Abstract Property Verifier based on Multiway Decision Graphs

Kamran Hussain

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 (Electrical & Computer Engineering)

at

Concordia University

Montréal, Québec, Canada

September 2007

Your file Votre référence
ISBN: 978-0-494-40883-4
Our file Notre référence
ISBN: 978-0-494-40883-4

ABSTRACT

Abstract Property Verifier based on Multiway Decision Graphs

Kamran Hussain

Symbolic model-checking tools encounter state-explosion problem when verifying designs with large data paths. Multiway Decision Graph (MDG) model-checker uses abstract data representation and applies abstract operations to address the state explosion problem. The MDG verification tool, also known as Abstract Verifier, takes as input the specification (written as properties) and the description of the design, and then proves or disproves if the design satisfies these properties. The original specification language of the Abstract Verifier was called L_{mdg} that provides temporal operators and abstract data types to formalize properties. Meanwhile, the Property Specification Language (PSL) has changed the verification world by introducing very rich temporal operators but without abstract data types. In this thesis, we propose a new specification language called Abstract Property Language (APL), for the MDG model-checker. This language replaces the L_{mdg} specification language by introducing new operators borrowed from PSL to improve its expressiveness. We provide the formal definition of this language in BackusNaur form (BNF) and provide its formal semantics based on the computational model of the Abstract Verifier. APL is associated with a front-end translator that accepts APL specifications and builds verification-ready models to be handled directly inside the MDG model-checker. Finally, we have validated our APL language and the translator tool on the verification of several test benches including Look-Aside Interface (LA-1) design.

To My Father - Tasadduq Hussain

ACKNOWLEDGEMENTS

First of all, I would like to thank the almighty God for giving me all the blessings throughout my life and for the opportunity to make this thesis possible.

It gives me great pleasure to thank all those who have helped me in my research work. The first person I would like to sincerely thank is my supervisor, Dr. Otmane Ait-Mohamed. Without his guidance, his expert advice, his support and continual encouragements, this thesis would not have been possible. I express my heartfelt gratitude to him. I sincerely thank Dr. Sofiene Tahar, who along with my supervisor have created one of the best research groups in this field. To all my fellow researchers in Hardware Verification Group (HVG) at Concordia University, I thank you for your friendship, your thoughtful discussions and productive feedbacks. Most importantly, I thank you all, the entire HVG family, for standing next to me when my father passed away. Without your encouragements and support, I would not have been able to continue my research work.

Last but not least, I thank my family and relatives, here and overseas, for their constant moral support and their prayers. Your support was invaluable in completing this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ABV	Assertion-Based Verification
AP	Atomic Proposition
APL	Abstract Property Language
ASM	Abstract State Machine
BDD	Binary Decision Diagram
BNF	Backus-Naur Form
CFG	Context-Free Grammar
CTL	Computation Tree Logic
DAG	Directed Acyclic Graph
DF	Directed Formula
DFA	Deterministic Finite Automata
FL	Foundation Language
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
LA-1	Look-Aside Interface
LTL	Linear Temporal Logic
MDG	Multiway Decision Graph
NFA	Non-deterministic Finite Automata
OBE	Optional Branching Extension
PSL	Property Specification Language
RE	Regular Expression
RTL	Register Transfer Level
SERE	Sequence Extended Regular Expression
SV	SystemVerilog
SVA	System Verilog Assertion
YACC	Yet another compiler-compiler

Chapter 1

Introduction

A major challenge in digital system design is ensuring correctness of the design at the earliest phase possible. Encountering design errors in manufactured products have lofty economic consequences. The impact is more severe in digital designs of critical applications. Given the number of devices and number of gates per IC chip growing radically, verification engineers must find a way to provide adequate verification coverage without drastic increase in verification time and cost.

Simulation-based methods are standard practice in the industrial community for hardware verification. However, they cannot offer complete coverage because the number of test cases grows significantly with the complexity of the design. Recently, formal verification methods have become an important complement because of their ability to find errors early in the design cycles through exhaustive exploration of all possible behavior.

In formal verification, the goal is to mathematically establish that the *system under verification* satisfies its specification. There are three main aspects when using formal verification techniques: the implementation of the system, its corresponding specification (or reference model) and the relationship between the implementation and the specification (or reference model). Among the three, it is the relationship that needs to be ascertained. The system representation can be at any level of

abstraction. Its specification, refers to the properties, can be given by behavioral descriptions, abstract structural description, temporal logic formulae, and so forth. Formal verification methods are usually categorized into three major techniques: theorem proving, equivalence checking and model-checking.

Theorem proving is generally an interactive approach where the implementation and the specification are stated in formal logic. The correctness is obtained by mathematically proving their relationship, equivalence or implication. The logic is characterized by a *proof system* that defines a set of axioms and a set of inference rules. Inference rules are applied until the desired theorem is proven. Theorem proving requires expertise and significant efforts on the part of the user in developing specifications of each component and in guiding the theorem prover through a large set of lemmas. As a result, this technique has limited practice in the industry, and it is mainly used for verifying critical parts of systems. The most popular theorem proving tools are Boyer-Moore Theorem Prover Nqthm [32], the Cambridge HOL system [27] and PVS [33].

Equivalence checking is a method to prove that two design representations of the same system are functionally equivalent. The two representations are usually at two different levels of abstraction. One common scenario of equivalence checking can be comparing a circuit's gate-netlist description with its RTL description. It is usually divided into two classes: *Combinational* equivalence checking and *Sequential* equivalence checking. In combinational equivalence checking, the circuits to be compared are converted into canonical representations of boolean functions, usually BDDs [14] which are then structurally compared to conclude the relationship. For example, MDG [1], an academic tool, and Synopsys Formality [3], a commercial tool, offer combination equivalence checking. On the other hand, in sequential equivalence checking, the two designs are represented using state-encoding. The equivalence is then proven by building the product finite state machine and checking whether the

output is invariant for any initial states of the product machine. It can verify equivalence between RTL and behavioral models because it only considers the behaviors of the models. However, due to state-space limitations, it cannot check large designs. MDG and VIS [4] are examples of tools that can perform sequential equivalence checking.

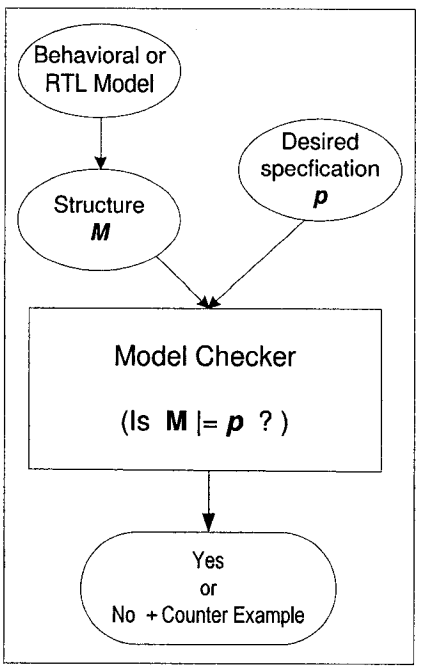


Figure 1.1: Model-checking method

Model-checking techniques are attractive verification method because of their high automation. Like equivalence checking, model-checking is historically a decision graph based method. It uses state space algorithms on finite-state models to check if the desired specification is satisfied. Specifications are expressed in a propositional temporal logic. An algorithm is employed to automatically explore the state-space and ascertain if the specification is verified by the state-transition graph. A top view of this method is depicted in Figure 1.1. Depending on *success* or *failure*, a model-checking tool gives the answer *yes* or *no* respectively. When the verification

fails, a counter example is provided that helps the user in tracing the source of the error. This counter example is also known as the *failure trace.*

Model-checking algorithms have been explored since early eighties and significant results have been published [26]. The introduction of Bryant's Binary Decision Diagrams (BDD's) [14] piloted a breakthrough in the size of a transition system that can be verified. Since then, a number of researchers have explored BDD-based symbolic technique and have published results [28] [30] [31]. The drawback of this method is that they usually suffer from the state-explosion problem when verifying designs with large data paths.

A new class of decision graph, called Multiway Decision Graphs (MDGs), was proposed as a solution to the state-space explosion problem by Cerny et al. in 1997 [15]. In MDG based model-checking approach, data signals are denoted by abstract variables, and data operators are represented by un-interpreted function symbols. As a result, a verification based on abstract-implicit-state-enumeration can be carried out independently of data path width, substantially lessening the state explosion problem. Table 1.1 shows the abstraction level of MDG compared to traditional methods. A model-checking methodology is typically comprised of three major parts: a specification language, a system modeling language and a set of algorithms to perform model-checking. In existing MDG methodology, these are L_{mdg} [37], MDG-HDL [38] and MDG model-checker [15] [35] respectively.

Table 1.1: Raising the abstraction level

Conventional method	Multiway Decision Graph
ROBDD [14]	MDG
Finite State Machine	Abstract State Machine
Implicit state enumeration [28]	Abstract state implicit enumeration of ASM
CTL based model-checking	Based on first-order abstract CTL*

In our work, we explore L_{mdg} property specification language that facilitates

the formal representation of desired specification in MDG based model-checking methodology. We also investigate how the written specifications are processed in MDG based model-checking.

1.1 Thesis Contribution

The existing MDG model-checking process requires two sets of tools: a back-end model-checker and a front-end comprised of L_{mdg} tools [37]. The front-end is a set of tools that provides a mechanism to process the desired specification written in terms of properties. The existing L_{mdg} specification language, used as specification language in MDG model-checking to write properties, is based on the first-order abstract CTL* class of temporal logic [36]. In order to create the verification-ready model, the property formulae are processed to construct additional circuitry that is added to the original model. The result is a composite model (Figure 1.2) in MDG-HDL. The composite model is fed to the back-end model-checker to formally verify if the specification holds.

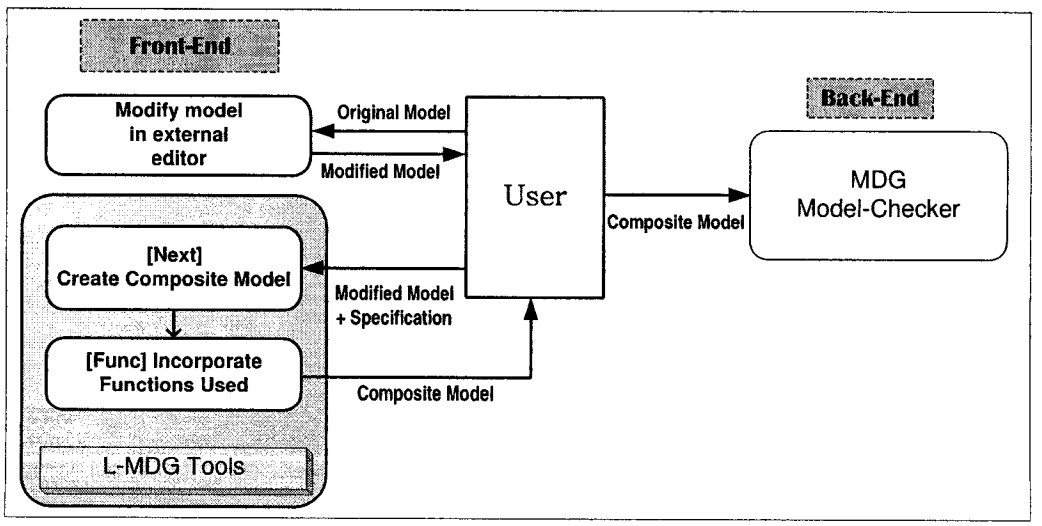


Figure 1.2: L_{mdg} Tools: front-end of the Abstract Verifier

After investigating the language description of L_{mdg} and the implementation of its tools, we have concluded that both can be improved without affecting the model-checking algorithms offered by the MDG model-checker [35]. The apparent issues are following:

- The front-end parsing process requires modifying the syntax of original model in an external editor. This is due to lexical restrictions in L_{mdg} language.
- An extra file specifying all the functions used in specification is needed. This is parsed by the *Func* tool (Figure 1.2).
- Concrete functions (cross-terms) used in specification cannot be handled [12].
- More than one tool is needed to build the composite model.
- The generated composite model needs to be manually fixed (edited) in order to comply with the rules of MDG-HDL [12].

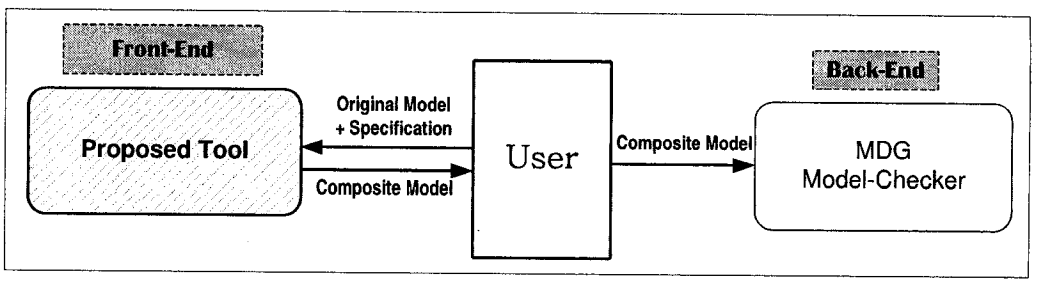


Figure 1.3: Proposed front-end of MDG methodology

Our goal is to develop a new front-end (Figure 1.3), where a single tool will accept the original model with its specifications written in an improved language and will efficiently create the verification-ready composite model. Thus, the proposed improvements are:

- Remove L_{mdg} language restrictions, thereby creating a new language.

- Support expressions of L_{mdg} but use appropriate standardized operators from PSL [6]. Also add PSL expressions to improve expressive-power.
- Develop a single tool (Figure 1.3) that can process specifications and generate respective composite models.
- Improve performance of the front-end specification processing.

1.2 Thesis Outline

This thesis is made up of six chapters. It is organized as follows:

- In Chapter 2, we give brief background information on topics that are required to comprehend materials presented in Chapters 3 and 4.
- In Chapter 3, the description of Abstract Property Language (APL) is provided along with its syntax and semantics. Here, we also present the subsets of L_{mdg} [36] and PSL [6] considered in constructing APL.
- In Chapter 4, the requirement, the specification and the design of the proposed tool (Figure 1.3) are presented in details. Major part of the tool is a translator that translates APL specification to MDG-HDL. We provide the lexical and syntactic rules used in this translator as regular expressions and in BNF respectively.
- In Chapter 5, experimental results are presented. It includes performance analysis and analysis of generated circuit area. It also presents model-checking results of the Read-port of Look-Aside Interface [20].
- Finally we conclude in Chapter 6 with a brief discussion on future work.

1.3 Related Work

As related work to ours, we cite the work of Eric Gascard [16]. He presented a process of generating deterministic finite automata from a given PSL SERE expression [6]. The work describes a tool which generates a VHDL description of a monitor checking circuit from a PSL SERE formula. This is achieved by creating a parser for PSL language. The syntax tree produced by the PSL parser is then given to the automata construction module. The VHDL monitors are directly translated from the expressions. These monitors are later synthesized on FPGA (Field-Programmable Gate Array).

Marc Boulé and Zeljko Zilic described a way to generate hardware assertion checkers to be used in an emulation environment [25]. To use assertions in hardware emulation, they have introduced a checker generator tool, called MBAC, that can transform PSL assertion units into Verilog modules. The transformation includes boolean layer, temporal layer and *assert* directive of the verification layer. The Verilog modules are synthesized on FPGA. The assertion signals are externally monitored by the verification engineer.

Morin-Allory and Borrione showed a unique method of synthesizing monitors from PSL assertions [24]. In their work, primitive monitor blocks are built for each foundation language operator of PSL. Generic monitors are built for operators that can have different number of operands. These monitors are written in synthesizable subset of VHDL. Construction of complex monitors are done by interconnecting primitive monitors. They have proved the correctness of the monitors using PVS theorem prover [33]. To do this, each monitor is converted to its respective finite state machine (FSM) and translated into PVS input formalism.

Abarbanel et al. gave birth to FoCs (Formal Checkers), an automatic generation of simulation checkers [34]. It takes PSL assertions and translates them into HDL Checkers, which in turn are integrated into the simulation environment. These Checkers monitor the simulation results on a cycle-by-cycle basis for violation of the

properties. Each Checker implements a state machine that enters and asserts an error state if the respective property fails to hold in a simulation run. The language of the checkers can be VHDL or Verilog. The tool first translates a given assertion into non-deterministic finite state automaton (NFA). The obtained NFA is converted into a deterministic finite state automaton (DFA). Finally, HDL form of the checker is generated from the DFA. The translation process is based on the work of Ilan Beer [23].

The major difference between the work presented in this thesis and the works presented in the above paragraphs is the target application. The above mentioned related works are presented with application to either simulation or emulation based verification environment in mind. The research closely related to ours is the work done by Ying et al. [36], in creating L_{mdg} language and its processing tools, called L_{mdg}-Tools. In this case the targeted application is MDG based model-checking. In our work, we present a new language based on L_{mdg}, with added operators from PSL, and present an improved tool to perform the processing of specifications written in terms of properties. Given a model of a hardware design in MDG-HDL and its respective specification in terms of properties, the outcome of both L_{mdg}-Tools and our proposed tool is a verification-ready composite model. This model is combination of the original model and the generated monitor (checker) circuit, representing the specification, in MDG-HDL.

Chapter 2

Background

In this chapter, we provide a brief background on following topics:

- Classification of temporal logic and specification.
- Multiway Decision Graphs (MDGs).
- Abstract Verifier or MDG verification tool.
- Construction of Translators.

2.1 Temporal Logic and Specification

In model-checking methods, desired specifications are usually written in propositional temporal logic formulae [9]. This allows the user to write propositions with respect to time. The model of time is represented either in linear time (LTL) [29] or branching time (CTL) [26]. CTL* is a logic that combines the expressive power of LTL and CTL. It is also known as full branching-time logic.

2.1.1 Linear Time Logic

In linear time logic, the structure of time is a totally ordered set $(S, <)$, isomorphic to the set of natural numbers $(N, <)$ [17]:

If AP is a set of atomic propositions, a linear time structure is defined as $M(S, R, L)$, where:

- S: a set of states.
- $R \subseteq S \times S$: transition relation function with $\forall_{s \in S}\ \exists_{s' \in S}\ .\ (s,s') \in R$.
- $L : S \rightarrow 2^{AP}$ is a labeling of each state with the set of atomic propositions true in that state.

If R is a function, i.e. for every state there exists exactly one successor state, then we obtain a *linear* structure. Following is an example (Figure 2.1):

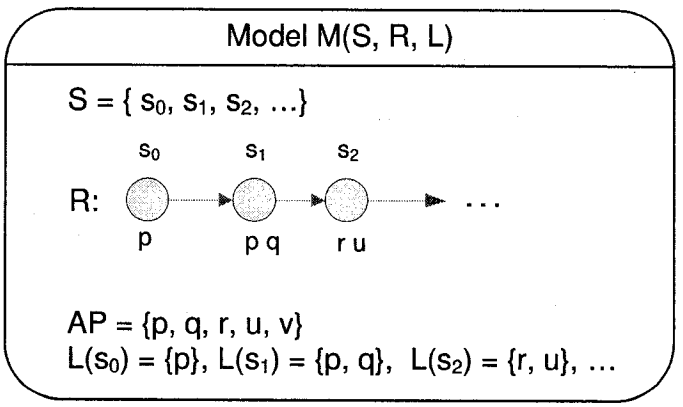


Figure 2.1: Model Structure

In propositional linear temporal logic (PLTL), one can use propositional logic as building block and apply temporal operators to specify properties. The *syntax* of it is defined as a least set of formulae generated by the following rules [17]:

1. Each atomic proposition is a formula;

2. If p and q are formulae then $\neg p$ and $p \wedge q$ are formulae;

3. If p and q are formulae then $p\mathrm{U}q$ and $\mathrm{X}p$ are formulae.

Semantics of a formula p of PLTL with respect to a linear-time structure $M(S, x, L)$, where x is the transition relation, is defined below. Here, we write $M, x \models p$ iff $p \in L(s_0)$ for atomic proposition p to mean that in structure M formula p is true on timeline (path) x; x^i denotes the suffix path s_i, s_{i+1}, s_{i+2}, and so forth.

1. $M, x \models p$ `iff` $p \in L(s_0)$.

2. $M, x \models \neg p$ `iff` not $M, x \models p$.

3. $M, x \models p \wedge q$ `iff` $M, x \models p$ and $M, x \models q$.

4. $M, x \models \mathrm{X}p$ `iff` $M, x^1 \models p$.

5. $M, x \models p$ U q `iff` $\exists j(M, x^j \models q$, and $\forall_{0 \leq i < j}(M, x^i \models p))$.

A PLTL formula p is satisfiable `iff` there exists a linear-time structure $M{:=}(S,$ $x,$ $L)$ such that $M, x \models p$, and any such structure defines a model of p.
Following are some examples (Figure 2.2):

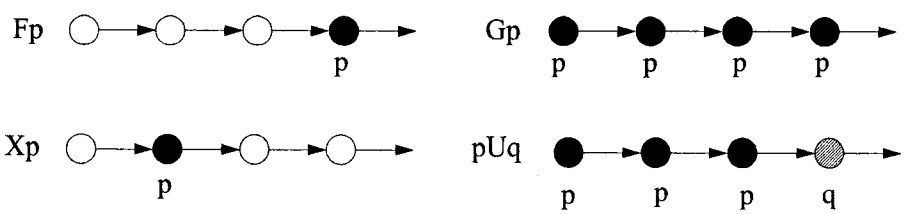


Figure 2.2: LTL formulae and time.

2.1.2 Computation Tree Logic

Computation tree logic (CTL) was first proposed by Clarke and Emerson [18]. It is based on branching time temporal logic (BTTL). Here, the time is modeled as a branching tree-like structure where each moment may have many different successor moments. Along each path, the timeline is isomorphic to the natural number. To specify a property in CTL, we simply apply the path operators along with temporal operators to the propositional building blocks. There are two strict restrictions in CTL:

1. The LTL operators F, G, X and U are immediately preceded by a path quantifier.

2. Time operators cannot be combined directly with the propositional connectives.

The syntax of CTL are governed by the following rules:

1. Every proposition is a CTL formula;

2. If p and q are CTL formula, then so are $\neg p$, $(p \wedge q)$, AXp, EXp, A(pUq), E(pUq).

The remaining operations can be derived from the above rules. The truth of a formula is determined on a given state and not on a branch of the time structure. The structure resembles an infinite computation tree. A temporal formula p is satisfied by a model M with transitions T, if it is true for all the initial states s_0 of the model. The semantics of CTL formula is given below:

1. $M, s_0 \models p$ iff $p \in L(s_0)$.

