner. *Graphviz.* www.research.att.com.

[23] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319-349, 1987.

[24] N. Francez. *Program Verification.* Addison-Wesley, 1992.

[25] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[26] R. Gupta, M.L. Soffa, and J. Howard. Hybrid slicing: integrating dynamic information with static analysis. *ACM Transactions on Software Engineering and Methodology*, 6(4):370–397, 1997.

[27] Y. Mokhtari H. Peng and S.Tahar. Syntactic model reduction. Technical report, Concordia University, Department of Electrical and Computer Engineering, Montreal, Quebec, Canada, April 2001.

[28] C. Hankin. Program analysis tools. *Iternational Journal on Software Tools for Technology Transfer*, 2(1), 1998.

[29] T. A. Henzinger. New directions in computer-aided verification. *ACM SIGSOFT Software Engineering Notes*, pages 56–57, 2000.

[30] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[31] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 23, Atlanta, Georgia, USA, June 1988.

[32] Y. Hsieh and S. Levitan. Control/data-flow analysis for vhdl semantic extraction. In *Proceedings of the 4th Asia-Pacific Conference on Hardware Description Languages*, pages 68–75, Taiwan, 1997.

[33] Y. Hsieh and S. Levitan. Model abstraction for formal verification. In *Design, Automation and Test in Europe Conference*, Paris, France, 1998.

[34] M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura. Program slicing on VHDL descriptions and its applications. In *Proceedings of Asian Pacific Conference on Hardware Description Languages*, pages 132–139, India, 1996.

[35] C. Kern and M. R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.

[36] B. W. Kernighan and D. M. Ritchie. *The C programming language.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1988.

[37] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131. Univ. of California, Berkeley, USA, 1992.

[38] J. Krinke. Static slicing of threaded programs. In *ACM SIGPLAN/SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pages 35–42. ACM Press, 1998.

[39] R. P. Kurshan. Formal verification in a commercial setting. In *Design Automation Conference*, pages 258–262, California, USA, 1997.

[40] R.P. Kurshan. *Computer aided verification of coordinating processes: the automata theoritic approach.* Princeton University press, 1994.

[41] J. L. Lions. Ariane 5 flight 501 failure report. http://java.sun.com/ people/jag/ Ariane5.html, 1996.

[42] J.C. Fermandez M. Bozga and L. Ghirvu. State space reduction on live variables analysis. In *Static analysis Symposium*, Venezia, Italy, September 1999.

[43] K. L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[44] M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *33th ACM Symposium on Theory of Computing*, pages 647–656, Hersonissos, Creete, Greece, July 2001. ACM SIGACT, ACM Press.

[45] R. Kurshan N. Amla and K. Namjoshi. Autoabs: Syntax-directed program abtraction. In *Submitted*, Bell labs, NJ, USA, 2002.

[46] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *Proceedings of Computer-aided Verification'00*, volume 1855 of *LNCS*, pages 433–449, Chicago, IL, USA, July "2000". Springer Verlag.

[47] A. Pardo. *Automatic Abstraction Techniques for Formal Verification of Digital Systems.* PhD thesis, University of Colorado at Boulder, Dept. of Computer Science, August 1997.

[48] D. Peled. Ten years of partial order reduction. In *Proceedings of 10th International Conference on Computer-Aided Verification*, volume 1427 of *LNCS*. Vancouver, BC, Canada, 1998. Springer-Verlag.

[49] H. Peng. *Improving Compositional Verification Environment Synthesis and Syntactic Model Reduction.* PhD thesis, Concordia University, Montreal, Quebec, Canada, 2002.

[50] H. Peng, Y. Mokhtari, and S. Tahar. Model reduction based on value dependency. In *Proceeding of IEEE International ASIC/SOC Conference.* Washigton, DC, USA, September 2001.

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

[52] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification CAV*, New Brunswick, NJ, USA, 1996. Springer Verlag.

[53] D. Sciuto, L. Baresi, and C. Bolchini. Software methodologies for VHDL code static analysis based on flow graphs. In *Proceedings of the conference with EURO-VHDL'96*, Geneva, Switzerland, 1996.

[54] IEEE standard 1076-1993. IEEE standard description language based on the VHDL hardware description language, 1993.

[55] IEEE standard 1364-1995. IEEE standard description language based on the Verilog hardware description language, 1995.

[56] U. Stern. *Algorithmic techniques in verification by explicit state enumeration.* PhD thesis, Technical University of Munich, Munich, Germany, 1997.

[57] T. Kropf. *Introduction to Formal Hardware Verification.* Springer, 1999.

[58] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[59] N. D. Jones und F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. *Handbook of Logic in Computer Science*, 1995.

[60] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 107–119, Toronto, Ontario, Canada, 1991. ACM Press.

[61] K. Yorav. *Exploiting syntactic structure for automatic verification.* PhD thesis, IIT, Haifa, June 2000.

[62] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of International Conference on Automated Deduction*, Australia, 1997. Springer.