2. $M, s_0 \models \neg p$ iff not $M, s_0 \models p$.

3. $M, s_0 \models p \wedge q$ iff $M, s_0 \models p$ and $M, s_0 \models q$.

4. $M, s_0 \models \mathrm{AX}p$ iff for all states s_0' with $(s_0, s_0') \in T$, $M, s_0' \models p$.

5. $M, s_0 \models \mathrm{EX}p$ iff for some state s_0' with $(s_0, s_0') \in T$, $M, s_0' \models p$.

6. $M, s_0 \models \mathrm{A}[p\mathrm{U}q]$ iff for all paths $(s_0, s_1, \ldots)$, there exists a $j \geq 0$ with $M, s^j \models q$, and $M, s^i \models p$ holds $\forall_{0 \leq i < j}$.

7. $M, s_0 \models \mathrm{E}[p\mathrm{U}q]$ iff for some path $(s_0, s_1, \ldots)$, there exists a $j \geq 0$ with $M, s^j \models q$, and $M, s^i \models p$ holds $\forall_{0 \leq i < j}$.

Figure 2.3 shows intuitive meanings of some CTL formulae.

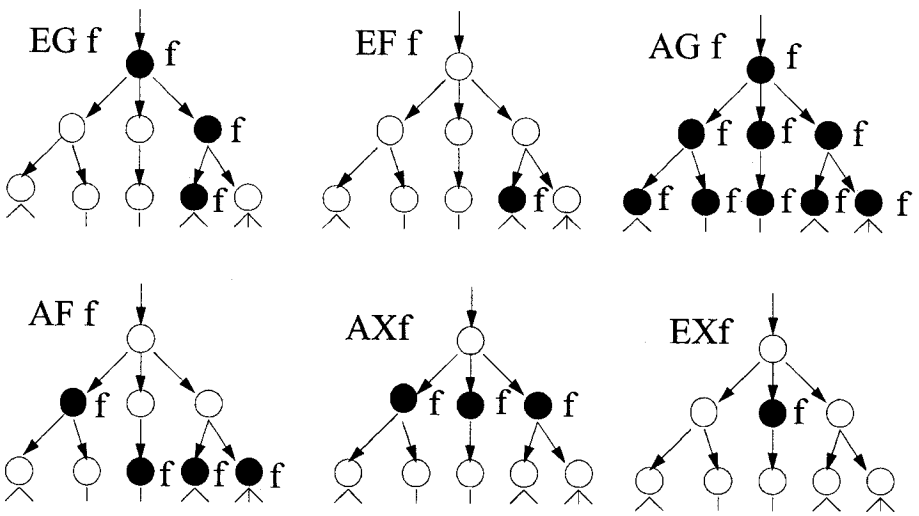


Figure 2.3: CTL formulae and time.

2.1.3 Full Branching-Time Logic

This class of logic formula combines the branching-time and linear-time operators. In CTL*, a path quantifier can be a prefix to an assertion composed of arbitrary combination of the temporal operators: F, G, X and U. Like CTL, the tree is formed by designating a initial state s_0 in model M, and then unwinding the structure into

an infinite tree with s_0 as the root. The semantics of the path quantifiers and temporal operators remain the same.

2.1.4 Categories of specification

The specifications are written as properties of the system. They are categorized as follows:

1. Safety property: ensures that nothing 'bad' will ever happen. Depicted as $\models$ Gp, where p is true at the time.

2. Liveness property: ensures that something 'good' will eventually happen. Depicted as $\models$ Fp, where p will eventually be true at some point in the future.

3. Precedence property: ensures precedence order of events. Depicted as $\models p$Uq, where q is true in present time or p is true until q becomes true.

Among the three, safety property is the most used when writing specifications of a design under verification.

2.2 Multiway Decision Graphs

The underlying formal system of MDGs is a subset of many-sorted first-order logic improved with a distinction between *abstract sorts* and *concrete sort.* Concrete sorts have finite enumerations, while abstract sorts do not. The enumeration of a concrete sort α is a set of distinct constants of sort α. The constants occurring in enumerations are referred to as individual constants and the other constants as generic constants. They could be viewed as `0-ary` function symbols. The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols.

The vocabulary of many-sorted first-order logic consists of sorts, constants, variables, and function symbols or (operators). A function symbol is defined as follows:
Let f be a function symbol of type $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$.

1. If α_{n+1} is an abstract sort, then f is an *abstract function symbol.*
2. If $\alpha_1 ... \ \alpha_{n+1}$ are concrete, then f is a *concrete function symbol.*
3. If α_{n+1} is concrete while at least one of the $\alpha_1 ... \ \alpha_n$ is abstract, then f is referred to as *cross-operator.*

Concrete function symbols must have explicit definition. Abstract function symbols and cross-operators are *uninterpreted.*

An *interpretation* is a mapping Ψ that assigns a denotation to each sort, constant and function symbol, such that:

1. The denotation $\Psi(\alpha)$ of an abstract sort α is a non-empty set.
2. If α is a concrete sort with enumeration $a_1, a_2, \cdots, a_n$ then
   $\Psi(\alpha) = \Psi(a_1), \Psi(a_2), \cdots, \Psi(a_n)$ and $\Psi(a_i) \neq \Psi(a_j)$ for $1 \leq i < j \leq n$.
3. If c is a generic constant of sort α, then $\Psi(c) \in \Psi(\alpha)$.
4. If f is a function symbol of type $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$ then $\Psi(f)$ is a function from cartesian product $\Psi(\alpha_1) \times \cdots \Psi(\alpha_n)$ into the set $\Psi(\alpha_{n+1})$.

If X is set of variables then a variable assignment with domain X, compatible with an interpretation Ψ, is a function φ that maps every variable $x \in X$ of sort α to an element $\varphi(x)$ of $\Psi(\alpha)$. We write Φ_X^Ψ for the set Ψ-compatible assignments to the variables in X. $\Psi, \varphi \models P$ if P denotes truth under an interpretation Ψ and a Ψ-compatible variable assignment φ to the variables that occur free in P. $\models P$ if a formulae P denotes truth under every interpretation Ψ and every Ψ-compatible variable assignment to the variables that occur free in P.

2.2.1 Abstract State Machine

In MDGs, a state machine is described using finite sets of input, state and output variables, which are pair-wise disjoint. The behavior of a state machine is defined by its transition/output relations including a set of reset states. An abstract description of the state machine, called Abstract State Machine (ASM) [19], is obtained by letting some data input, state or output variables be of abstract sort(s), and the datapath operations be uninterpreted function symbols. As ROBDDs are used to represent sets of states and transition/output relations for finite state machines (FSM), MDGs are used to compactly encode sets of (abstract) states and transition/output relations for ASMs. This technique replaces the *implicit enumeration* technique [28] with the *implicit abstract enumeration* [15].

2.2.2 Structure of MDG

MDGs can be viewed as a generalization of BDDs. With BDDs, we can represent formulae written in boolean logic with leaf nodes labeled with boolean values. But it can also be viewed as representing an assertion, with the leaf nodes labeled by propositions. This idea can be generalized to accommodate abstract types. For example, let G represent the graph of the boolean formula $(\neg x \wedge F_0) \vee (x \wedge F_1)$, where, F_0 and F_1 are the boolean formulas represented by the sub-graphs G_0 and G_1 respectively. In many sorted first-order logic, the graph G can be viewed as representing a formula:

$$((x = 0) \wedge F_0) \vee (x = 1) \wedge F_1)).$$

Three possible generalizations of G and the corresponding formulas are shown in Figure 2.4. F_0, F_1 and F_2 are first-order formulas represented by the sub-graphs G_0, G_1 and G_2 respectively. Sorts and functions are annotated in the graphs.

1. From G to G': $x \in \{0,1\} \rightarrow x \in \{0,2,3\}$, and Graph G' represents the

formula:

$$((x = 0) \wedge F_0) \vee ((x = 2) \wedge F_1) \vee ((x = 3) \wedge F_2).$$

2. From G to G'': $x \in \{0, 1\} \rightarrow x \in \{a, y, f(a, y)\}$, and Graph G'' represents the formula:

$$((x = a) \wedge F_0) \vee ((x = y) \wedge F_1) \vee ((x = f(a, y)) \wedge F_2).$$

3. From G to G''': $x \in \{0, 1\} \rightarrow g(x) \in \{0, 2, 3\}$, and Graph G''' represents the formula:

$$((g(x) = 0) \wedge F_0) \vee ((g(x) = 2) \wedge F_1) \vee ((g(x) = 3) \wedge F_2).$$

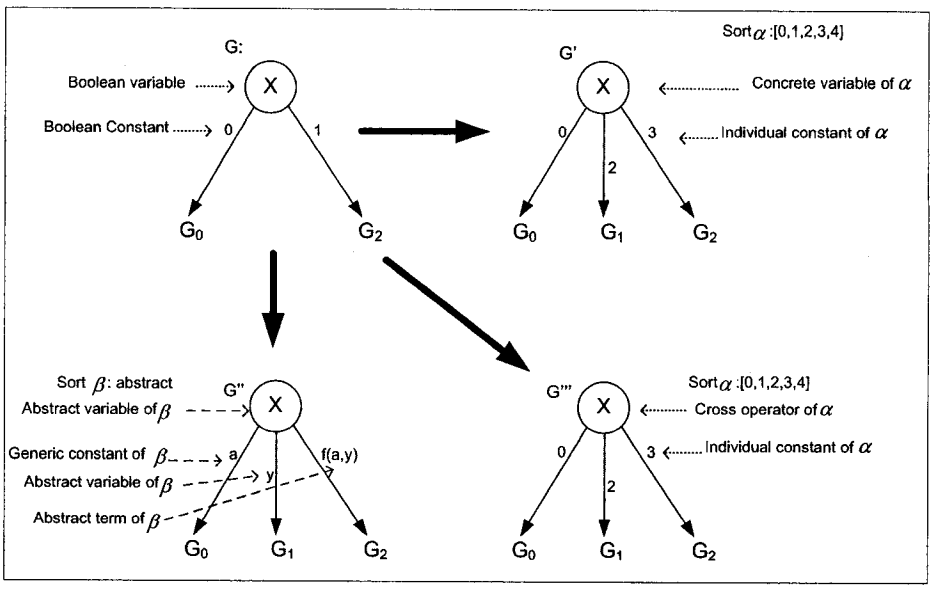


Figure 2.4: BDDs to MDGs.

The above generalized decision graph G', G'' and G''' are examples of Multiway Decision Graphs (MDGs).

2.2.3 Verification Algorithms

The MDG software package includes algorithms for disjunction, relational product, pruning-by-subsumption (PbyS), and reachability analysis. Except for PbyS, the operations are a generalization of first-order terms of algorithms on ROBDD, with some restrictions on the appearance of abstract variables in the arguments. In the reachability analysis procedure, starting from the initial set of states, the set of states reached in one transition is computed by the relational product operation. The frontier set of states is obtained by removing the already visited states from the set of newly reached states using the pruning-by-subsumption (PbyS) operation. If the frontier set of states is empty, then the reachability analysis procedure terminates, since there are no more unexplored states. Otherwise the newly reached states are merged (using disjunction) with the already visited states and the procedure continues where the next iteration with the states in the frontier set as the initial set of states. A facility to carry out simple rewriting of terms that appear in the MDGs is also included. This allows algorithms to provide a partial interpretation of an uninterpreted function symbol. A detailed description of the operations and algorithms can be found in [15]. The following sub-sections describe the important verification methods provided by the MDG tools.

Combinational Equivalence Checking

The MDGs representing the input-output relation of each circuit are computed using the relational product of the MDGs of the components of the circuits. Then taking advantage of the canonicity of MDGs, it is verified whether the two MDG graphs are isomorphic. Using this technique, we can verify the equivalence of two combinational circuits.

Sequential Equivalence Checking

The behavioral equivalence of two sequential circuits can be verified by checking that the circuits produce the same sequence of outputs for every sequence of inputs. This is achieved by forming a circuit from two circuits by feeding the same inputs to both and verifying an invariant asserting the equality of the corresponding outputs in all reachable states.

Model Checking

MDG model checker provides both *safety* and *liveness* property checking facilities using the *implicit abstract enumeration* of an abstract state machine. In MDG model-checking, the design is represented in MDG-HDL and the properties to be verified are expressed by formulae in L_{mdg}. The ASM model of L_{mdg} and the RTL model is composed along with a simplified invariant. The simplified invariant is checked on the composite machine using the *implicit abstract enumeration* of the ASM.

2.3 Abstract Verifier

Abstract Verifier, also known as MDG Verification Tool, consists of set of tools. These include a composite model generator in L_{mdg} tools (Figure 2.9) at the front-end and a collection of services that are implemented as a library package under Quintus Prolog [2] at the back-end. The package includes a reachability-analysis procedure, applications for combinational and sequential hardware verification at RT level [39], and model-checking applications for a subset of first-order abstract CTL* [36].

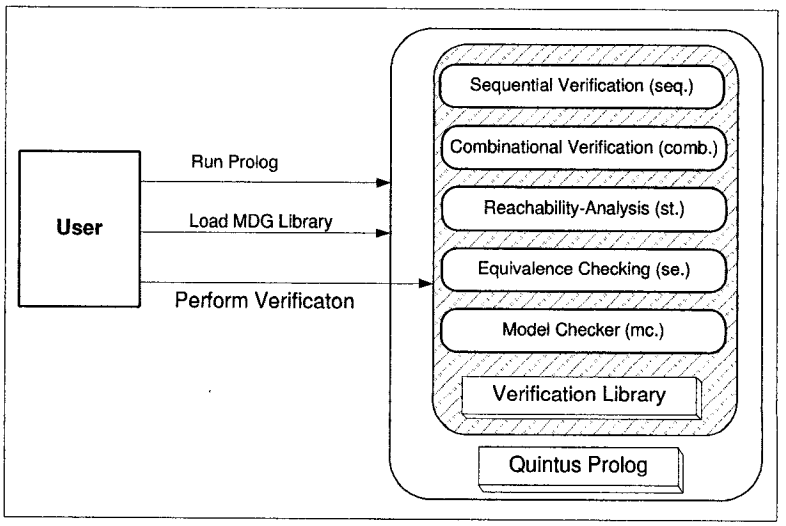


Figure 2.5: MDG Verification

A user needs to represent the model under verification in MDG-HDL according to its language specification [38]. He then needs to load the MDG library (Figure 2.5) in Prolog to run applications available in the library package. If model-checking needs to be performed, L_{mdg} set of tools must be used to get a composite model. The user must modify the original model using any text editor and write its specification in L_{mdg} language in accordance with the restrictions provided in *MDG Model Checker User's Manual* [35]. In the following subsections, we provide instructions on how to represent models in MDG-HDL and the process of MDG based model-checking.

2.3.1 MDG-HDL

MDGs describe circuits at the RT level as a collection of components interconnected by nets that carry signals. Each signal can be an abstract variable or a concrete variable. The input language for MDG based applications is MDG-HDL. It allows the use of abstract variables for representing data signals and uninterpreted function

symbols for representing data operations. MDG-HDL mainly supports structural description. In order to represent a design, a user must create three files: an algebraic description file, a circuit description file and an order description file. To perform model-checking, he also needs to feed the model-checker with the property file. In the following subsections, we provide brief discussion on each step.

Algebraic Description File

In this file the user defines custom signal types (sorts), function types and generic constants used in the hardware description. The built-in types are given in common.pl in MDG package library [38]. If rewrite-rules are needed to interpret the function symbols, they are also specified in this file.

- *Sorts*: Sorts can be one of the two types: abstract or concrete.

  1. `abs_sort(Sort).` - declares Sort as an abstract sort.
     - Example: `abs_sort(wordn).` - declares `wordn` is an abstract sort.
  2. `conc_sort(Sort, [list_of_values]).` - declares `Sort` as a concrete sort having list of constant values as its enumeration.
     - Example: `conc_sort(bool,[1,0]).` Here, `bool` is a concrete sort whose enumeration is given in the list $[1, 0]$, where 1 and 0 are treated as individual constants.

- *Functions*: There are two types of functions in MDG-HDL: abstract function symbol and cross function symbol.

  `function(Func_symbol, Args_sorts, Target_sort)` - declares a function type. `Func_symbol` is the function symbol. It can be an abstract function symbol or a cross-operator. `Args_sorts` is a list of types of the arguments and `Target_sort` is the type of the function return.

1. Example: `function(add,[wordn,wordn],wordn)`. Here, `add` is uninterpreted but could mean addition. It is a function that takes two arguments, whose sorts are `wordn`, and returns a result of sort `wordn`. In this case, `wordn` must be abstract sort and `add` is an abstract function symbol.

2. Example: `function(leq,[wordn,wordn],bool)`. `leq` is a function which is uninterpreted. It intends to mean less then or equal. The function has two arguments of abstract sort `wordn` and returns a result of boolean sort `bool`. Here, `leq` is called a cross-function symbol (or cross-operator or cross-op).

- *Generic constants*: The generic constants are declared the following way:

  - `gen_const(Gen_const, Sort)` declares a generic constant `Gen_const` having the type Sort.

  - Example: `gen_const(min,wordn)`. Here, `min` is a generic constant of sort `wordn`.

- *Re-writing rules*: The re-writing rules are written using two keywords: `rr` and `xtrr`.

  1. `rr`(C_s, LHS, RHS) is a conditional rewrite rule: $C_s \Rightarrow LHS \rightarrow RHS$. Here, LHS (left hand side) and RHS (right hand side) are terms. C_s is a list of conditions. A condition can be a pair (Xt_t, C) composed of a cross-term and an individual constant, or it is simply a term which is a goal of the predicate.

     - Example: `rr`$([(eq(X,Y),0),(iszero(X),1)], iszero(Y),0)$. It means if $(eq(X,Y) = 0)$ and $(iszero(X) = 1)$ then $iszero(Y)$ equals 0. Here, $[(eq(X,Y),0),(iszero(X),1)]$ is the condition list.

  2. `xtrr`(C_s, LHS, RHS) is a conditional rewrite rule for cross-terms:

$C_s \Rightarrow LHS \rightarrow RHS$. LHS must be a cross-term and RHS must be an individual constant. C_s may contain arithmetic expressions, but must not contain cross-term and individual constant pairs.

- Example 1: `xtrr`([], $eq(X, X), 1$).
  It represents the rewrite rule: $X{=}Y \Rightarrow eq(X, Y) \rightarrow 1$. The condition $X{=}Y$ (syntactic equivalence) is implicitly expressed in the rule. The condition list is thus empty.
- Example 2: `xtrr`($[square(X, Y)], sq(X, Y), 1$).
  square(X, Y):- Y is X^* X. It represents the rewrite rule:
  $X^2{=}Y \Rightarrow sq(X, Y) \rightarrow 1$. The condition $X^2{=}Y$ is an arithmetic expression. It is evaluated by a *Prolog predicate*: $square(X, Y)$:- Y is $X * X$.

Circuit Description File

A circuit can be described as a structural description, a behavioral ASM description, or a mixture of structural and behavioral descriptions. A structural description is usually a net-list of components (predefined in MDG–HDL) connected by signals. A behavioral description is the transition/output relations of an ASM given by table construct. Following are the basic constructs:

1. *Signal declaration*: Each signal must be declared only once in the following format:
   - Format: `signal(name, type)`. Here, `name` is the name of the signal and `type` is the sort of the signal.
   - Example: `signal(mySig, bool)`. - declares a signal `mySig` with type `bool`.
2. *Component declaration*: Components are the basic building blocks in MDG-HDL.

- Format: `component(name, definition)` - declares a component *name* that is defined by *definition*.
- Examples: Following are some definitions of components from the set of predefined components in MDG-HDL:
  (a) NOT gate: `not(input(Input: bool), output(Output: bool))`.
  (b) Two-input and-gate: `and(input(Input1: bool,Input2: bool), output(Output: bool))`.
  (c) Register with control signal: `reg(control(Control:bool), input(Input:Sort), output(Output: Sort))`.
  (d) Uninterpreted function block: `transform(inputs(Inputs:[`$Sort_1$`, `$Sort_2$`, ..., `$Sort_n$`]), function(Func_symbol), output(Output:Sort))`.

3. *Table declaration*: Tables are considered as components. It is similar to a truth table, but it allows first-order terms in the rows. In the list of rows, the first row is a list containing variables and cross-terms. The last element of the list must be a (concrete or abstract) variable. All the other variables in the list must be concrete variables. Starting from the second row, each row is a list of values that the corresponding variables or cross-terms can take. The last element in the value list could be a first-order term, which means an assignment to the output variable. The other elements in the list must be either *don't-cares* (represented by '*') or individual constants in the enumeration of their corresponding variable sort. The last element a row may be a term which serves as the default value.

   - Format: `table(rows)`.
   - Example: `component(myTable, table([[myInput,myOutput], [1, 0] | 1]))`.

     It specifies a NOT gate, where `myOutput` has value 0 if `myInput` has value 1; `myOutput` has value 1 if `myInput` is not 1.

4. *State variable initialization*: User needs to assign starting point for state variables.

   - Format: `init_val(Stvar, InitVal).` - assigns InitVal to Stvar.
   - Example:`init_val(myState, s0).` - assigns s0 as the starting state to myState variable.

5. *State variable generalization*: For abstract state variables, the initial states should be generalized.

   - Format: `init_var(InitVal, Sort).` - declares InitVal as a generalized state variable sort to be used in initializations.
   - Example: `init_val(pc,init_pc).  init_var(init_pc,wordn).`

6. *Next state declarations*: For sequential circuit, the user should provide the mapping between state variables and next-state variables.

   - Format: `st_nxst(Stvar, NextStvar)` declares that *NextStvar* is the next-state variable of a state variable *Stvar*.
   - Example: `st_nxst(myStateVar, n_myStateVar).`

7. *State partition declaration*: It is a 3-level nested list. The inner most list gives a list of next-state variables. The transition relation MDG, containing all the variables in the list, forms an individual transition relation. The middle level list specifies a default partition block.

   - Format: `next_state_partition(Partitions).`
   - Example: `next_state_partition([[[n_a]],[[n_b]],[[n_c]]]).`

8. *Output partition declaration*: It declares Outputs as a list of outputs of the circuit.

   - Format: outputs(Outputs).

- Example: outputs([myOutput]).

9. *Partition strategy declaration*:

   - Format: `par_strategy(OrdMethod, ParMethod).` - declares OrdMethod as the ordering strategy for individual relations and ParMethod as the partitioning strategy.
   - Example: `par_strategy(auto, auto).`

Complete details of the MDG-HDL language can be found in its user manual [38].

Order Description File

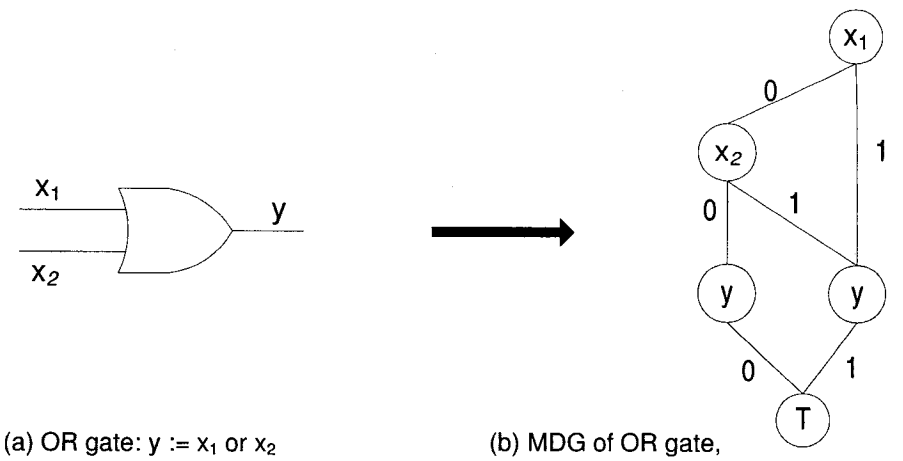
Like ROBDDs, the MDGs require a total order over all the nodes in the graph. This order is manually provided by the user using `order_main(Symbols)` function where `Symbols` is a list of variables and cross-operators. We say that symbol a comes before symbol b if a is at the left side of b in the list. In MDG, a will appear above b. We may also write it as $a < b$.

Example: `order_main([r,x,c_A,n_c_A,rm_A,n_rm_A,rM_A,n_rM_A,leq]).` For node ordering of abstract variables and cross-operators, MDGs have some requirements:

1. If a variable a will appear as a secondary variable in an edge label of node b, then $a < b$.
2. If a variable a will appear as a secondary variable in a cross-term having cross-op f, then $a < f$.
3. The state variables and next state variables must be in a correspondent order.
4. If using rewriting rule `rr(`C_s`, LHS, RHS)`, the cross-operators in C_s must come before the terms in LHS in the graph structure. For example, for rule: `rr([(eq(X,Y), 0), (iszero(X), 1)], iszero(Y), 0)`, cross-operator `eq` should come before `iszero`.

Modeling Examples

For logic gates, the input and output signals are always of concrete sort with boolean values. Abstract sort is pre-defined in the MDG library package. Cross operators are used when feedback of a datapath is used as part of control. If not, a function symbol is used. Both, however, are uninterpreted. A simple logic-gate design is given as an example below containing concrete sorts. The Figures 2.6 and 2.7 show MDG-HDL representations of an OR-gate and a comparator respectively. More details can be found in the user manual of MDG-Tools [38].

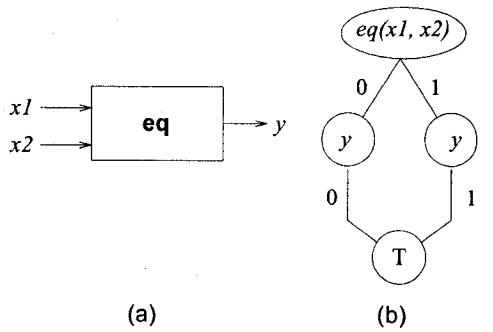


(a) OR gate: $y := x_1$ or x_2

(b) MDG of OR gate, $y := x_1$ or x_2

```
% Algebraic description:
  conc_sort(bool,[0,1]).
% Circuit description:
  signal(x1,bool).
  signal(x2,bool).
  signal(y,bool).
  component(myAND, and(input(x1,x2),output(y))).
% Order description:
  order_main([x0,x1,y]).
```

Figure 2.6: OR gate in MDG and its corresponding MDG-HDL description.




```
%Algebraic description:
abs_sort(wordn).
function(eq,[wordn,wordn],bool)}.
xtrr([],eq(X,X),1).
%Circuit Description:
signal(x1,wordn).
signal(x2,wordn).
signal(y,bool).
component(comp_function,transform(inputs([x1,x2]),
                                  function(eq),output(y))).
%Order description:
order_main([x0,x1,eq,y]).
```

Figure 2.7: MDG-HDL description of a Comparator

2.3.2 Specification Language for MDG Model-Checking

The property language of MDG model-checking methodology, L_{mdg}, is tailored to fit the powerful model-checking algorithms available in MDG. It is based on abstract CTL* [36]. The formulae are divided into two categories: state formulae and path formulae. State formulae give specification on certain states in the system and path formulae elaborate on them with respect to time using temporal operators. Its semantics can be found in the user manual of MDG model-checker [35].

Example: Here, we present some properties written in L_{mdg} specification language. Given the model of Figure 2.8 represented in MDG-HDL, the specification

can be written as follows:

1. `AG ( state = c_Fetch & input = c_Inc2) -> X(state = c_Inc1));`

   If *state* is `c_Fetch` and *input* is `c_Inc2`, then in the next transition *state* is `c_Inc2`.

2. `AG ((state = c_Fetch & input = c_Load) -> XX(state = c_Fetch));`

   If *state* is `c_Fetch` and *input* is `c_Load`, then in two transition steps *state* is `c_Fetch`.

3. `AG((state = c_Fetch & input = c_Inc2) ->`

   `(LET(pc = v1) IN (XXX(pc = finc(finc(v1))))));`

   If *state* is `c_Fetch` and *input* is `c_Inc2`, then in three transition steps the value of *pc* will be incremented two times. Here, the value of *pc* in the present state is stored in *v1*. The comparison is made between the value of *pc* after three transition steps and incrementing the old value of *pc* twice.

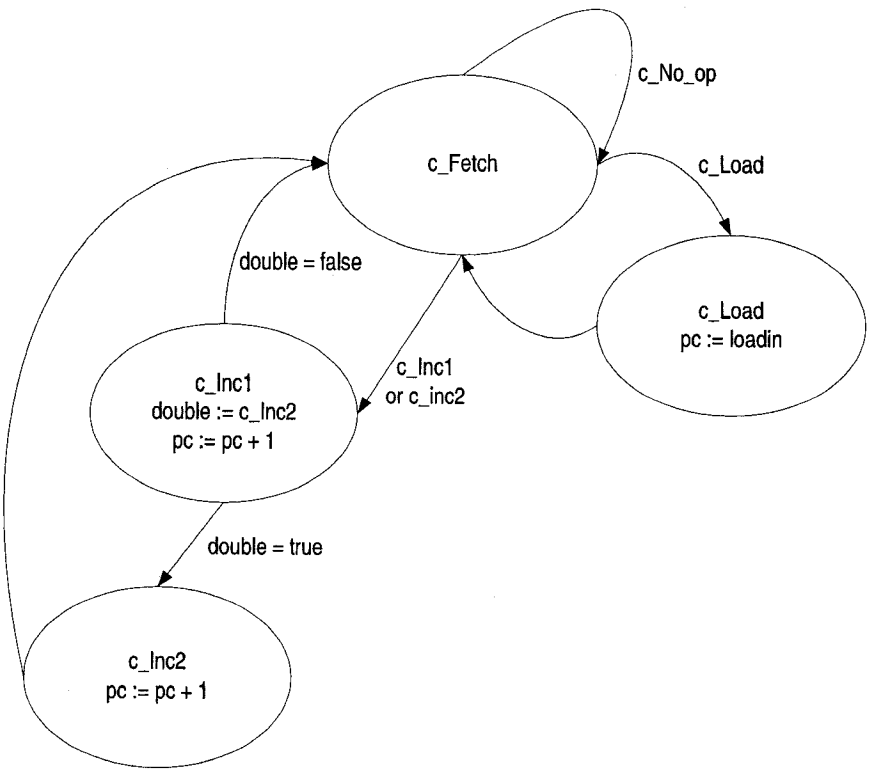


Figure 2.8: An abstract counter

L_{mdg} language is discussed in detail in the next chapter.

2.3.3 Model-Checking in MDG

To perform model-checking based on MDG, following steps are taken:

1. Represent the *design under verification* in MDG-HDL.
2. Perform *state exploration* to make sure that no error exists. The *state exploration* service in MDG software package detects and reports all syntax and semantic errors before performing reachability analysis.
3. Edit the model according to restrictions given in the manual [35].
4. Write the specification according to the manual.

5. Run the L_{mdg} tools to generate the composite model and the condition file(s).

6. Run model-checker to verify the model.

To verify subsequent specifications, user needs to repeat steps 4 to 6 in the above list. The user shall creates a few additional files in order to use MDG model-checker, part of MDG verification tool implemented as a library package under Quintus Prolog [2]:

1. A make.pl file that tell *prolog* application where the MDG library package is. After running *prolog*, a user loads the make file.

2. A circuit.pl file that provides the MDG applications the filenames of the models and their locations. This file can be automatically generated as well.

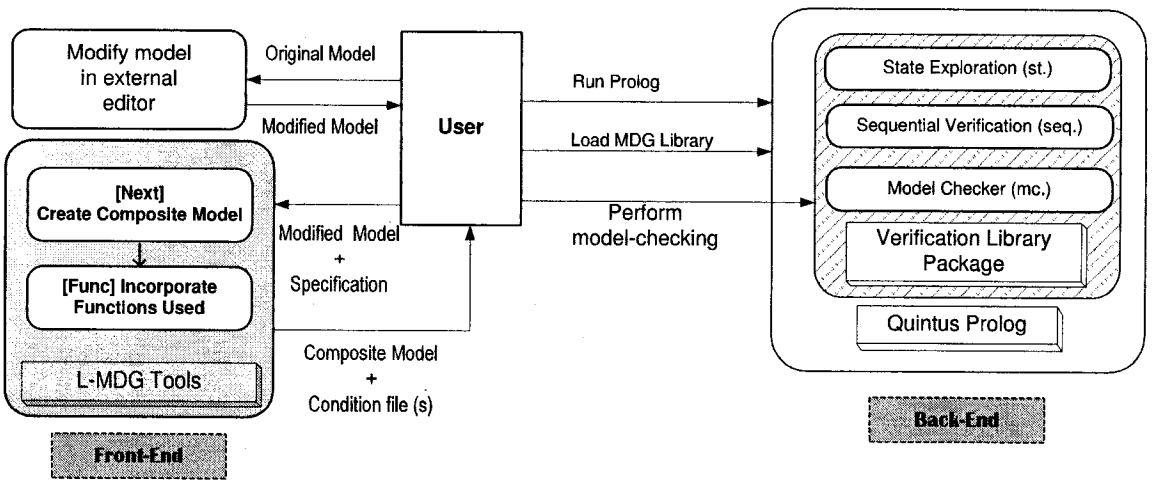


Figure 2.9: Model-Checking in MDG

Figure 2.9 shows the modules involved in the model-checking process. When a user runs the sequential verification application, it loads all the necessary algorithms that are required to run the model-checker. A user must do it before running the model-checker application. When the model-checker application is run, the user must select a property template from the following list:

1. A	2. AG
3. AF	4. AU
5. E	6. EG
7. EF	8. EU
9. AGAF	10. AGAU
11 .AF with fairness	12. AGAF with fairness
13 .AU with fairness	14. AGAU with fairness

Based on the chosen template, user must specify the filenames of the composite model and the condition files. The model checker than runs the algorithm and produces a result: *yes* or *no*. This informs the user if the specification is satisfied or not.

2.4 Construction of Translators

In this thesis, the proposed tool shall have a component that can perform a *translation* between Abstract Property Language (APL) to MDG-HDL. Thus, in this section, we provide the basic construction technique of such module that can perform the required operation. We follow the procedure of compiler construction for this purpose.

In its most general form - *a compiler is a program that accepts as input a program text in certain language and produces as output a program text in another language* [11]. This conversion process is called *translation.* The input language of this process is denoted as the *source language*, and the output language is denoted as the *target language.*

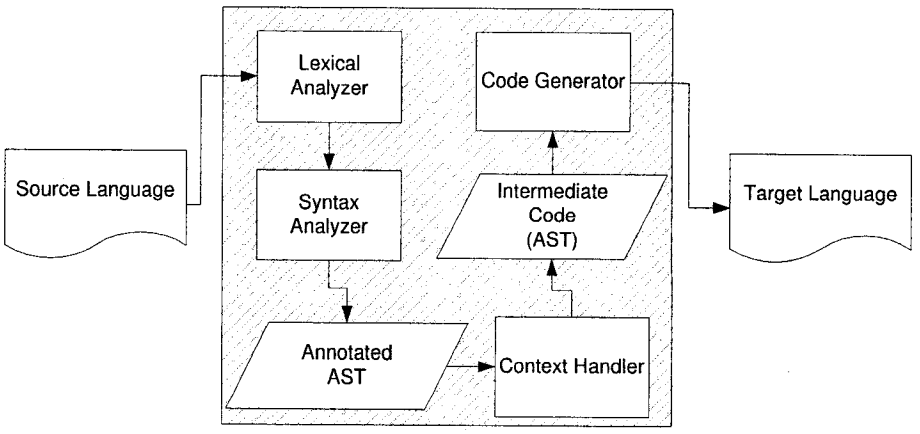


Figure 2.10: Basic structure of a compiler.

Figure 2.10 shows the basic structure of a compiler. The components are: Lexical Analyzer, Syntax Analyzer, Semantic Analyzer and Code Generation. A code optimizing module is usually present in compiler constructions. Since it is not used in building the proposed tool, we have omitted the module.

2.4.1 Lexical Analyzer

A lexical analyzer or lexer takes an arbitrary input stream and tokenizes it into lexical tokens. In this thesis, the lexical analyzer is built using a tool called Flex [22]. Flex automatically generates a lexical analyzer given its lexical specifications. The specifications are set of patterns written using regular expressions that are matched against the input. Each time one of the pattern is matched, corresponding action code is executed. In such a case, the *action* returns appropriate token to the syntax analyzer. Attributes of the tokens are also saved along with the tokens.

Example:

1. Pattern: `id [a-z][a-zA-Z0-9_]*`

2. Action: Return the token `ID` and store its attributes.

When pattern of `id` is matched, its attributes are saved and token `ID` is provided to the syntax analyzer.

2.4.2 Syntax Analyzer

A syntax analyzer determines if its input is syntactically correct given a set of syntax rules and also determines its structure. The syntax rules are given as grammar of the language. The structure is usually an abstract syntax tree (AST). AST is also known as parse-tree. The AST accurately shows how the segments of program text are to be viewed in terms of the specified grammar. In this thesis, Bison [21] is used to build the syntax analyzer. In Bison, a user specifies the grammar in Backus-Naur Form (BNF) and provides corresponding actions per production rule in order to build the AST. User also needs to specify the association types of the operators and their precedence. When an AST is built, the nodes are annotated with its attributes. It is then called an annotated AST. In this thesis, AST means annotated AST.

Example:

The grammar given below presents a grammar of an expression that can be an identifier, or it can be a *plus* operation between two expressions. Figure 2.11 shows an example AST of the expression $a + b + c$, where the *plus* operator is left associative.

- Grammar:

  1. `expression: expression + expression`
  2. `| id;`

- Actions:

  1. For the first line: create a node in the AST with corresponding value of *plus* and the expressions as its children.

2. For the second line: create a node in the AST with associated value of *id* with two NULL children. Here the *id* is a leaf node.

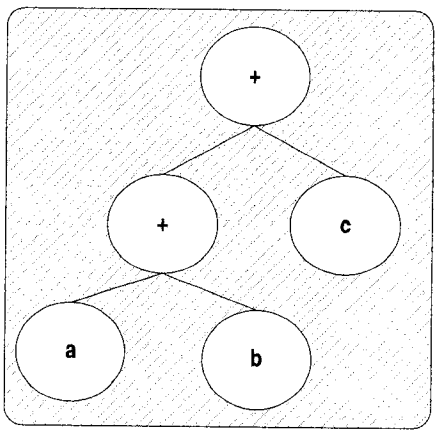


Figure 2.11: AST for a+b+c.

2.4.3 Context Handler

Based on the attributes of tokens and the context of use, this process checks the semantics of program text. It is also known as *semantic analyzer*. For example, a *plus* can be an arithmetic operation or a logical OR. Based on the semantic, its usage should be analyzed. In this case, the operands should be either logical or numeric. During semantic analysis phase, optimization information is collected, and later it is used in the code generation.

2.4.4 Code Generator

A code generator produces statements in target language through a process called *code generation*. It is a process of systematic replacement of nodes and sub-trees of the AST by target code segments. This process preserves the semantics. The AST is re-written to produce a linear sequence of instructions based on the target

language. For example, if we consider the AST given on Figure 2.11, as representing some statement in the source language and consider a target language supporting procedure $plus(input_1, input_2, output)$, we can do the following:

1. a+b generating a target statement: $plus(a, b, output_1)$.

2. (a+b) + c generating a target statement: $plus(output_1, c, output_2)$.

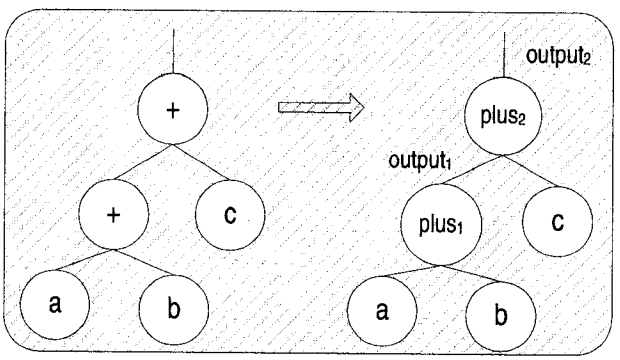


Figure 2.12: AST to target code.

Figure 2.12 shows one of the ways it can be done. Here, $plus_1$ can be replaced with $output_1$ and $plus_2$ with $output_2$. In this thesis, simple traversal algorithms are used to generate the target text from AST. The details of which are given in Chapter 4.

Chapter 3

Language Description

The purpose of using formal logic to define hardware specification is to avoid ambiguities associated with descriptions in natural languages. Defining specification in formal logic also facilitates automation of verification processes. In model-checking based verification, specification languages, as expected, are based on temporal logic. They allow verification engineers to specify how the behavior of a system should progress over time. This is possible in temporal logic without explicit introduction of time. *Abstract property language* (APL) described in this chapter is such a language.

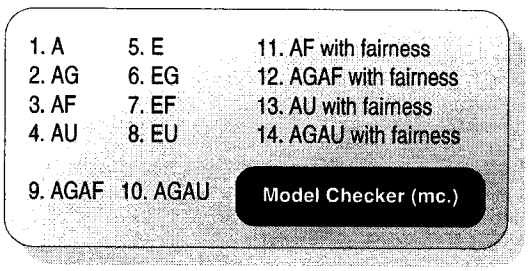


Figure 3.1: MDG Model-Checking Algorithms.

In this chapter, we provide detail description of the proposed language. We introduce *Property Specification Language* (PSL) and L_{mdg} *specification language*

and then define subsets of the two languages that are considered in constructing APL. When we define these subsets, we must keep in mind the model-checking algorithms implemented in the *MDG model-checking* package.

The package supports 14 different formats of temporal logic (Figure 3.1). There are also four algorithms dealing with fairness formulae that allow users to put constraints on the environment of the design. In this thesis, we only focus on properties based on the first 10 algorithms given in Figure 3.1.

3.1 PSL and its subset in APL

Property Specification Language (PSL) has recently become an IEEE standard in 2005 [6]. With its crisp syntax and comprehensive formal semantics, it is becoming the primary choice among assertion languages intended for hardware verification. In this section, we present a brief introduction to PSL and its subset relevant to APL.

The PSL language structure is based on four layers: Boolean Layer, Temporal Layer, Verification Layer and Modeling Layer.

3.1.1 Boolean Layer

This layer consists of expressions with valuations true or false. The boolean operators in this layer are from the respective underlying HDL flavor. For APL, the underlying layer is MDG-HDL. It is a structural language where boolean operations are represented by logic components and not by boolean expressions. However, for its specification language, we are free to choose boolean expressions to represent logical operations, as the composite model shall be generated by the proposed tool. We choose the logical operators given in Table 3.1 to be used in APL. The primary reason for considering these operators is that most users are accustomed to using them in this syntax.

Table 3.1: Logical Operators considered for APL

Operator	Logical Operation	HDL Flavor
!	NOT	Verilog, SystemVerilog , SystemC
||	OR	Verilog, SystemVerilog , SystemC
&&	AND	Verilog, SystemVerilog , SystemC
→	IMPLICATION	Verilog, SystemVerilog , SystemC
↔	EQUIVALENCE	Verilog, SystemVerilog , SystemC

Bit and bit-vectors can be suitably represented by creating concrete signals in MDG-HDL. Union operator cannot be considered because non-deterministic assignments of concretes are not applicable in MDG. The supported expressions and operators are constrained by the rules of simple subset of PSL [6].

3.1.2 Temporal Layer

This layer consists of temporal properties that allow us to represent relationships between boolean expressions over time structure. Since PSL is mostly used in simulation environment, it offers weak and strong temporal operators. Weak operators allow assertions to pass if simulation traces (finite) end prematurely and nothing else fails. In such cases, use of strong operators results in failed assertions. By definition, in formal verification we are dealing with 'infinite traces'. In MDG based model-checking algorithms, operators are treated as strong by default. Therefore, we do not distinguish operators based on the type. In PSL, LTL and SERE style temporal properties together are known as *Foundation Language* (FL). In all styles of temporal properties in PSL, we follow the rules of simple subset [6], briefly discussed in this section. We shall then discuss *Foundation Language* and CTL based *Optional Branching Extension* styles of PSL.

Simple Subset

In simple subset of PSL, the time is seen as flowing from *left to right* through the property. The intuitive meaning of it is that if we need to evaluate an atomic entity at cycle Δ, then valuation of any entity to the right of it, in the property, should not need to occur before Δ. An example of an invalid property is given in Table 3.2. In the invalid property, c and d need to be asserted before we assert b. Details of the rules are given in the language reference manual [6].

Table 3.2: Example property violating simple subset rules

Property	Valid (?)
AG ((a && XX(b)) → (c && X(d)))	No
AG ((p && XX(q)) → (XXX (r)))	Yes

Foundation Language (FL)

LTL style support all the temporal operators of LTL. Table 3.3 shows LTL operators and corresponding PSL synonyms [10]. Here, items 2, 4 and 5 have strong operators. As mentioned before, we shall not make distinction on types of temporal operator. In APL, all temporal operators shall be treated as *strong* [10].

Table 3.3: LTL operators in PSL

No.	LTL	PSL synonym
1	Gp	always p
2	Fp	eventually! p
3	Xp	next p
4	X! p	next ! p
5	p U q	p until ! q

In addition, PSL offers expressive classes of *'next'*, *'until'* and *'before'* operators. Table 3.4 gives two variations of *next* that are considered for APL. The reason for considering them is their ease of use and applicability in our property templates. For instance, if a property requires to reason about a state after 32 cycles, it is much simpler to write *next*[32] than writing X thirty two times. The complete set of LTL operators can be found in the PSL LRM [6].

Table 3.4: *next* operators considered for APL

Property	LTL Equivant	Description
next! p	Xp	p holds in the next cycle
next! [* n] p	X...X (n-times) p	p holds in the n^{th} next cycle

The other member of the Foundation Language is SERE style. SERE, *sequence extended regular expression*, was created so that users can write temporal properties as easily as they write *regular expressions*. In this style, the time sequence events are delimited by a *concatenation operator*. A set of expressive SERE operators are provided in this style to easily specify how long or when a boolean entity should be asserted. Most are very useful in simulation environment. Complete details of SERE operators and expressions can be found in PSL's LRM [6]. For now we only consider the *concatenation operator* and one of the *repetition operators*. Table 3.5 gives their details. More operators can be considered for APL if time permits. Given the underlying abstraction in MDG methodology, we will use the name *abstract SEREs* when they are used in APL.

Table 3.5: *SERE* operators considered for APL

Operator	Name	Example	Description
;	Concatenation operator	*req* ; *ack*	*ack* is asserted one cycle after *req* is asserted.
[* n]	Consecutive repetition operator	*p*[*3]; *q*	*p* is asserted for three consecutive cycles and then *q* is asserted in the fourth cycle.

Since clock is always implicit in MDG methodology, we do not consider any clock related expressions. The SERE expressions used in APL are thereby *unclocked.* Based on the strict set of algorithms we have, we shall not consider few other useful operators: suffix, fusion, disjunction, within, etc. Endpoint declarations and built-in functions are not applicable for the very same reason. Syntax and semantics of *foundation language* are given in the language reference manual of PSL [6].

Optional Branching Extension (OBE)

For branching-time temporal logic based formal verification, PSL offers *Optional Branching Extension* (OBE). It is based on CTL. Since our MDG model checker is based on first-order abstract CTL*, we support all OBE operators, with their applicability governed by their corresponding verification algorithms (See Figure 3.1). Restriction on *until* operator still applies based on simple subset rules of PSL.

3.1.3 Verification Layer

This layer consists of directives to the verification tool indicating what to do with the property in concern. There are five classes of directives: *assert*, *assume*, *restrict*, *cover* and *fairness.* In MDG model-checking algorithms all properties are asserted, and thus *assert* is the implicit directive. The fairness constraints can be employed by using fairness algorithms available in the L_{mdg} package.

3.1.4 Modeling Layer

This layer helps model behavior of design inputs and allows declaring and providing behavior to auxiliary signals and variables. In MDG model-checking methodology, the properties are written in first-order abstract CTL*, and the syntax is vastly different then its HDL flavor. In MDG-HDL, the logic gates are represented structurally using components. The written properties are parsed and embedded as additional circuit to the original model as checkers. The signals and components needed to represent these additional circuits are automatically generated.

3.2 L_{mdg} and its subset in APL

L_{mdg} is constructed based on first-order temporal logic. It is a subset of abstract CTL* [36]. The syntax and semantics of it have already been clearly defined and established [37]. Our goal is to improve the existing specification language, L_{mdg}, and its parser implementation. Therefore, we must consider the L_{mdg} language in its entirety for APL. Only a few lexical rules for operands are changed, and some operators are replaced by their synonyms. However, the semantics remain as defined. Let us revisit the syntax of L_{mdg}.

With a given description of an *abstract state machine* (ASM) and a set of ordinary variables (used as storage in properties), the formulae are divided into two categories: *state* formulae and *path* formulae.

State Formulae

1. $t_1 = t_2$ is a *state* formula, if t_1 is an ASM variable and t_2 is either an ASM variable, or a constant, or an ordinary variable.

2. If p, q are *state* formulae, then so are:

   (a) $!p$ (NOT p),

(b) p & q (p AND q),

(c) p | q (p OR q) and

(d) $p \rightarrow q$ (p IMPLIES q).

3. LET ($v = t$) in p is a *state* formula, if v is an ordinary (storage) variable, t is an ASM variable and p is a *state* formula.

4. If p is a *path* formula, then Ap (for all path) and Ep (there exist a path) are *state* formulae.

Path Formulae

1. Each *state* formula is a *path* formula.

2. If p, q are *path* formulae, then so are

   (a) !p (NOT p),

   (b) p & q (p AND q),

   (c) p | q (p OR q),

   (d) $p \rightarrow q$ (p IMPLIES q),

   (e) Xp (next p),

   (f) Gp (always p),

   (g) Fp (eventually p) and

   (h) pUq (p until q).

3. LET ($v = t$) in p is a *path* formula, if v is an ordinary (storage) variable, t is an ASM variable and p is a *path* formula.

Here, LET (v_1=t_1 & ... & ($v_n = t_n$) IN p is a shorthand for LET (v_1= t_1) IN (LET (v_2= t_2) ... (LET (v_n= t_n) IN p))...)

The proposed changes to incorporate into APL are following:

1. Replace comparison operator in the state formula 1 by a conventional one, as seen in SystemC [7]. t_2 in the formula can also be a function symbol. The syntax given in Backus-Naur Form (BNF) of L_{mdg} [35] supports it when it is prefixed by a *let-equation* in the property. The proposed improvement is to support functions as right terms in any comparison. The assignment operator, =, used in state formula 3 and path formula 3, remain the same in APL.

2. Replace two boolean operators (Table 3.6): logical AND and OR. Instead of using "&" for logical AND and | for logical OR, we use conventional operators [7] [5] given in Table 3.1.

Table 3.6: Operator Synonyms

Lmdg Operator	APL Operator	Description
|	||	Logical OR
&	&&	Logical AND
=	==	Comparison operator

In addition to the above, few L_{mdg} lexical rules are not present in APL. The syntax of L_{mdg} given in BNF imposes few restrictions to differentiate between ASM variable, ordinary variable, function name and symbolic constant. In APL, we shall only impose lexical restrictions that are inherent in MDG-HDL. Differentiating identifiers can be done by scanning the implementation of the design. Details of the scanning process are presented in Chapter 4. The numeric constants in APL do not support empty strings. These differences are presented in Table 3.7.

Table 3.7: L_{mdg} and APL Lexical Rules

Identifier	Lmdg	APL / MDG-HDL
ASM variable name	[a-bd-eg-uw-z][A-Za-z0-9_]*	[a-z][a-zA-Z0-9_]*
Ordinary (storage) variable name	[v][A-Za-z0-9_]*	[a-z][a-zA-Z0-9_]*
Function name	[f][A-Za-z0-9_]*	[a-z][a-zA-Z0-9_]*
Symbolic constant	[c][A-Za-z0-9_]*	[a-z][a-zA-Z0-9_]*
Integer constant	[0-9]*	[0-9]+

Excluding the above mentioned changes, the entire L_{mdg} is incorporated into APL.

3.3 Abstract Property Language(APL)

Abstract Property Language is based on existing L_{mdg} language. It is a subset of first-order abstract CTL* with added operators and expressions from PSL. In this section, we present its syntax and semantics.

3.3.1 Syntax of Abstract Property Language

Given a description of an abstract state machine (ASM) and a set of ordinary variables $\mathcal{V}$ to be used as storage in properties, the syntax of APL is given as follows:

1. $t_1 == t_2$ and $t_1 != t_2$ are compare formulae, where t_1 is an ASM variable; t_2 is an ASM variable, or a concrete value of t_1 or an ordinary (storage) variable. In not-equal formula, both operands are of concrete sort.

2. Every compare formula is a *state* formula.

3. If p, q are *state* formulae, then so are

   (a) $!p$

(b) p &&& q

(c) $p \parallel q$

4. If p is a *state* formula, and $v \in \mathcal{V}$, then following is a *state* formula

   - LET $(v = t)$ IN p

5. If p is a path formula, then Ap and Ep are state formulae.

6. Each state formula is a path formula.

7. If p, q are path formulae and n is a natural number greater than 0, then so are

   (a) $!p$

   (b) p &&& q

   (c) $p \parallel q$

   (d) Xp

   (e) Gp

   (f) Fp

   (g) pUq

   (h) $p;q$

   Equivalent to $(p$ &&& $Xq)$ in the absence of empty SERE [6].

   (i) $p[*n]$

   Represents n consecutive repetitions of p along the timeline. For example, $p[*3]$ is equivalent to writing $(p$ &&& Xp &&& $XXp)$

The last two formulae are from unclocked PSL SERE expressions.

8. If p is a path formulae, and $v \in \mathcal{V}$, then following is a path formula

   - LET $(v = t)$ IN p

Implication ($p \rightarrow q$) and equivalence ($p \leftrightarrow q$) formulae are derived from existing formulae. *next* p is equivalent to $X\ p$.

3.3.2 Semantics of Abstract Property Language

APL is constructed using syntax from L_{mdg} and PSL. The semantics of those syntax remain the same. The semantics of SERE expressions are presented here in different context, and they are named *abstract SERE*. The intuitive meaning remain as is. Given a description of an abstract state machine (ASM) and a set of ordinary variables $\mathcal{V}$ to be used as storage in properties, we define the following:

- All formulae are interpreted in the computation forest derived from a given ASM.
- total state is a set of variables representing state, inputs and outputs;
- $\pi_0 = (s_0, s_1, s_2, \dots)$ is a full path;
- $\pi_i = (s_i, s_{i+1}, s_{i+2}, \dots)$ is a suffix path;
- $Val_{s \cup \sigma}(t)$ denotes the value of the term t where value s is assigned to total state and value σ is assigned to ordinary variables; the assignments are Ψ-compatible.
- $s, \sigma \models p$ to denote that state formula p is true at total state s with σ assigned to ordinary variables;
- $\pi_i, \sigma \models p$ to denote that path formula p is true along path π_i with σ assigned to ordinary variables;

Given the above, we can now present relation $\models$ inductively as follows:

- $s,\sigma \models t_1 == t_2 \quad :\Leftrightarrow Val_{s \cup \sigma}(t_1) = Val_{s \cup \sigma}(t_2)$

- $s,\sigma \models t_1 \; != \; t_2 \quad :\Leftrightarrow Val_{s\cup\sigma}(t_1) \neq Val_{s\cup\sigma}(t_2)$; both operands are of concrete sort.
- $s,\sigma \models !p \quad :\Leftrightarrow$ not $s,\sigma \models p$
- $s,\sigma \models p \;\&\&\; q \quad :\Leftrightarrow s,\sigma \models p$ and $s,\sigma \models q$
- $s,\sigma \models p \parallel q \quad :\Leftrightarrow s,\sigma \models p$ or $s,\sigma \models q$
- $s,\sigma \models$ LET $(v{=}t)$ IN $p \quad :\Leftrightarrow s,\sigma' \models p$, where $\sigma' = (\sigma \setminus \{v, \sigma(v)\}) \cup \{v, Val_{s\cup\sigma}(t)\}$
- $s,\sigma \models \mathrm{A}p \quad :\Leftrightarrow \pi_i,\sigma \models p$ for every path $\pi_i = (s_i, s_{i+1}, s_{i+2}, \dots)$ in the computation forest.
- $s,\sigma \models \mathrm{E}p \quad :\Leftrightarrow \pi_i,\sigma \models p$ for some path $\pi_i{=}(s_i, s_{i+1}, s_{i+2}, \dots)$ in the computation forest.
- $\pi_i,\sigma \models p$, where p is a state formula $\quad :\Leftrightarrow s_i,\sigma \models p$
- $\pi_i,\sigma \models !p \quad :\Leftrightarrow$ not $\pi_i,\sigma \models p$
- $\pi_i,\sigma \models p \;\&\&\; q \quad :\Leftrightarrow \pi_i,\sigma \models p$ and $\pi_i,\sigma \models q$
- $\pi_i,\sigma \models p \parallel q \quad :\Leftrightarrow \pi_i,\sigma \models p$ or $\pi_i,\sigma \models q$
- $\pi_i,\sigma \models \mathrm{X}p \quad :\Leftrightarrow \pi_{i+1},\sigma \models p$
- $\pi_i,\sigma \models p;q \quad :\Leftrightarrow \pi_i,\sigma \models p$ and $\pi_{i+1},\sigma \models q$
- $\pi_i,\sigma \models p[\star n] \quad :\Leftrightarrow \forall_{i\leq j\leq i+n}\, \pi_j,\sigma \models p$
- $\pi_i,\sigma \models \mathrm{G}p \quad :\Leftrightarrow \forall_{j\geq i}\, \pi_j,\sigma \models p$
- $\pi_i,\sigma \models \mathrm{F}p \quad :\Leftrightarrow \exists_{j\geq i}\, \pi_j,\sigma \models p$
- $\pi_i,\sigma \models p\mathrm{U}q \quad :\Leftrightarrow \exists_{k\geq i}\, \pi_k,\sigma \models q$ and $\forall_{i\leq j<k}\, \pi_j,\sigma \models p$
- $\pi_i,\sigma \models$ LET $(v{=}t)$ IN $p \quad :\Leftrightarrow \pi_i,\sigma' \models p$, where $\sigma' = (\sigma \setminus \{v, \sigma(v)\}) \cup \{v, Val_{s_i\cup\sigma}(t)\}$

3.3.3 APL Language Restrictions

There are few restrictions that apply to the syntax of APL. These restrictions are described below:

- Based on the algorithms implemented in MDG Package(See Figure 3.1), the nesting of CTL* operators is restricted. Only *next* operator is free. Table 3.8 summarizes the property templates allowed in accordance with the MDG model checker. However, more templates are covered based on equivalent formulas [17]. The description of temporal logic operators can be found in section 2.1.

Table 3.8: Property templates

Algorithm	Template
A	A (*property*)
AG	AG (*property*)
AF	AF (*property*)
AU	A (*property_1*) U (*property_2*)
E	E (*property*)
EG	EG (*property*)
EF	EF (*property*)
EU	E (*property_1*) U (*property_2*)
AGAF	AG (*property_1* => F *property_2*))
AGAU	AG (*property_1* => (*property_2* U *property_3*))

- Restrictions on operands are given in Table 3.9. The description of the operators can be found in section 3.1.1.

Table 3.9: APL restrictions

Operator	Operand restriction
==	Left operand cannot be a function or an ordinary (storage) variable.
!=	Both operands must be concrete; left operand cannot be a function or an ordinary (storage) variable.
→	Left operand must not contain temporal operators.
↔	Neither side can contain temporal operators.
||	Left operand must not contain temporal operators.

- Lastly, SERE operators are not to be combined with LTL operators.

Chapter 4

Generating Composite Model

In this chapter, we present the details of our proposed tool that can process the original model of a design along with its specification written in APL and can create a composite model ready to be verified by the MDG model-checker. This composite model represents the *Design Under Verification* with some added circuitry [37]. The added circuitry has one boolean output for each property. These additional circuits are called *monitors* and the boolean outputs are called *flags*. In Chapter 3, available property templates were provided. Figure 4.1 shows a composite model, where a monitor circuit is added to the original design with its flag output.

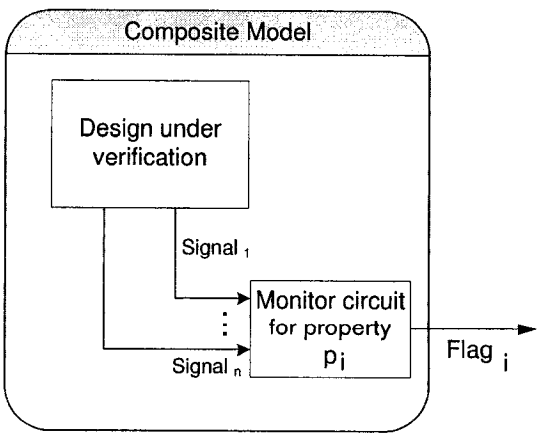


Figure 4.1: Composite model with flag output.

Before we present the details of the proposed tool, let us discuss the requirements (without precedence):

1. It must be a single tool capable of performing the task.
2. It should not impose any new syntax restriction on the model in MDG-HDL.
3. Property templates offered by the tool must concur with the algorithms allowed in MDG model-checker.
4. The specification will follow the APL language specification
5. The proposed tool should accept a single specification along with the MDG-HDL source files of the model.
6. It should process the specification to build monitor circuitry in MDG-HDL needed for model-checking process.
7. The added source code must be appended into copies of the original source files to construct the composite model.
8. Composite model source files must have convenient names.
9. It must report errors, indicating success or failure.
10. It must produce relevant condition files that are needed for different model-checking algorithms.

4.1 Tool Specification

In this section, we present a detailed specification of the proposed tool, called Abstract Verifier Translator (AVT). This tool acts as the new front-end of the Abstract Verifier. In Figure 4.2, a top-level view of the tool is illustrated.
A design implemented in MDG-HDL is composed of three files:

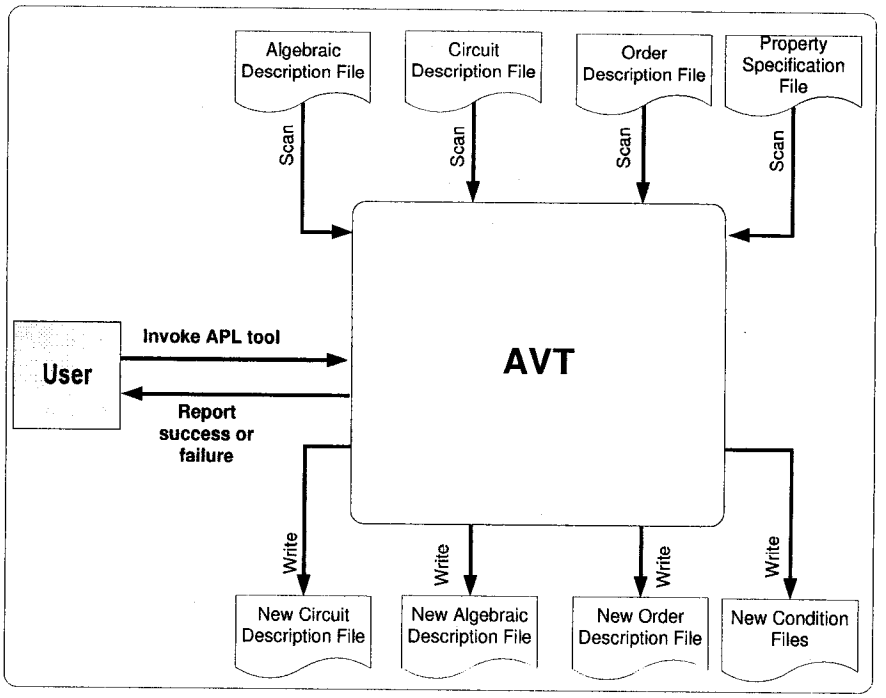


Figure 4.2: Top view of the generator tool.

1. Algebraic description file: Contains custom sorts (signal types), function declarations and re-writing rules giving the definitions of the functions.

2. Circuit description file: Contains detail description of the design.

3. Order description file: Contains order of the variables and functions to be used in construction of the graph, MDG.

The AVT is a command line tool accepting four files: algebraic description, order description, circuit description and a property specification file written in Abstract Property Language. User must write desired specification of the system in a separate text file according to APL specification. In the following sub-sections, we specify platform, file naming strategy, structure of the tool and description of the proposed new language.

4.1.1 Platform and interface

The tool needs to reside on a *Sun Solaris* system which has *Quintus Prolog* (version 3.2 or above) installed. Command-line interface requirement/specification is given below:

Usage: shell_prompt> *./**apl** Algebraic_file Circuit_file Order_file Specification_file*

4.1.2 Output filenames

To name a composite model file, the tool shall add '*new_*' at the beginning of the corresponding original filename. For example, if the original model filenames are: adder_alg.pl, adder_s.pl and adder_o.pl, then the composite model filenames will be: *new_adder_alg.pl*, *new_adder_s.pl* and *new_adder_o.pl*.

Table 4.1: Condition file naming strategy.

Property Template	Number of condition file(s)	Names
A, E	1	new_condition.pl
AG, EG	1	new_condition.pl
AF, EF	1	new_condition.pl
AU, EU	2	new_condition1.pl, new_condition2.pl
AGAF	2	new_condition1.pl, new_condition2.pl
AGAU	3	new_condition0.pl, new_condition1.pl, new_condition2.pl

In addition to the composite model, the tool must create one or more condition file(s). When performing model-checking in MDG, the condition files(s) are required along with the composite model files. The condition files shall be named according to the property templates (Table 4.1).

The tool will report to the user the filenames of the composite model and filename(s) the condition file(s). It is the user's responsibility to make a note of the filenames before invoking Prolog to run the MDG model-checker.

4.1.3 Processing files

The model files of a design are to be scanned by small scanners implemented in C/C++. Source files are copied to destination files as is. Destination files are modified or appended to create the composite model files. All structures of data are collected and stored to be used later by the tool.

Property files are to be parsed using a translator sub-module with the help of data collected on symbols. The lexical analyzer and the syntax analyzer shall be implemented using Flex [22] and Bison [21] respectively. We shall discuss them in the next sections where the Abstract Verifier Translator (AVT) is presented.

4.2 AVT Architecture

In this section, we present the design of AVT and describe how it generates the composite model. The tool accepts four files as parameters: three MDG-HDL source files representing the model and a specification file written in APL. It reports syntax and semantic errors found in the specification file. It informs the user whether the construction of the composite model succeeded or not. In case of a failure, it provides detail of the error.

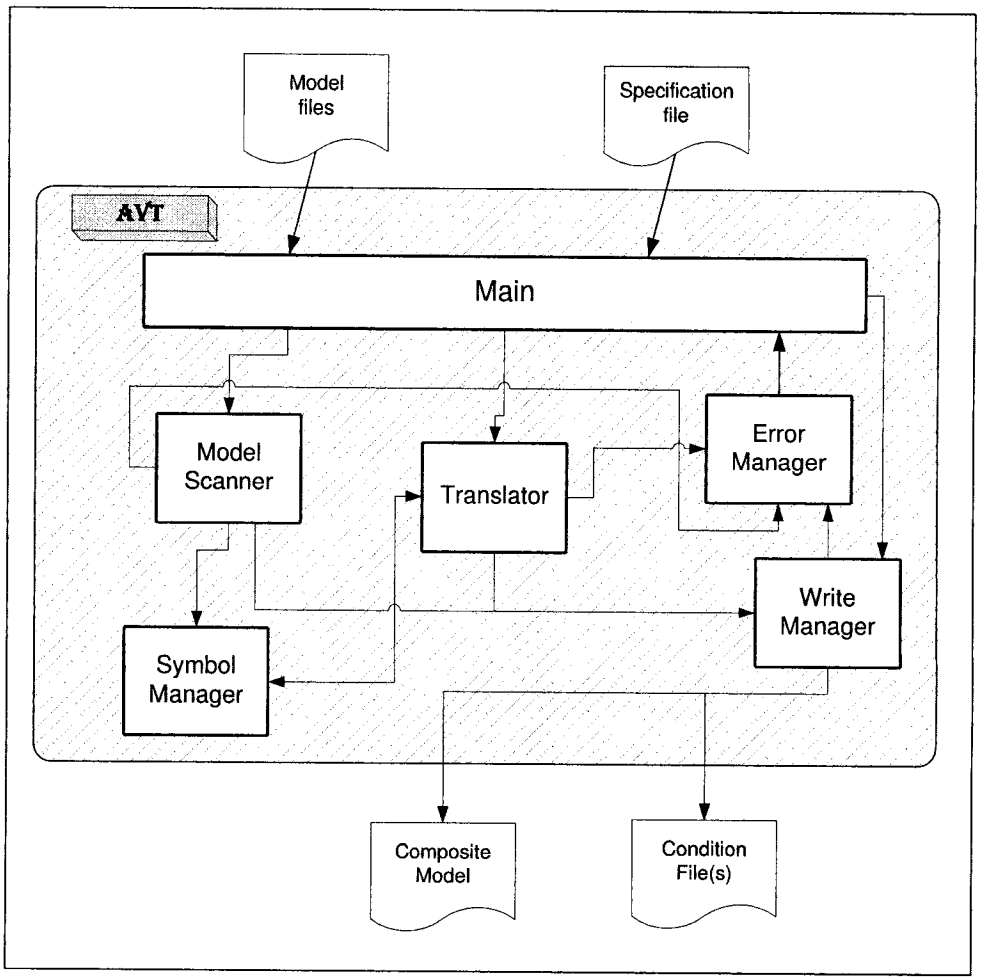


Figure 4.3: Structure of AVT.

The tool consists of six modules (Figure 4.3):

1. Main program: Responsible for dispatching tasks to other modules.

2. Model scanner: Responsible for scanning the original model files and providing collected data to Symbol Manager.

3. Symbol manager: Manages all symbol data.

4. Translator: Responsible for parsing the specification file, generating monitors with the help of Symbol manager.

5. Write manager: Responsible for writing output files.

6. Error manager: Responsible for providing error details.

Model scanner, Symbol manager and Translator are collection of sub-modules, performing smaller tasks.

4.2.1 Main Module

Main module (Figure 4.4) manages all the tasks required to generate the composite model and the condition file(s). It is invoked by a user with the model filenames and its specification filename as command line parameters. The model files represent the *Design Under Verification* in MDG-HDL.
It reports failure to the user if number of parameters is wrong and shows in console the proper usage. The dispatching of tasks occurs in the following sequence:

1. The task of scanning model files to gather symbol information is dispatched

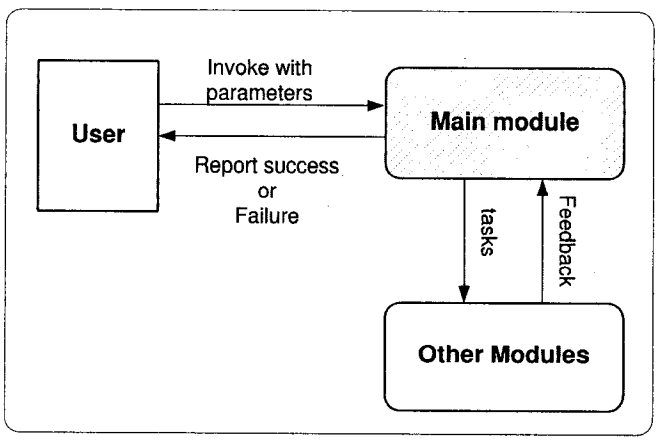


Figure 4.4: Main Module.

to the Model scanner with filenames of the original model files. The scanner reports success of failure back to the Main module.

2. It sends the specification filename to the Translator module. The module receives feedback on success and also gets the type of property templates used in specification.

3. It sends request to Write manager to write *state partition block* to the target circuit file, *end* of the target order file and to generate the condition file(s).

4. Upon success of previous task, it reports the completion to the user. Prints composite model filenames and the condition filename(s) on the console. It suggests to the user which template option to choose in MDG model-checker.

4.2.2 Model Scanner

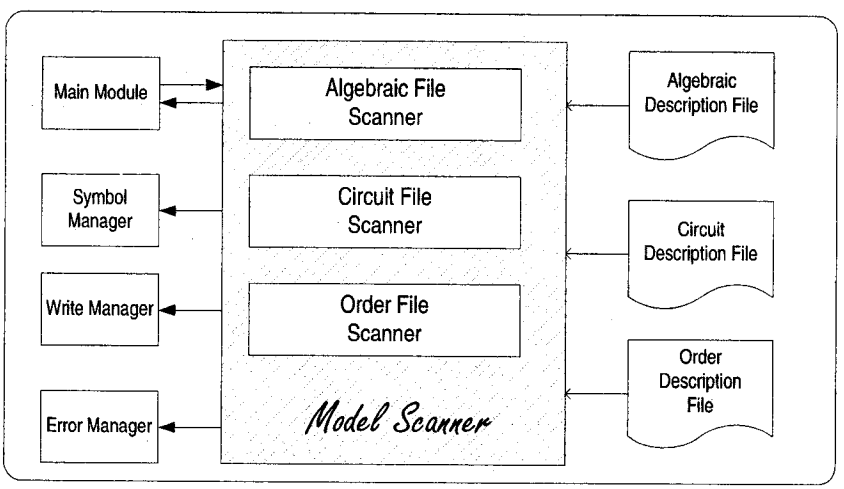


Figure 4.5: Model scanner.

The Model scanner is a collection of three small scanners, one for each model file (Figure 4.5). It receives the filenames of the model files from main module and then

assigns the tasks to appropriate scanner. If all three report *success*, it conveys the message to Main module, otherwise sends a failure message. These scanners do not look for syntax or semantic errors in the design. It assumes that user has compiled the design using MDG-Tools. In case of a file handle error, all scanners report to the Error manager. If the scan of the files shows a component, function or a signal name starting with *'apl_'*, an error is reported. It is reserved to be used in the monitor circuit(s).

Algebraic File Scanner

The scanner for the algebraic file looks for four types of declarations:

1. Abstract sort declarations.
2. Concrete sort declarations.
3. Generic sort declarations..
4. Function declarations.

It collects all symbol data (Table 4.2) and gives them to appropriate list managers in the Symbol manager. It requests Write manager to write a copy of the algebraic file as part of the composite model.

Table 4.2: Data collected from algebraic file.

Declaration	Example	Data Collected
Abstract sort	abs_sort (myAbs).	Type: myAbs Values: abstract
Concrete sort	conc_sort (bool, [1,0]).	Type: bool Values: 0,1
Generic sort	gen_const (rom,wordn).	Type: rom Values: abstract
Function	Function (my_fn, [wordn, wordn], bool).	Fn_name: my_fn Return type: bool Argument list: wordn, wordn

Circuit File Scanner

This scanner looks for signal declarations and collects signal names and their types. For example, if a signal is declared as *signal(mySig, bool)*, then it collects:

- Signal_name: mySig
- Type: bool

The scanner requests the write manager to write a copy of the file as target source without the block representing state partitions. The partition list is copied to a list structure and given to the symbol manager. Finally, the partition variables used in the monitor circuit are added to the list by the Translator module and written to the target source at the end.

Order File Scanner

The order file contains the order of signals and functions of a given design. This is used by the MDG-Tools to generate the MDG graph. The scanner scans the order structure and requests Write manager to write to the target order file without the *end* of the structure. Signals and functions of the monitor circuit are added to the list by the Translator module.

4.2.3 Symbol Manager

The Symbol manager is a collection of sub-modules that keep track of symbol names, their types and values. For functions, function name, its return type and the argument types are stored. Scope is not an issue in MDG-HDL. All variables and functions have global scope. It provides add-to-list services to Model scanner and several query services to the Translator module.

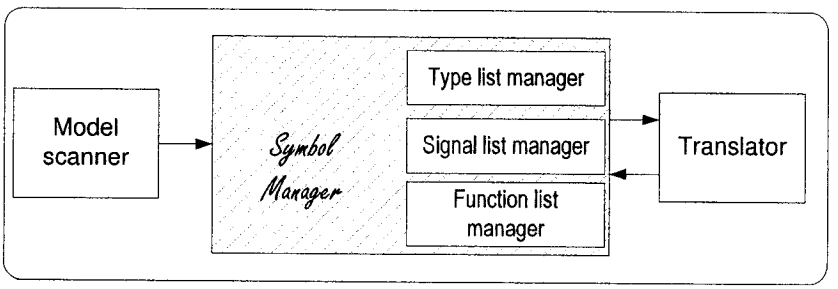


Figure 4.6: Symbol Manager

Type List Manager

The type list manager keeps track of all the custom sorts. All abstract sorts have a string value *wordn* indicating that they are abstracts and do not have any value. For the concrete sorts each type is saved with its possible values.

- Examples: Suppose if we have the following in the algebraic file:

  1. conc_sort(bool, [0,1]).
  2. conc_sort(word2, [0,1,2,3]).
  3. conc_sort(state, [s0,s1,s2]).

  then the corresponding collected data shall be:

  1. type: bool
     values: 0, 1
  2. type: word2
     values: 0, 1, 2, 3
  3. type: state
     values: s0, s1, s2

Signal List Manager

The signal list manager contains all the signal names along with their types.

- Examples: Suppose we have the following in the circuit file:

  1. signal(reset,bool).
  2. signal(myData,word2).
  3. signal(myStateVar, state).

  then the corresponding collected data shall be:

  1. signal: reset
     type: bool
  2. signal: myData
     type: word2
  3. signal: myStateVar
     type: state

Function List Manager

The function list manager stores all the function names along with their definition attributes. Definition attributes are argument sorts (types) and the target sort (return type).

- Examples: Suppose if we have the following in the algebraic file:

  1. function(eq,[wordn,wordn],bool).
  2. function(mul,[wordn,wordn],wordn).

  then the corresponding collected data shall be:

  1. function: eq
     argument_sort: wordn, wordn
     target_sort: bool

2. function: mul

   argument_sort: wordn, wordn

   target_sort: wordn

4.2.4 Translator Module

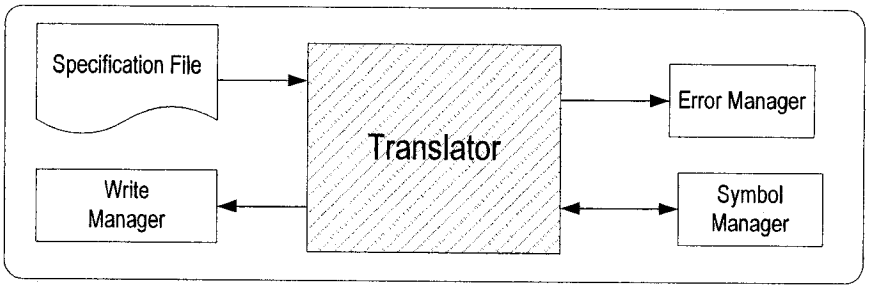


Figure 4.7: Top view of the Translator module.

The Translator module (Figure 4.7) compiles the specification file and generates, for each property, a monitor circuit with a flag output. Our source language is Abstract Property Language (APL) and the target language is MDG-HDL. It reports syntax and semantic errors to the Error Manager. It sends queries to the Symbol Manager anytime symbol data is required to generate a circuit or to simply check semantics. Since this module is the major component of the AVT, we discuss it in details in the next section. The following sub-sections discuss the other modules of the design.

4.2.5 Write Manager

The job of the write manager is simply to write all composite files and the condition file(s). It initially receives the request from the Main Module to write duplicate of the algebraic file, specification file without the next-state partition, and the order file without the closing brackets of the order structure. All subsequent requests come from the Code Generator. For each MDG-HDL component written, it writes

the signals, next state variables and the signals to the order file. The next state variables are also added to the next-state partition list. When all components have been written, it generates the order structure, the next-state partition list and the condition files. The condition files are generated according to Table 4.3.

Table 4.3: Flag output according to property templates.

Property Template	**Condition File(s)**	**Flag signals**
A, E	new_condition.pl	flag
AG, EG	new_condition.pl	flag
AF, EF	new_condition.pl	flag
AU, EU	new_condition1.pl	flag1
	new_condition2.pl	flag2
AGAF	new_condition1.pl	flag1
	new_condition2.pl	flag2
AGAU	new_condition0.pl	flag0
	new_condition1.pl	flag1
	new_condition2.pl	flag2

4.2.6 Error Manager

The job of Error Manager is to communicate to the user the occurrence of an error. It keeps a list of errors with identifications. Based on the reported error number, it prints a message to the user with indication on how to fix it. The AVT is not fault tolerant. At the presence of an error, it simply prints the error message and informs the Main Module to quit application. General error problems are following:

- File pointer returns NULL.
- Request for memory returns NULL. System is out of memory.

- User does not provide valid parameters for AVT. This error comes from the Main Module.
- Disk quota errors, etc.

The main tasks the error manager is responsible for are following:

- Convey all syntax errors reported by the Syntax Analyzer.
- Convey all semantic errors reported by the Context Handler.

4.3 Translator

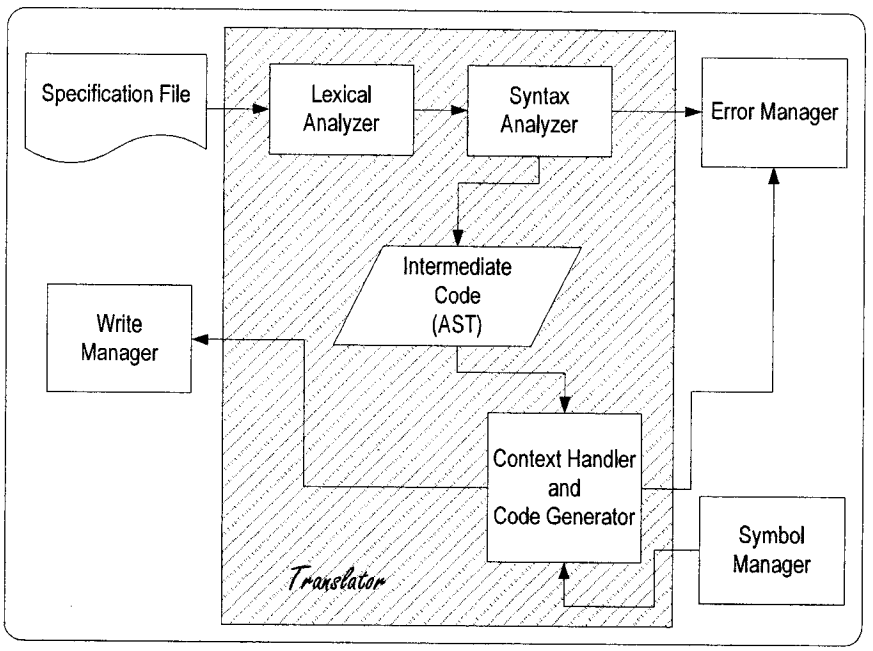


Figure 4.8: Translator.

This module is invoked by the main program with the file name of the specification file. For the translation, it has APL as the *source language* and MDG-HDL as the

target language. The process of generating the monitor circuit (Figure 4.1) from the specification has the following steps:

1. Lexical Analyzer tokenizes the specification text based on lexical specification of the *source language*, APL.

2. Syntax Analyzer creates annotated abstract syntax tree (AST) based on the grammar of APL.

3. Simple traversal algorithm is used to handle context and generate the target code at the same time. Whenever it finds a component to be built, it consults the Symbol Manager to check semantics. It reports semantic errors to the Error Manager. If no error exists for the component in concern, it generates MDG-HDL component code and its associated signals. The components are written to the target files by the Write Manager.

In the following sub-sections, we discuss each element of the translator module.

4.3.1 Lexical Analyzer

The lexical analyzer reads the input file character by character. It tokenizes the input based on matched patterns. It provides the current token to the syntax analyzer (Figure 4.9), when a request is made. Lexer for translator was automatically constructed using Flex [22].

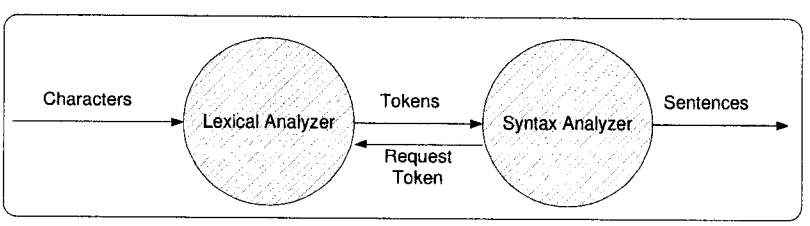


Figure 4.9: Syntax flow.

In our tool, the tokens are: identifiers, numeric value, temporal and path operators, boolean operators, delimiters, etc. In the following subsections, specification of the lexer is provided.

Definition of White Space

There is no restriction on using white space anywhere in the source. They are used only to separate tokens. White space is defined by following characters:

1. Space
2. New line character
3. Carriage return
4. Tab

Case Sensitivity

Case insensitivity can lead to very bad programming practices. For example, using same variable with different cases is not a good programming practice and may create lot of confusion. Since *Quintus Prolog* is case sensitive, the obvious choice is to make the lexer case sensitive as well.

Token Length

There is no restriction imposed on token length. This specification may change depending on the implementation choices.

Error Reporting

The lexical analyzer does not report any errors. Any character that does not comprise a token is simply passed on to the calling application (Syntax Analyzer).

Table 4.4: Token List.

	TOKEN	Description
Identifier & Numeric Value	NUMBER	[0-9]+
	IDENTIFIER	[a-z][a-zA-Z0-9_]*
Path and temporal operators	A	"A"
	E	"E"
	AG	"AG"
	AF	"AF"
	U	"U"
	F	"F"
	X	"X"
	SEMICOLON	";"
	NEXT	"next"
LET equation operators	LET	"LET"
	IN	"IN"
Boolean Operators	IMPLICATION	"->"
	IMPLICATION2	"=>"
	EQUIVALENCE	"<->"
	OR	"||"
	AND	"&&"
	NOT	"!"
	EQUAL	"=="
	NOT_EQUAL	"!="
Assignment Operator	ASSIGNMENT	"="
Delimiters	OB	"["
	CB	"]"
	OP	"("
	CP	")"
	OC	"{"
	CC	"}"
	COMMA	","
	ASTERISK	"*"
	DOT	"."
Ignored white spaces		" \t\n\r"

List of Tokens

Here, the list of tokens is specified (Table 4.4). In accordance with MDG-HDL, an identifier must begin with a small letter. A numeric value consists of only digits and may not start with a unary symbol. Since we have followed lexical rules of MDG-HDL, L_{mdg} restrictions are not present in APL. As a result, users no longer

need to modify their original model in order to verify them.

4.3.2 Syntax Analyzer

The syntax analyzer requests tokens from the lexer, and based on specified grammar, it recognizes valid syntax. The traditional way to represent a parsed sentence is to construct an abstract syntax tree (Figure 4.10). In AVT, these sentences are expressions that construct a single statement. We use Bison [21] to automatically construct our Syntax Analyzer. The context-free-grammar of Abstract Property Language is presented in this chapter.

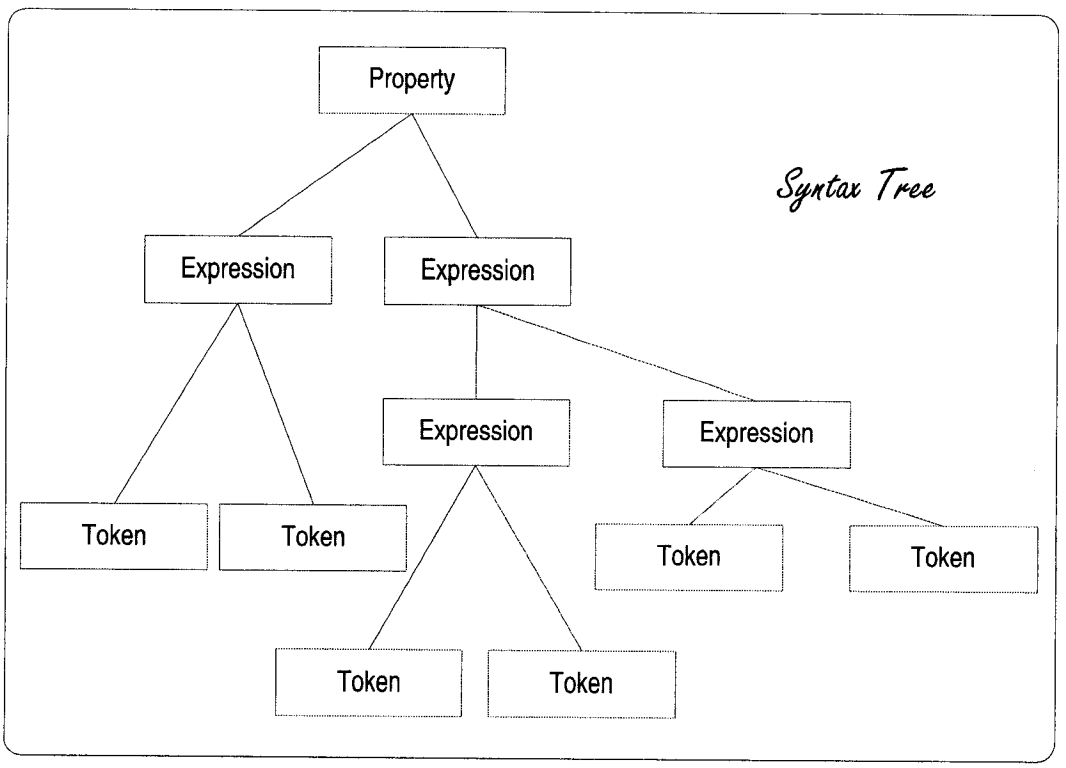


Figure 4.10: Syntax Tree

Grammar used for AST generation:

Here, we specify the grammar of APL in BNF. The tokens are from Table 4.4. Delimiters are given as is to improve readability. The grammar used as syntax specification in Bison is given in following sub-sections.

□ Property Templates:

```
apl_property: A (expression).
  | E  (expression).
  | AG (expression).
  | EG (expression).
  | AF (expression).
  | EF (expression).
  | A  (expression) U (expression).
  | E  (expression) U (expression).
  | AG ((expression) IMPLICATION2 (F(expression))).
  | AG ((expression) IMPLICATION2 ((expression) U (expression))).
  ;
```

□ Expressions:

```
expression: boolean_expression
          |   sere_expression
          |   next_expression
          |   let_expression
          ;
```

□ Boolean Expressions:

```
boolean_expression: compare_equation
                  |  expression AND expression
                  |  expression OR expression
```

```
                | expression IMPLICATION expression
                | NOT (expression)
                | IDENTIFIER
                | NOT IDENTIFIER
                | expression EQUIVALENCE expression
                ;
compare_equation    :   IDENTIFIER EQUAL right_term
                |   IDENTIFIER NOT_EQUAL right_term
                ;
right_term: IDENTIFIER
          |   NUMBER
          |   function
          ;
```

□ Function Expression:

```
function:   IDENTIFIER (param_list)
        ;
param_list: param
          |   param_list , param
          ;
param   :   ID
      |   Function
      ;
```

□ Temporal Expressions:

```
next_expression:    X expression
          |   NEXT expression
          |   NEXT [NUMBER] (expression)
          ;
```

```
sere_expression:    {boolean_expression}
          |   {repeated_sere}
          |   {sere_expression ; sere_expression}
          ;
repeated_sere   :   sere_expression [* NUMBER]
              ;
```

□ LET Expressions:

```
let_expression: LET(assign_equation) IN (expression)
          ;
assign_equation:    IDENTIFIER ASSIGNMENT IDENTIFIER
          |   assign_equation AND assign_equation
          ;
```

4.3.3 Annotated AST Generation:

Bison [21] provides a very easy way to construct abstract syntax tree from syntax rules. Every symbol in Bison parser has a *value*. This *value* gives the actual string or numeric valuation of the symbol. Each syntax rule can be associated with an action, C code segment. Whenever a sentence is matched, the corresponding code segment is executed. We use these features to build our syntax tree. Root of each tree is the expression in the property template. For example, say we have a property of the following template: *AG(expression)*. In this case, the *expression* will be the root node of the AST.

In order to make context handling and code generation a matter of writing a simple traversal algorithm, we must build the nodes with basic building blocks of MDG-HDL in mind. In this way, we do not need to re-write the AST but simply annotate it with pertinent information required for later tasks. In MDG-HDL, the basic constructs that can be used for generating monitors are:

1. Abstract function symbol used in specification are represented as it is defined in algebraic specification.

2. Table components representing comparisons that contain concrete signals.

3. Cross-term function symbol representing comparisons that contain abstract signals.

4. Components representing boolean operations.

5. Register component for preserving values.

The automatically generated syntax analyzer reports error if syntax rules are not matched. Detail errors are reported along with line and word number. In the following subsections we discuss how to build AST for different class of expressions.

AST for function expressions

An abstract function symbol can only appear on the right hand side of a compare expression. It has a right child representing the parameter list. The parameter list containing more than one parameter is separated by comma nodes. Figure 4.11 shows tree constructions of some function expressions. Each signal or variable parameter is a node with null children. Constructing AST for nested function call is also shown in Figure 4.11. If a parameter is an ordinary (storage) variable coming from a LET expression, then it is replaced by the preserved value of the variable. The nodes are annotated with pertinent information. For example, a simple integer value per node is used to indicate node type. This allows the context handler and the code generator to distinguish nodes.

AST for compare expressions

The most basic building block in APL is a compare equation, where the left hand side must be a signal. The signal can be concrete or abstract. The right hand side

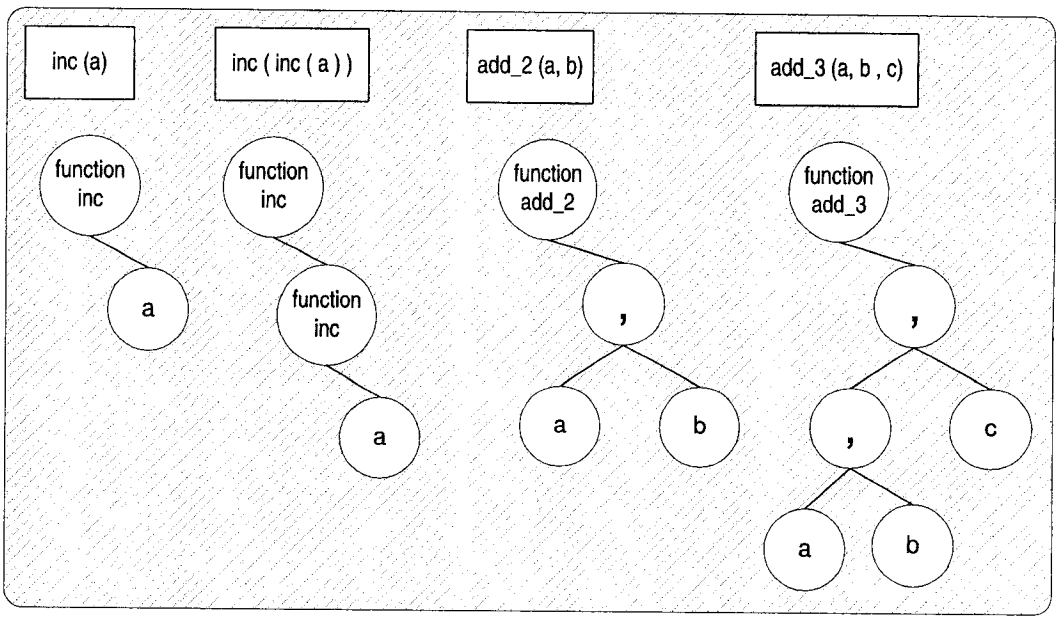


Figure 4.11: AST for function expressions.

can be a signal, a value or a function. The signal on the left hand side is a leaf node. The right hand side is also a leaf node if it is a signal or a value. If it is a function, then the right pointer of the comparison node points to the function's sub-tree. Node types are annotated to the nodes. The equal operator type is also annotated based on the type of signal present on the left operand. This allows the context handler and code generator to comprehend if the comparison is between concretes or abstracts. Figure 4.12 gives some example AST's of compare expressions.

AST for boolean expressions

In the grammar of APL, we use some expressions that do not have corresponding component in MDG-HDL to represent them. Table 4.5 gives their interpretations. Here, *my_bool_sig* is a signal of concrete sort *bool* having values 0 or 1; p and r are boolean expressions; q can be any expression. For the first two expressions in the table, we treat them as compare expression and create nodes accordingly.

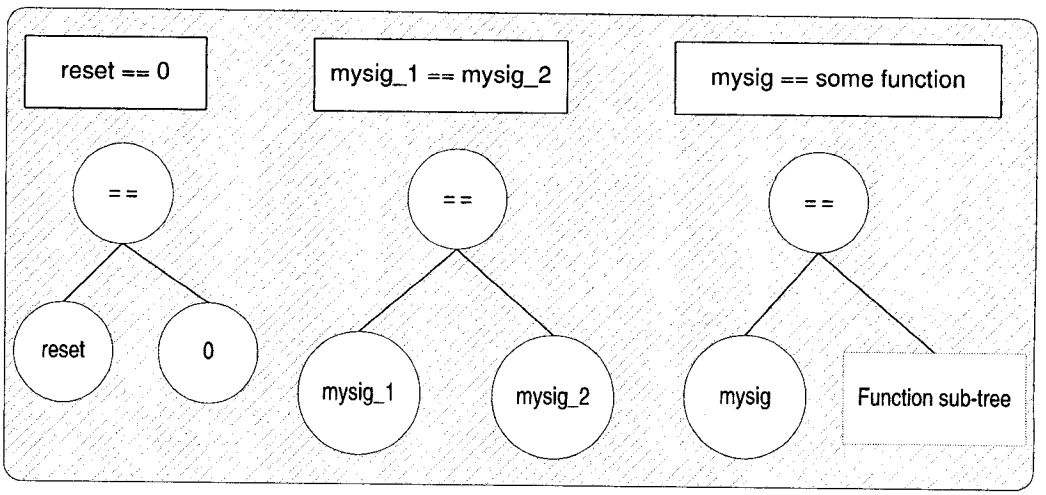


Figure 4.12: AST for compare expressions.

Table 4.5: Boolean expression mapping.

Expression	Mapping
my_bool_sig	my_bool_sig = = 1
! my_bool_sig	my_bool_sig = = 0
p -> *q*	!*p* || *q*
p <-> *r*	(!*p* || *q*) && (!*q* || *p*)

In MDG-HDL, we have NOT, OR and AND components that we can use, thus we only use these three operators in AST construction. Figure 4.13 gives basic AST's constructed from boolean expressions. Nodes are annotated with node types indicating the type of operators used. The NOT operator has the operand as its right child and the left child pointer is assigned to NULL.

AST for temporal expressions

Due to restriction in property templates, only X operators are *free* to be used without a path quantifier. The other temporal operators are taken care of by the

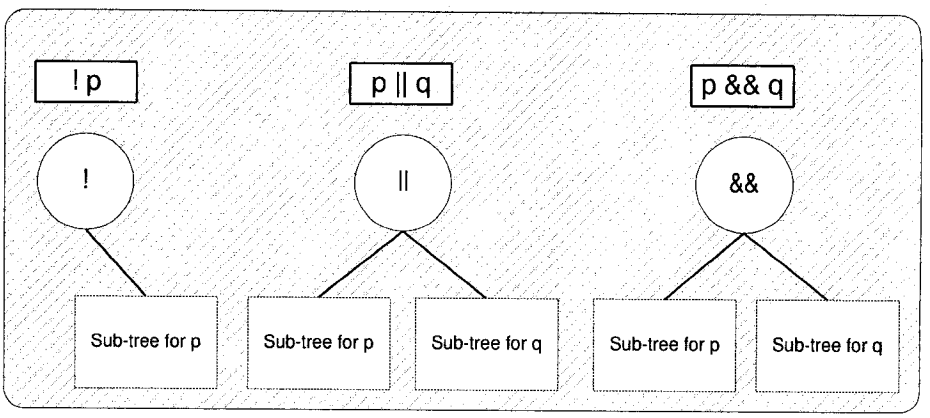


Figure 4.13: AST for boolean expressions.

algorithms associated with the specification templates. To preserve a value in MDG-HDL for multiple transitions, we can use the *reg* component. The $next[Number]$ and the abstract $SERE$ expressions can be built by using the X operator. Table 4.6 shows their mappings, where p, and q are expressions.

The technique we use to denote X nodes is that instead of creating multiple nodes for multiple X's, we simply create one X node with an attribute specifying the repetition times. X nodes are annotated with this numeric value. This saves memory and also makes traversal algorithms faster. Figure 4.14 shows example AST constructed for temporal expressions, where p is an expression.

Table 4.6: Temporal expression mapping.

Expression	Mapping using X operator	Intuitive meaning
next [3] p	XXX p	p is true after 3rd transition.
p ; q	p && (X q)	p is true in present transition and q is true in the next transition step.
p [*2]	p && (X p)	p is true for two transition steps including present cycle.

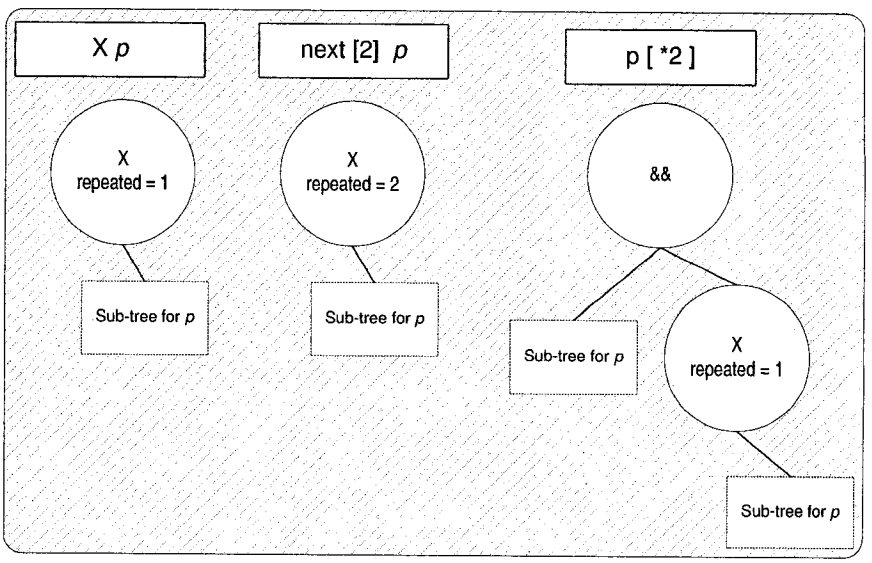


Figure 4.14: AST for temporal expression examples.

AST for LET expression

The LET expressions are constructed with a top node having IN operator. The left sub-tree contains all the variable assignment(s) and the right sub-tree contains the expression where the variable(s) are used. LET expression has the following format:

- LET (p) IN (q),
  where where p represents the variable assignment(s) and q represents an expression, usually a temporal expression.

Figure 4.15 shows an AST constructed from an example. Here, the two AND operators are annotated with different node types: one indicating a logical AND and the other is a delimiter to separate the variable assignments. A list of all assignments are maintained.

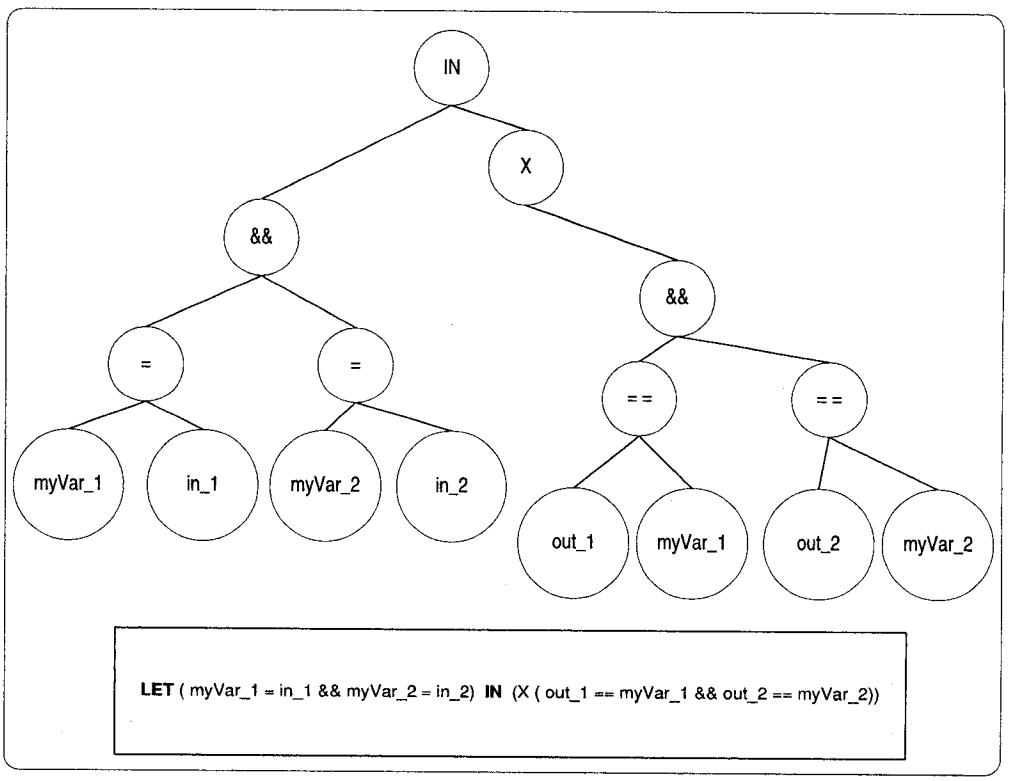


Figure 4.15: AST for LET expression.

4.3.4 Context Handling and Code Generation

The procedure to build additional circuit for MDG based model-checking was introduced by Ying Xu et al. [37]. Soundness of the verification procedures has been established [36]. To follow this established procedure in generating additional circuits for model-checker, we must further annotate the AST. Following sub-section discusses the further annotation procedure, and subsequent sub-sections present context handling and code generation algorithms.

Further annotation of AST

To make search algorithms faster, we convert the AST into a binary search tree (BST). This makes search algorithms having the complexity of O($log_2 n$) instead

of O(n). This is done by using a depth-first *in-order* traversal of the AST and assigning order number to the nodes sequentially. The algorithm that takes the root node pointer as argument is given below, where *counter* is initialized to *zero*:

```
Make_bst (node)
  if node.left  not null then Make_bst(node.left)
  node.order_number = counter
  increment counter by 1
  if node.right not null then Make_bst(node.right)
```

In order to deal with temporal operators in specification, we must use registers to reserve values of predicates at transition steps. In our AST, we only have one type of temporal operator, X. Determining how many registers to put on outputs of boolean operators, we must keep track of the X operators. To do this, we put a counter on node structure. The default value of this counter attribute is set to *zero*. The annotation can be done using the *post-order* traversal algorithm given below, where the module is called with a pointer to the root node of the AST. Here, the $Xrepetition$ is a node attribute that tells us how many X's are there and $Xcounter$ counts the number of X's in the corresponding node's sub-tree.

```
Assign_Xcounter(node)
 if node.left  not null then Assign_Xcounter(node.left)
 if node.right not null then Assign_Xcounter(node.right)
 if node.type is a function or a NOT
  then node.Xcounter = node.right.Xcounter
 else if node.type is an X
    then node.Xcounter = node.right.Xcounter + node.X repetition
 else
    node.Xcounter = maximum (node.left.Xcounter, node.right.Xcounter)
```

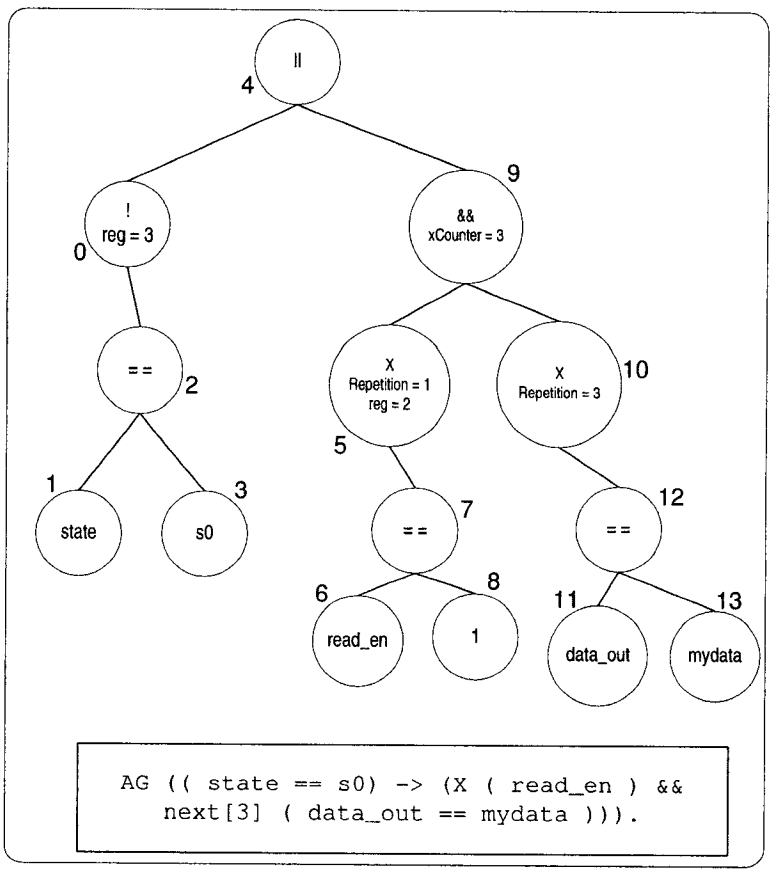


Figure 4.16: Annotated AST.

To complete the annotation procedure in this phase, we perform one more traversal to assign registers to the to-be-generated MDG-HDL components. It is a simple pre-order traversal where *reg* is a node attribute that is initialized to *zero*:

```
Assign_registers (node)
  if node.type is possible MDG-HDL component
   then node.left.reg = node.right.Xcounter - node.left.Xcounter
  if node.left  not null then Assign_registers(node.left)
  if node.right not null then Assign_registers(node.right)
```

This algorithm depends on the restriction that the use of temporal operators allows time to move from left-to-right fashion through the property. Figure 4.16 gives an example, where the order numbers are given adjacent to the nodes.

Context Handling

Context checking is performed every time a function or a compare equation is encountered during code generation.

- Functions:

  1. The parameter types are checked by communicating with the Symbol Manager.
  2. If the node type of the parameter yields a variable, a search algorithm is used to find which type of signal was last assigned to it. The type is checked with the argument type to preserve semantic correctness.

- Compare equations:

  1. The signal type of left hand operand is requested from the Symbol Manager.
  2. The right hand side could be a variable, a signal, a value or a function. The node type tells us if it is a variable or a function.
  3. If a query to the Symbol Manager yields a signal then the signal type is checked to match the signal type of the left operand.
  4. If the query tells us that it is a value, the value list of the signal type is checked to see if the value provided is within the range of possible values of the signal in left operand.
  5. If the right operand is a function, then the function's return type (target sort) is matched with the type of left operand signal.

6. If the right operand is a variable, then a search algorithm is used to find which type of signal was last assigned to it. The type is checked with the left operand signal type.

In all cases, semantic mismatches are reported as semantic errors to the Error Manager.

Code Generation

Building the AST with MDG-HDL in mind allows us to use simple traversal algorithms to perform both context handling and code generation. We traverse trough the AST in depth-first *post-order* fashion and look for components to build. Every time we encounter such an operator that corresponds to a component in MDG-HDL, we build the circuitry and replace the sub-tree of that node with the output of the component. This is done by storing an attribute called *output string* representing the output of the sub-tree. This string is further modified if registers are needed for that operator. Throughout the process, requests are made to the Write manager to write additional signals and circuits to the composite model files. Next state variables are added to the next-state partition table. List of signals, next state variables and functions used are added to the target order description file. In our depth-first *post-order* traversal, every time we visit a node, we replace the sub-tree with its output based on following:

1. If the node is a signal or a value of a signal, then the output of that node is string representation of that signal or the value.

2. If the node is a variable, then its output is its assigned value going through possible registers. Register components are built and the output signal of the final register replaces the sub-tree.

3. If the node is a parameter and it is a variable, then the output is replaced in the same manner as given in item 2.

4. If the node is a function, then a function component is built. If registers are needed then output signal of the final register replaces the sub-tree. Otherwise, the output signal of the function circuit replaces the sub-tree.

5. If the node is a comparator, then we evaluate the type of the comparison:

   (a) If the comparison is between a concrete signal and its value or another concrete signal, a table component in MDG-HDL is generated and the sub-tree is replaced with the component's output signal.

   (b) If the comparison is between two abstract signals, then a predefined *eq* function symbol is used to generate required circuit. The output signal of the *eq* component replaces the sub-tree.

   (c) If the comparison is between a signal and a function, then a table or an abstract function symbol is used based on the type of signal. The inputs of the components are the signal and the output of the function sub-tree.

   In all cases, if registers are required, then register components are built and the output of the final register in HDL replaces the sub-tree.

6. If the node is of type X or IN, we simply replace the sub-tree with the output signal of the right sub-tree.

7. If the node is a *comma* in between parameters, we simply concatenate its children output strings and the resultant string replaces the sub-tree.

8. If the node is a boolean operator, we simply construct the corresponding circuitry in MDG-HDL and output signal of the component replaces the sub-tree. If registers are required, then register components are built and the output of the final register in HDL replaces the sub-tree.

Figure 4.17 depicts the circuitry generated from AST given in Figure 4.16. Here, we assume: *state* is a *concrete* type with $s0$ as one of its value, *mydata* and

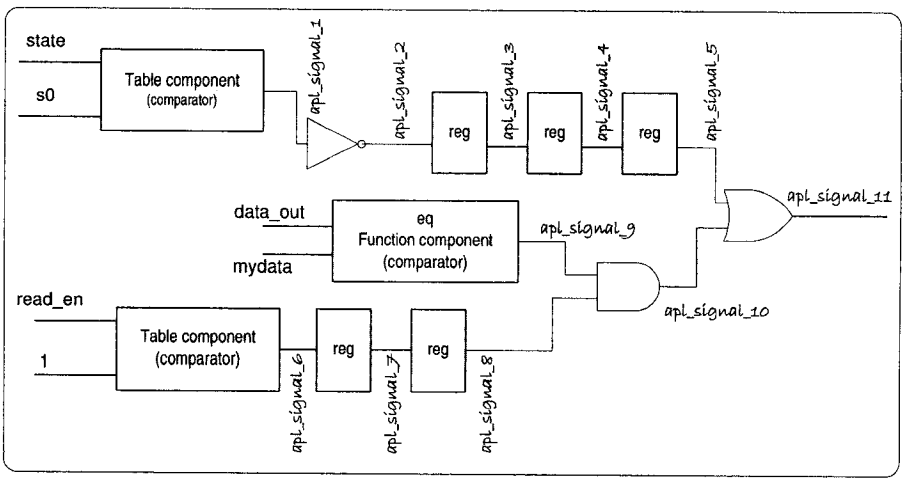


Figure 4.17: Circuit representing property.

data_out are of abstract sort, *read_en* is of concrete type with *bool* sort. In this example the sub-tree outputs are assigned as follows:

1. The comparator ($state == s0$) sub-tree is replaced with *apl_signal_1*
2. The NOT sub-tree is replaced with *apl_signal_5*
3. The comparator ($read\_in == 1$) sub-tree is replaced with *apl_signal_8*
4. The comparator ($data\_out == mydata$) is replaced with *apl_signal_9*
5. The AND sub-tree is replaced with *apl_signal_10*
6. The top OR is represented by *apl_signal_11*

For all the register components, a control signal is also generated. The control is assigned the value *true* by default.

4.3.5 Flag Circuit Generation

In order to complete a monitor circuit generation for a property, we must produce a flag circuit for it (Figure 4.19). This is in accordance with the MDG model-checking

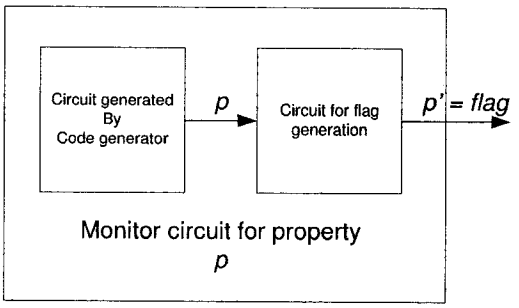


Figure 4.18: Completion of Monitor Circuit.

algorithms developed by Ying Xu et. al [36]. The flag signal output is used by the respective algorithm to verify the property in concern. Architecture of a flag circuit is dictated by the corresponding property template. The rules are given below:

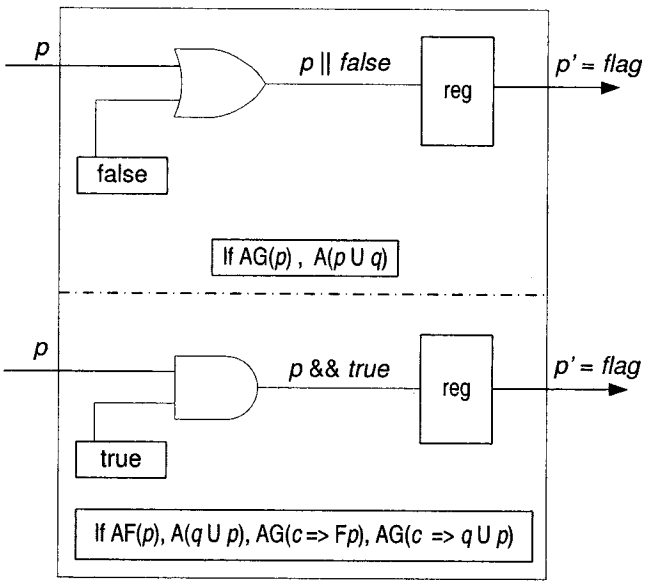


Figure 4.19: The Flag Circuit

1. If the property template used is one of AFp, AqUp, AG($c \Rightarrow$ Fp) or AG($c \Rightarrow$ qUp), then the flag output:

   $p' = true$ && p. Here, *true* is a constant signal; an AND component is used to produce the flag output; the inputs for the component are *true* and p.

2. If the property template used is either AGp or ApUq, then flag output:

   $p' = false \parallel p$. Here, $false$ is a constant signal; an OR component is used to produce the flag output; the inputs for the component are $false$ and p.

It is a requirement of the MDG model-checking algorithm that this flag be an ASM state variable and must be stored in a register in order for the algorithms to produce correct result (Figure 4.19). Specification and correctness of this method has already been established [37].

Chapter 5

Experimental Results

In this chapter, we present results of performance analysis of the AVT compared to the L_{mdg}-Tools, analysis of the generated code and results of model-checking a sub-block of Look-Aside Interface design [8]. In the first two experiments, we do not consider memory consumption, because in this case resource requirements of text-processing compared to the actual model-checking process is negligible. The experiments were run on a *Sun Enterprise* E3500, which is a multi-processor system (6 × UltraSPARC-IIi 400MHz) with 6GB of main memory.

5.1 Performance Comparison

In this section, we compare the performance of AVT and L_{mdg}-Tools. We analyze the required time for generating a monitor circuit given a specification. Thus, the design we choose for this experiment is not critical. For the results given in Table 5.1, we use a design that has no functionality in real world but provides an interface for us to perform rigorous experiments. For our specification, we choose a AG(p) property that has many comparators of both concrete and abstract sorts, comparing stored values in variables with signal values.

- AG(LET (v_1 && v_2 ... && v_n) IN ($c_0 \rightarrow (X(c_1 \wedge c_2 \ldots \wedge c_n))$)).

$v_1, v_2, \ldots, v_n$ are variable assignments and

$c_1, c_2, \ldots, c_n$ are comparisons of stored values in variables with some signal values. c_0 simply compares a signal with its value. It serves as the trigger condition.

In each test property, we insert all comparators after the presence of the *next* operator and all variable assignments before it. In this way, each increase in the number of Xs in the test property results in creation of n registers in the monitor circuit, one register per stored signal value. As a result, we get better impact on the generated monitor circuit when we increase the number of Xs. Ideally the *Until* operator has better impact than the *next* operator. However, its restrictive usage in available property templates makes it ineffective in our experiments.

The inspection of CPU usage and execution time indicates that the AVT is less processor intensive but produces the same result in less time. We calculate the average of Execution times. For AVT it is 0.254, and for L_{mdg}-Tools it is 0.498. The relative performance on average is:

Table 5.1: Performance of AVT compared to L_{mdg}-Tools

Number of 'X' operator	AVT Execution Time (E_1) in seconds	AVT Performance ($P_1=1/E_1$)	AVT CPU Usage (%)	Lmdg Execution Time (E_2) in seconds	Lmdg Performance ($P_2=1/E_2$)	Lmdg CPU Usage (%)
1	0.09	11.11	22.0	0.19	5.26	63.0
5	0.10	10.00	24.0	0.21	4.76	88.0
10	0.12	8.33	36.0	0.32	3.13	84.0
50	0.16	6.25	28.5	0.37	2.70	76.4
100	0.17	5.88	52.9	0.39	2.56	81.6
150	0.20	5.00	74.3	0.42	2.38	91.4
200	0.21	4.76	65.6	0.47	2.13	86.7
250	0.24	4.17	77.5	0.48	2.08	85.0
300	0.33	3.03	76.4	0.53	1.89	78.0
350	0.35	2.86	80.9	0.60	1.67	85.5
400	0.41	2.44	63.1	0.73	1.37	76.5
450	0.44	2.27	63.6	0.81	1.23	82.6
500	0.49	2.04	80.9	0.95	1.05	88.5

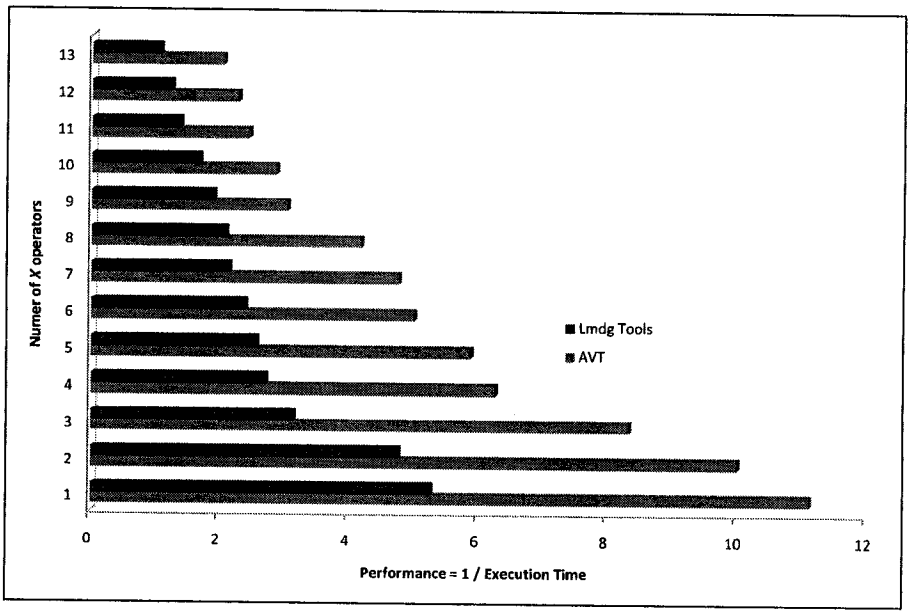


Figure 5.1: Performance Comparison.

- Relative Performance = Avg(E_2) / Avg(E_1) $\cong$ 1.95

From this result, we can say that on average the AVT performs about 1.95 times faster than the L_{mdg}-Tools. A graph (Figure 5.1) is presented in this section, illustrating the performance difference between the two.

The performance gain in APL over L_{mdg} tools can be contributed to three key factors:

- Instead of using an ad-hoc approach we have utilized a conventional method of translator design.

- By annotating our abstract syntax tree via *in-order traversal*, we have converted our AST to binary search tree (BST). This makes our search algorithm having much better complexity of Θ ($log_2 n$) over Θ (n).

- We have replaced the use of multiple X nodes by a single X node having an attribute specifying its repetition. This reduces the size of the AST (memory) and also reduces traversal time.

5.2 Area Evaluation of the Generated Circuit

In this section, we evaluate the area of the generated circuit in MDG-HDL given a specification. In this experiment, we keep in mind that the flag circuit requires one extra logic gate and one extra register (Section 4.3.5). The design used in these experiments is the same as the one used in the previous section.

We start by using 'next' or 'X' operator in one place in the specification and increment the number of comparisons to observe their affects on the area. The results are given in Table 5.2. The collected data confirms our expectation that number of gates and signals will increase linearly with the number of comparisons we have in our specification formula.

We now use 'next' operator in one place in the specification and increment its repetition to observe its affect on the area in composite model. The results are given in the Table 5.2. The collected data confirms the expectation that number

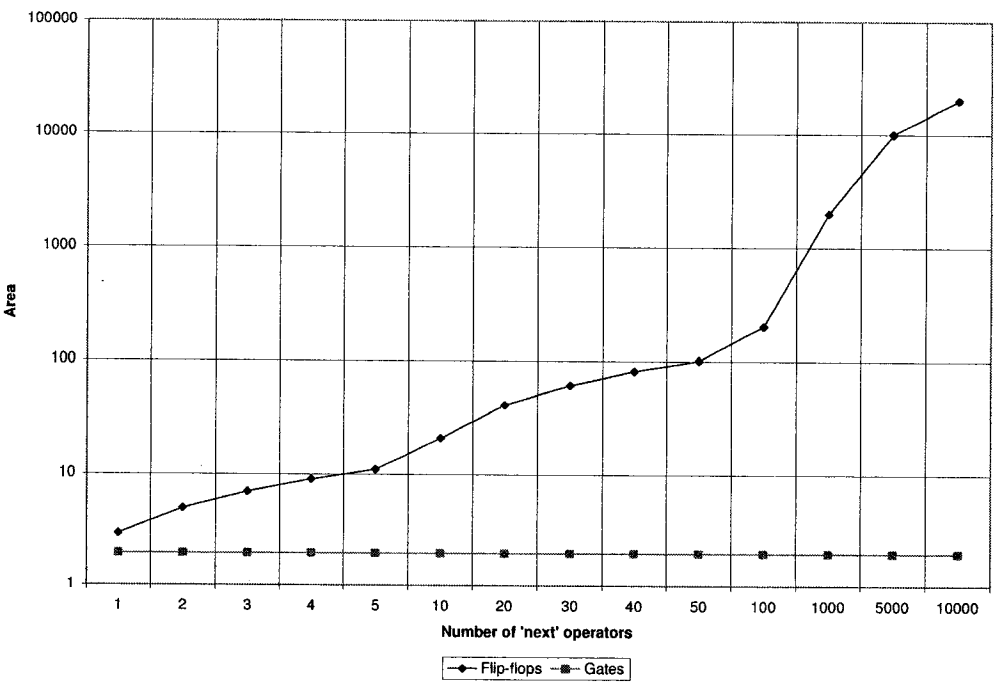


Figure 5.2: Effect of 'next' on area (logarithmic scale on Y-axis).

Table 5.2: Area Evaluation.

comparisons	Abstract	Concrete	bool operator	Number of 'next'	signals	Flip-flops	Gates	tables	functions	Execution Time (seconds)
2	1	1	1	0	8	1	2	1	1	0.07
4	2	2	3	0	12	1	4	2	2	0.08
6	3	3	5	0	16	1	6	3	3	0.10
8	4	4	7	0	20	1	8	4	4	0.12
10	5	5	9	0	24	1	10	5	5	0.12
2	1	1	1	1	10	3	2	1	1	0.07
2	1	1	1	2	12	5	2	1	1	0.07
2	1	1	1	3	14	7	2	1	1	0.09
2	1	1	1	4	16	9	2	1	1	0.11
2	1	1	1	5	18	11	2	1	1	0.11
2	2	0	1	1	9	3	3	2	0	0.01
2	2	0	1	3	13	7	3	2	0	0.09
2	2	0	1	10	27	21	3	2	0	0.09
2	2	0	1	100	207	201	3	2	0	0.09
2	2	0	1	1000	2007	2001	3	2	0	0.41
2	2	0	1	10000	20007	20001	3	2	0	27.53
4	4	0	3	1	11	5	5	4	0	0.01
4	4	0	3	3	19	13	5	4	0	0.07
4	4	0	3	10	47	41	5	4	0	0.09
4	4	0	3	100	407	401	5	4	0	0.06
4	4	0	3	1000	4007	4001	5	4	0	0.18
4	4	0	3	10000	40007	40001	5	4	0	36.24
2	1	1	1	1	8	3	2	1	1	0.02
2	1	1	1	2	10	5	2	1	1	0.07
2	1	1	1	3	12	7	2	1	1	0.08
2	1	1	1	4	14	9	2	1	1	0.09
2	1	1	1	5	16	11	2	1	1	0.09
2	1	1	1	10	26	21	2	1	1	0.09
2	1	1	1	20	46	41	2	1	1	0.10
2	1	1	1	30	66	61	2	1	1	0.11
2	1	1	1	40	86	81	2	1	1	0.11
2	1	1	1	50	106	101	2	1	1	0.12
2	1	1	1	100	206	201	2	1	1	0.12
2	1	1	1	1000	2006	2001	2	1	1	0.50
2	1	1	1	5000	10006	10001	2	1	1	12.24
2	1	1	1	10000	20006	20001	2	1	1	46.62

of registers (flip-flops) will increase linearly with the number of 'next' operators. We present the fact that even at an unusually large number of 'next' operators, we still get the desired generated circuit to be produced within a minute. However, at 500,000 'next' operators, the tool did not produce a result. We do not envision a specification containing that numbers of 'next' operators. Figure 5.2 gives the effect of 'next' operators on area in logarithmic scaled graph. From these experiments, we conclude the following:

- Number of 'next' operators dictate the number of registers required. In terms of the representation in AST, each increase of number of 'next' operator in one path, increments the number of registers by one for each path that does not go through that operator.
- Number of comparisons involving concrete signals will dictate the number of tables in the generated circuit.
- Number of comparisons involving abstract signals will dictate the number of abstract function symbols.
- An extra gate and an extra register are always needed for the flag circuit.
- Number of signals and next state variables to be declared are dictated by all of the above.

5.3 Application: Verification of the LA-1 Interface

In this section, we verify a part of a data transfer protocol, called Look-Aside Interface (LA1) [20]. This design was originally implemented by the Network Processor Forum. It is illustrated in Figure 5.3. The design was implemented and verified by Li et al. [13] using MDG based model-checking methodology.

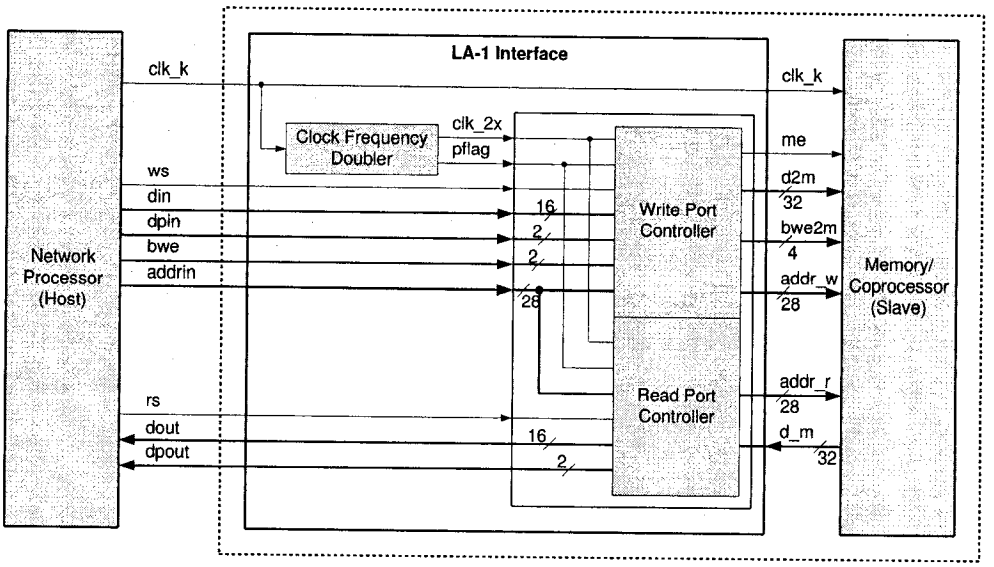


Figure 5.3: Look-Aside Interface (LA1).

We collect Read-port specifications of this design, process them using AVT and present the experimental results of MDG based model-checking in this section. The MDG-HDL model for the Read Port is shown in Figure 5.4, where signals and components are represented as follows:

1. input signals *clk_2x*, *pflag* and *rs* are of concrete type (bool),
2. input signals *d_m* and *addrin* are of abstract sort (wordn),
3. output signals *dpout1* and *dpout0* are of concrete type (bool),
4. output signals *dout* and *addr_r* are of abstract sort (wordn) and
5. components *msw*, *lsw*, *parity1*, *parity2*, *parity3* and *parity4* are abstract function symbols.

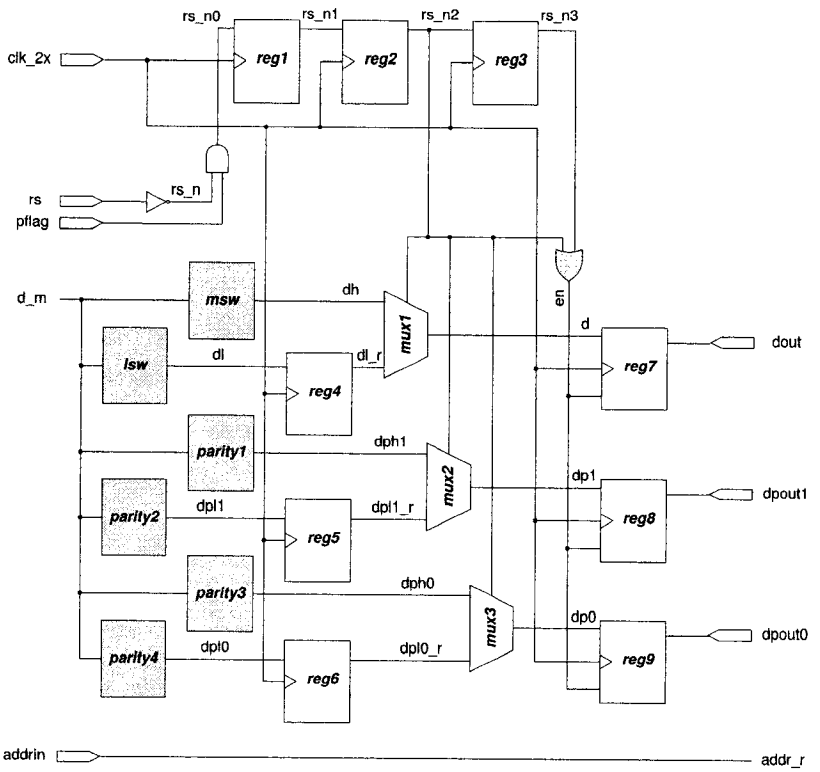


Figure 5.4: Read-port representation in MDG-HDL.

5.3.1 Specification

The specifications are re-written in APL and are presented in Table 5.3. The names of functions and variables in APL have no restrictions. The LET equations do not have to start at the beginning of the specification. Boolean signals do not need to be in compare equation format.

5.3.2 MDG Model-checking Results

We take the specifications and model of Read-port module and process it with AVT one at a time. The tool gives us the composite model with its condition file based on the specification. We then proceed to run the MDG model-checker to verify

Table 5.3: Read-port specification in APL.

No.	Read-port specification in APL
1	AG ((pflag && !rs) → (XX (LET (stored_data=d_m) IN (X (dout == msw(stored_data)))).
2	AG ((pflag && ! rs) → (XX (LET (stored_data =d_m) IN (X (dout == lsw(stored_data))))).
3	AG ((pflag & !rs) → (X (LET (stored_data =d_m) IN (X (dpout1 == parity1(stored_data))))).
4	AG ((pflag & !rs) → (XX (LET (stored_data =d_m) IN (XX (dpout1 == parity3(stored_data))))).
5	AG ((pflag & !rs) → (XX (LET (stored_data =d_m) IN (XX (dpout1 == parity3(stored_data))))).

the specification. Table 5.4 gives runtime, memory usage and the number of MDG nodes used for verifying each specification. It also gives time (in seconds) it takes for AVT to generate the composite models.

Table 5.4: Verification results for Read-port specifications.

Spec. No.	AVT runtime (seconds)	MDG Runtime (seconds)	Memory (MB)	MDG nodes
1	0.11	18.04	20.19	53837
2	0.09	19.46	23.89	54352
3	0.07	14.24	10.46	49638
4	0.12	55.82	64.45	173087
5	0.14	62.30	92.61	178321

Chapter 6

Conclusion and Future Work

MDG based model-checking presents an improvement over traditional BDD-based model-checking by introducing a technique of representing the design in higher abstraction and simplifying the data path operations. As a result, users can effectively overcome the state explosion problem.

The model-checking algorithms in verification package of Abstract Verifier require that a user feed the model-checker a composite model that includes a monitor circuit with one or more flags as output. The flags are monitored by the verification algorithms to inform the user if a given specification is satisfied by its model of the design. In case the specification does not hold *true*, the model-checker produces a counter example. Using this counter example, the user can investigate the source of the error in design. In this thesis, our focus was on the front-end of the method where we process the specification text and generate the composite circuit. In Chapter 1, we have outlined the issues that are evident in L_{mdg} set of tools. We have explained the shortcomings of the present representation and implementation, and our objectives on how to resolve these issues were stated as well.

The specification of the new language, called APL, was presented in Chapter 3. In APL, we have eliminated all the lexical restrictions that were present in the

L_{mdg} specification language. Consequently, users do not need to modify their original model in order to get the verification-ready composite model. With APL, users can simply use the original model, write their specification in APL and process both with the AVT tool. While replacing L_{mdg} with APL, we have added standardized operators and introduced new operators borrowed from PSL to improve expressiveness over L_{mdg}. We have provided the formal definition of this language and provided its formal semantics following L_{mdg} semantics. In Chapter 4, we have presented details on how we have accomplished the goal of generating verification-ready composite models and their condition files, concurring with the requirements of the MDG model-checking algorithms. We have designed and implemented AVT as a single independent application, capable of handling first-order temporal logic. As a result, there is no need for employing multiple tools to process the specification (properties). Also, users are not required to create a separate file specifying the functions used in the properties. By following a recognized process of designing a translator, we have created an efficient tool that produces the result in almost half the time compared to the existing tool. Detail results of the performance analysis and generated-circuit area analysis were presented in Chapter 5. In the same chapter, we have also presented results obtained by model-checking the Read-port module of the Look-Aside Interface.

As future work, we provide following research directions:

1. Investigate and design a single user-interface representing both front-end and the back-end of the MDG model-checking process. If implemented, a user can write a property, create composite model and perform model-checking by interacting with the interface.

2. Investigate model-checking algorithms of MDG-Tools and implement new property templates allowing the use of 'until' operator without restrictions. In the present templates, this operator is only allowed in predefined positions. If

the operator is unrestricted, we can incorporate more SERE-like expressions where a signal is asserted unknown number (zero or more) of times.

3. Investigate and develop a translator that can process a design represented in SystemVerilog or other traditional HDL's and can automatically generate its corresponding representation in MDG-HDL.

4. Investigate and improve counter-example generation process of MDG model-checking.

Appendix A

A.1 Read-port of Look-Aside Interface in MDG-HDL

In this section, the source code of the original model is provided. The model is represented in MDG-HDL in the following three files:

1. Algebraic description file: read_port_2x_alg.pl
2. Circuit description file: read_port_2x_s.pl
3. Order description file: read_port_2x_o.pl

A.1.1 Algebraic Description

```
%-----------------------------------------
% File: read_port_2x_alg.pl
%-----------------------------------------
% Sort specification
abs_sort(mi_sort,wordn).
% Functions
function(fmsw,[wordn],wordn). function(flsw,[wordn],wordn).
function(fparity1,[wordn],bool). function(fparity2,[wordn],bool).
```

function(fparity3,[wordn],bool). function(fparity4,[wordn],bool).

A.1.2 Circuit Description

```
%-----------------------------------------
% File: read_port_2x_s.pl
%-----------------------------------------
%========== Inputs and Outputs =====
%-- Inputs --
signal(pflag,bool).
signal(pflag_n, bool).
signal(rs,bool).
signal(d_m,wordn).
signal(addrin,wordn).
%-- Internal signals --
signal(rs_n,bool).
signal(rs_n0,bool).
signal(rs_n1,bool).
signal(rs_n2,bool).
signal(rs_n3,bool).
signal(en,bool).
signal(dh,wordn).
signal(dl,wordn).
signal(dl_r,wordn).
signal(dph1,bool).
signal(dpl1,bool).
signal(dpl1_r,bool).
signal(dph0,bool).
signal(dpl0,bool).
```

```
signal(dpl0_r,bool).
signal(d,wordn).
signal(dp1,bool).
signal(dp0,bool).
%— Outputs —
signal(dout,wordn).
signal(dpout1,bool).
signal(dpout0,bool).
signal(addr_r,wordn).
%— Components —
signal(const_one,bool).
component(const1,constant_signal(value(1), signal(const_one))).
component(not1,not(input(pflag),output(pflag_n))).
component(reg0,reg(input(pflag_n),output(pflag))).
component(not2,not(input(rs),output(rs_n))).
component(and1,and(input(rs_n,pflag),output(rs_n0))).
component(reg1,reg(control(const_one),input(rs_n0),output(rs_n1))).
component(reg2,reg(control(const_one),input(rs_n1),output(rs_n2))).
component(reg3,reg(control(const_one),input(rs_n2),output(rs_n3))).
component(or1,or(input(rs_n2,rs_n3),output(en))).
component(msw,transform(inputs([d_m]),function(fmsw),output(dh))).
component(lsw,transform(inputs([d_m]),function(flsw),output(dl))).
component(parity1,transform(inputs([d_m]),function(fparity1),output(dph1))).
component(parity2,transform(inputs([d_m]),function(fparity2),output(dpl1))).
component(parity3,transform(inputs([d_m]),function(fparity3),output(dph0))).
component(parity4,transform(inputs([d_m]),function(fparity4),output(dpl0))).
component(reg4,reg(control(const_one),input(dl),output(dl_r))).
component(reg5,reg(control(const_one),input(dpl1),output(dpl1_r))).
component(reg6,reg(control(const_one),input(dpl0),output(dpl0_r))).
component(mux1,mux(sel(rs_n2),inputs([(1,dh),(0,dl_r)]),output(d))).
```

```
component(mux2,mux(sel(rs_n2),inputs([(1,dph1),(0,dpl1_r)]),output(dp1))).
component(mux3,mux(sel(rs_n2),inputs([(1,dph0),(0,dpl0_r)]),output(dp0))).
component(reg7,reg(control(en),input(d),output(dout))).
component(reg8,reg(control(en),input(dp1),output(dpout1))).
component(reg9,reg(control(en),input(dp0),output(dpout0))).
component(fork1,fork(input(addrin),output(addr_r))).
%— Partitions —
outputs([]).
output_partition([[[]]]).
next_state_partition([
[[pflag_n]], [[n_rs_n1]], [[n_rs_n2]], [[n_rs_n3]], [[n_dl_r]], [[n_dpl1_r]], [[n_dpl0_r]], [[n_dout]],
[[n_dpout1]],[[n_dpout0]]]).
%— State variable, next state variable mapping—
st_nxst(pflag,pflag_n).
st_nxst(rs_n1,n_rs_n1).
st_nxst(rs_n2,n_rs_n2).
st_nxst(rs_n3,n_rs_n3).
st_nxst(dl_r,n_dl_r).
st_nxst(dpl1_r,n_dpl1_r).
st_nxst(dpl0_r,n_dpl0_r).
st_nxst(dout,n_dout).
st_nxst(dpout1,n_dpout1).
st_nxst(dpout0,n_dpout0).
%— Partition strategy—
par_strategy(default, default).
```

A.1.3 Order Description

```
%————————————————————————————————————————
```

% File: read_port_2x_o.pl

%----------

order_main([

const_one, pflag, pflag_n, rs, d_m, addrin, rs_n, rs_n0, rs_n1, rs_n2, rs_n3, en, dh, dl, dl_r, dph1,dl1, dpl1_r, dph0,dpl0, dpl0_r, d, dp1, dp0, dout, dpout1, dpout0, addr_r, n_rs_n1, n_rs_n2, n_rs_n3, n_dl_r, n_dpl1_r, n_dpl0_r, n_dout, n_dpout1, n_dpout0, fmsw, flsw, fparity1, fparity2, fparity3, fparity4]).

A.2 Generated monitor circuit for LA-1 in MDG-HDL

In this section, we give an example specification and then provide source code of the generated monitor circuit in MDG-HDL. The new verification-ready model is comprised of the following three files:

1. Circuit description file: new_read_port_2x_s.pl
2. Order description file: new_read_port_2x_o.pl
3. Algebraic description file: new_read_port_2x_alg.pl

A.2.1 Example Specification

```
AG( (pflag && !rs)-> ( XX( LET (v1=d_m) IN ( X( dout==fmsw(v1)))))).
```

Here, only one condition file is needed. It is named `new_condition.pl`

A.2.2 Monitor Circuit in Circuit Description File

```
%-------------------------
% File: new_read_port_2x_s.pl
%-------------------------
...
% *** Added Signals and Components for Model-checker =
% *** NOTE: Original next_state_partition is modified and added at the end.
% — Control signal for all reg components.
signal(apl_control_signal,bool).
component(apl_control_signal_comp, constant_signal(value(1),signal(apl_control_signal))).
% — Boolean constant component —
signal(apl_added_signal_1,bool).
component(apl_comp_1_constant, constant_signal(value(0),signal(apl_added_signal_1))).
signal(apl_added_signal_2,bool).
signal(n_apl_added_signal_2,bool).
component(apl_reg_comp_0, reg(control(apl_control_signal),
input(apl_added_signal_1),output(apl_added_signal_2))).
st_nxst(apl_added_signal_2, n_apl_added_signal_2).
init_val(apl_added_signal_2, 1).
signal(apl_added_signal_3,bool).
signal(n_apl_added_signal_3,bool).
component(apl_reg_comp_1, reg(control(apl_control_signal),
input(apl_added_signal_2),output(apl_added_signal_3))).
st_nxst(apl_added_signal_3, n_apl_added_signal_3).
init_val(apl_added_signal_3, 1).
signal(apl_added_signal_4,bool).
signal(n_apl_added_signal_4,bool).
component(apl_reg_comp_2, reg(control(apl_control_signal),
input(apl_added_signal_3),output(apl_added_signal_4))).
st_nxst(apl_added_signal_4, n_apl_added_signal_4).
```

```
init_val(apl_added_signal_4, 1).
% — Concrete comparator component —
signal(apl_added_signal_5, bool).
component(apl_comp_2_concrete_comparator, table([[pflag,apl_added_signal_5], [1, 1] |0])).
% — Concrete comparator component —
signal(apl_added_signal_6, bool).
component(apl_comp_3_concrete_comparator, table([[rs,apl_added_signal_6], [0, 1] |0])).
% — AND gate component —
signal(apl_added_signal_7,bool).
component(apl_comp_4_AND,
and(input(apl_added_signal_5,apl_added_signal_6),output(apl_added_signal_7))).
% — NOT gate component —
signal(apl_added_signal_8,bool).
component(apl_comp_5_NOT,
not(input(apl_added_signal_7),output(apl_added_signal_8))).
signal(apl_added_signal_9,bool).
signal(n_apl_added_signal_9,bool).
component(apl_reg_comp_3, reg(control(apl_control_signal),
input(apl_added_signal_8),output(apl_added_signal_9))).
st_nxst(apl_added_signal_9, n_apl_added_signal_9).
init_val(apl_added_signal_9, 1).
signal(apl_added_signal_10,bool).
signal(n_apl_added_signal_10,bool).
component(apl_reg_comp_4, reg(control(apl_control_signal),
input(apl_added_signal_9),output(apl_added_signal_10))).
st_nxst(apl_added_signal_10, n_apl_added_signal_10).
init_val(apl_added_signal_10, 1).
signal(apl_added_signal_11,bool).
signal(n_apl_added_signal_11,bool).
```

```
component(apl_reg_comp_5, reg(control(apl_control_signal),
input(apl_added_signal_10),output(apl_added_signal_11))).
st_nxst(apl_added_signal_11, n_apl_added_signal_11).
init_val(apl_added_signal_11, 1).
% — Variable component —
signal(apl_added_signal_12,wordn).
signal(n_apl_added_signal_12,wordn).
component(apl_reg_comp_6, reg(control(apl_control_signal),
input(d_m),output(apl_added_signal_12))).
st_nxst(apl_added_signal_12, n_apl_added_signal_12).
% — Function component —
signal(apl_added_signal_13,wordn).
component(apl_comp_function_1, transform(inputs([apl_added_signal_12]),
function(fmsw), output(apl_added_signal_13))).
% — Abstract comparator component —
signal(apl_added_signal_14,bool).
component(apl_comp_function_2, transform(inputs([dout,apl_added_signal_13]),
function(absComp), output(apl_added_signal_14))).
% — OR gate component —
signal(apl_added_signal_15,bool).
component(apl_comp_7_OR,
or(input(apl_added_signal_11,apl_added_signal_14),output(apl_added_signal_15))).
% — OR gate component —
signal(apl_added_signal_16,bool).
component(apl_comp_8_OR,
or(input(apl_added_signal_4,apl_added_signal_15),output(apl_added_signal_16))).
% — Property flag component —
signal(flag,bool).
signal(n_flag,bool).
```

```
component(ag_property_comp, reg(control(apl_control_signal),
input(apl_added_signal_16),output(flag))).
st_nxst(flag, n_flag).
init_val(flag, 1).
% — Modified Next State Partition —
next_state_partition([
[[pflag_n]],
[[n_apl_added_signal_2]],
[[n_apl_added_signal_3]],
[[n_apl_added_signal_4]],
[[n_apl_added_signal_9]],
[[n_apl_added_signal_10]],
[[n_apl_added_signal_11]],
[[n_apl_added_signal_12]],
[[n_flag]],
[[n_rs_n1]],
[[n_rs_n2]],
[[n_rs_n3]],
[[n_dl_r]],
[[n_dpl1_r]],
[[n_dpl0_r]],
[[n_dout]],
[[n_dpout1]],
[[n_dpout0]]
]).
```

A.3 Modified Order Description File

```
%-----------------------
% File: new_read_port_2x_o.pl
```

```
%----------------------
order_main([
...
apl_control_signal,
apl_added_signal_1,
apl_added_signal_2,
n_apl_added_signal_2,
apl_added_signal_3,
n_apl_added_signal_3,
apl_added_signal_4,
n_apl_added_signal_4,
apl_added_signal_5,
apl_added_signal_6,
apl_added_signal_7,
apl_added_signal_8,
apl_added_signal_9,
n_apl_added_signal_9,
apl_added_signal_10,
n_apl_added_signal_10,
apl_added_signal_11,
n_apl_added_signal_11,
apl_added_signal_12,
n_apl_added_signal_12,
apl_added_signal_13,
apl_added_signal_14,
apl_added_signal_15,
apl_added_signal_16,
flag,
n_flag
]).
```

A.4 Modified Algebraic Specification File

We simply use the built-in eq function for abstract comparator in this case, so no modification is needed. We make a copy of the original for the composite model and name it "new_read_port_2x_alg.pl".

A.5 Generated Condition file for the MDG model-checker

```
%-----------------------
% File: new_condition.pl
%-----------------------
signal(signal102,bool).
signal(flag,bool).
component(select102,constant_signal(value(1), signal(signal102))).
component(c_cond_comp102,fork(input(signal102),output(flag))).
outputs([flag]).
order_cond([signal102,flag]).
```

Bibliography

[1] MDG home page. http://hvg.ece.concordia.ca/mdg/.

[2] Quintus Prolog. http://www.sics.se/quintus/.

[3] Synopsys Formality home page. http://www.synopsys.com/products/verification/verification.html.

[4] VIS home page. http://vlsi.colorado.edu/~vis/whatis.html.

[5] IEEE Std 1800: SystemVerilog - Unified Hardware Design, Specification, and Verification Language, 2005.

[6] IEEE Std 1850: IEEE Standard for Property Specification Language (PSL), 2005.

[7] IEEE Std 1666: SystemC Language Reference Manual, 2006.

[8] Habibi A., Ahmed A. I., Mohamed O. A., and Tahar S. On the design and verification methodology of the look-aside interface. In *Proceedings on Design, Automation and Test in Europe*, volume 3, pages 290–295, 2005.

[9] Pnueli A. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, Jerusalem, Israel, Israel, 1997. Weizmann Science Press of Israel.

[10] Eisner C. and Fishman D. *A Practical Introduction to PSL.* Springer Science and Business Media, U.S.A., 2006.

[11] Grune D., Bal H., Jacobs C., and Langendoen K. *Modern Compiler Design.* John Wiley and Sons Ltd., England, 2004.

[12] Li D. Towards first-order symbolic trajectory evaluation using mdgs. Masters thesis, Concordia University, Montreal, Quebec, Canada, 2006.

[13] Li D. and Mohamed O. A. Mdg-based verification of the look-aside interface. In *Canadian Conference on Electrical and Computer Engineering*, pages 1064–1068, 2006.

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

[15] Cerny E., Corella F., Langevin M., Song X., Tahar S., and Zhou Z. *Automated Verification with Abstract State Machines using Multiway Decision Graphs.* Formal Hardware Verification: Methods and Systems in Comparison. Springer Verlag, 1997.

[16] Gascard E. From sequential extended regular expressions to determinstic finite automata. In *ITI 3rd International Conference on Information and Communications Technology*, pages 145–157, 2005.

[17] Emerson E.A. *Temporal and Modal Logic.* Handbook of Theoretical Computer Science. Elsevier Science Publisher, 1987.

[18] Clarke E.M. and Emerson E.A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131, pages 52–71. Springer Verlag, 1981.

[19] Corella F., Langevin M., Cerny E., Zhou Z., and Song X. State enumeration with abstract descriptions of state machines. In *Proc. IFIP WG 10.5*, 1995.

[20] Network Processing Forum. *Look-Aside (LA-1) Interface, Implementation Agreement, Revision 1.1.* Kluwer Academic Publishers, 2004.

[21] GNU. Bison: GNU parser generator. `http://www.gnu.org/software/bison/`.

[22] GNU. Flex: The fast lexical analyzer. `http://flex.sourceforge.net/`.

[23] Beer I., Ben-David S., and Landver A. On-the-fly model checking of rctl formulas. In *Computer Aided Verification*, pages 184–194, 1998.

[24] Morin-Allory K. and Borrione D. Proven correct monitors from psl specifications. In *DATE '06. Proceedings on Design, Automation and Test in Europe*, volume 1, pages 1–6, 2006.

[25] Boule M. and Zilic Z. Incorporating efficient assertion checkers into hardware emulation. In *Computer Design: VLSI in Computers and Processors.*, pages 221–228, 2005.

[26] Clarke E. M., Emerson E. A., and Sistla A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Language and Systems*, 8(2):244–263, 1986.

[27] Gordon M. and Melham T. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, Cambridge, U.k., 1993.

[28] Coudert O., Berthet C., and Madre J. C. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 365–373, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[29] Lichtenstein O. and Pnueli A. Checking that finite state concurrent programs satisfy their linear specification. In *POPL '85: Proceedings of the 12th ACM*

SIGACT-SIGPLAN symposium on Principles of programming languages, pages 97–107, New York, NY, USA, 1985. ACM Press.

[30] Burch J. R., Clarke E. M., McMillan K. L., and Dill D. L. Sequential circuit verification using symbolic model checking. In *27th Proceedings on Design Automation.*, pages 46–51, 1990.

[31] Burch J. R., Clarke E. M., McMillan K. L., Dill D. L., and Hwang L. J. Symbolic model checking: 1020 states and beyond. In *Fifth Annual IEEE Symposium on Logic in Computer Science, LICS '90*, pages 428–439, 1990.

[32] Boyer R.S. and Moore J.S. A theorem prover for a computation logic. Technical Report 54, Computational Logic, Inc., 1990.

[33] Owre S., Rushby J.M., and Shankar N. Pvs: a prototype verification system. In *International Conference on Automated Deduction*, pages 748–752, 1992.

[34] Abarbanel Y., Beer I., Glushovsky L., Keidar S., and Wolfsthal Y. Focs: Automatic generation of simulation checkers from formal specifications. In *Computer Aided Verification*, pages 538–542, 2000.

[35] Xu Y. Mdg model checker user's manual. Technical report, 1999.

[36] Xu Y. *Model Checking for a first-order temporal logic using multiway decision graphs.* Phd. thesis, University of Montreal, Quebec, Canada, 1999.

[37] Xu Y., Cerny E., Song X., Corella F., and Mohamed O. A. Model checking for a first-order temporal logic using multiway decision graphs. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 219–231, London, UK, 1998. Springer-Verlag.

[38] Zhou Z. Mdg tools v(1.0) user's manual. Technical report, 1996.

[39] Zhou Z., Song X., Corella F., Cerny E., and Langevin M. Description and verification of rtl designs using multiway decision graphs. In *ASP-DAC '95,CHDL '95, VLSI '95: Design Automation Conference*, pages 575–580, 1995.