Towards First-Order Symbolic Trajectory Evaluation using MDGs

Donglin Li

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

April 2006

Your file Votre référence
ISBN: 0-494-14270-7
Our file Notre référence
ISBN: 0-494-14270-7

ABSTRACT

Towards First-Order Symbolic Trajectory Evaluation using MDGs

Donglin Li

SoC design becomes more complex with the increasing amount of different kinds of IPs on the chip. How to ensure correctness of functionality in an SoC chip is one of the biggest challenges in SoC designs. Traditional BDD-based symbolic model checking techniques are an attractive subset of formal verification methods because of their high automation and little requirement for human effort to guide the proof process, whereas they usually suffer from the state explosion problem. Symbolic Trajectory Evaluation (STE) technique and MDG-based model checking technique improve the traditional BDD-based symbolic model checking approaches in two different ways. In this thesis, we investigate the possibility of combining STE and MDGs, for each of which we study the underlying theory and methodology, discuss the verification tool, and provide a detailed case study. The main purpose of these two case studies is to obtain an in-depth understanding of the underlying theories and methodologies of these two model checking techniques, which may facilitate the achievement of their combination. Two attempts to combine the STE with the MDG are discussed: one in the STE verification environment and the other in the MDG tools. We focus on the second attempt and propose a hybrid approach of First-Order Symbolic Trajectory Evaluation using MDGs, which can not

only increase the scale of circuits verified using STE but also improve the performance of STE by raising the level of abstraction. This study may also provide direction for further research in the application of MDGs.

To my family

ACKNOWLEDGEMENT

I would like to express my sincere gratitude and appreciation to my supervisor Dr. Otmane Ait Mohamed for providing me the opportunity to work in this challenging but exciting field and to be part of his research group, for his expert guidance, and for his support and encouragement throughout my research and thesis process.

Thanks must also go to Dr. Sofiène Tahar, who put great efforts to provide a truly wonderful research environment. I benefited a lot from his inspiring lectures as well.

I also owe a debt of gratitude to all the members of the HVG group. They were always friendly, supportive and encouraging to me. We really had a large group and it is almost impossible to name them all. Special thanks to Haja and Nasser for their valuable comments on the thesis.

I would especially like to thank Fariborz, his wonderful wife and lovely daughter. They were like family to me, who brought me happiness and helped me out whenever I was in difficulties.

Finally, acknowledgement and thanks to my parents and sister. I appreciate your love, support and understanding.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ABV	Assertion-Based Verification
AP	Atomic Proposition
ASM	Abstract State Machine
BDDs	Binary Decision Diagrams
CAM	Content Addressable Memory
CTL	Computation Tree Logic
DAG	Directed Acyclic Graph
DF	Directed Formula
DLL	Delay-Locked Loop
LA-1	Look-Aside Interface
LTL	Linear Temporal Logic
MDGs	Multiway Decision Graphs
ML	Meta Language
NPU	Network Processing Unit
PSL	Property Specification Language
QDR	Quad Data Rate
SML	Standard Meta Language
SoC	System-on-Chip
STE	Symbolic Trajectory Evaluation

Chapter 1

Introduction

System-on-Chip (SoC) design becomes more complex with the increasing amount of different kinds of IPs on the chip. How to ensure correctness of functionality in an SoC chip is one of the biggest challenges in SoC designs. Any SoC verification plan must cover the verification of the individual cores as well as that of the whole chip. The better knowledge of the external interfaces of each IP and their interactions with the SoC, the more complete the SoC verification will be. That is why people are putting more and more focus on the verification of different interfaces for SoC design. As for the SoC verification methods, basically there are no new relevant techniques which are different from what we have applied to the ASIC verification but just some adapted methodologies, like assertion based verification, theorem proving, model checking, etc. In this thesis, we are interested in two model checking techniques: Symbolic Trajectory Evaluation (STE) and model checking based on Multiway Decision Graphs (MDGs).

Model checking is a formal method for automatically verifying correctness of finite state transition systems, which has been studied since early 1980's and several important results of which have been established [18] [8]. These early model checking techniques were attractive because of their high automation and little requirement for human effort to guide the proof process, whereas they usually suffer from the state explosion problem and the size of the transition systems that could be verified were very limited. The

introduction of Bryant's Binary Decision Diagrams (BDD's) [20] into the original model checking algorithms led to a breakthrough in the size of transition systems that could be handled. A number of researchers have explored this BDD-based symbolic technique in the field of model checking and have published results of separate studies [17][24][13][14]. These symbolic model checking techniques provided exhaustive verification of a system by implicitly representing a state space through the use of a symbolic representation [12], and could deal with larger designs than traditional model checking techniques. However these BDD-based techniques were still not powerful enough for many real systems, when their models were larger than 100000 states [25], due to the state explosion problem.

Two model checking approaches: Symbolic Trajectory Evaluation and MDG-based model checking have been proposed to improve the traditional BDD-based symbolic model checking approaches.

Symbolic Trajectory Evaluation is a symbolic simulation based bounded model checking approach devised by Bryant and Seger [23]. By complementing the exhaustive analytical capabilities of symbolic model checking with the circuit modeling/manipulation methods of symbolic quaternary simulation, which gives STE the desirable property that the number of variables needed for the BDD's in an STE run depends only on the assertion being checked, not on the size of the circuit, STE effectively overcomes the state explosion problem and can verify much larger circuits, although it has its own limitation on the kind of properties it can handle. It's widely used at Intel, Compaq, IBM, and Motorola. At Motorola, it has been used to verify several

memory units, some with millions of transistors [22]. Also in [16] [42] STE has been used to verify CAMs (Content Addressable Memories) and PowerPC microprocessors.

The MDG-based model checking approach was proposed by Corella et al. in 1997 [10]. An MDG is an extended BDD-like data structure with arbitrary number of children for each node and with much more powerful labeling capability for both the nodes and the edges. BDDs can be viewed as a special case of MDGs. In this MDG-based approach, data signals are denoted by abstract variables instead of Boolean variables, and data operators are represented by uninterpreted or partially interpreted function symbols instead of Boolean functions. Thus, the verification based on abstract implicit state enumeration can be carried out independently of data path width, which therefore can effectively alleviate the state explosion problem.

The Symbolic Trajectory Evaluation technique and MDG-based model checking technique improve the traditional BDD-based symbolic model checking approaches in two different ways: the first one can dramatically reduce the computations for the next state space and enhance the computational efficiency, while the latter one can simplify the data path operations and thus can effectively alleviate the state explosion problem. This observation led to the idea of combining these two techniques, which makes it possible for us to take the advantages of both of them. The basic idea of such a combination is to replace the use of the BDDs with the MDGs for the encoding of the symbolic expressions and to implement the STE algorithm at a higher level of abstraction which can further alleviate the state explosion problem in STE. This combined approach will be discussed in detail in a later chapter of this thesis.

1.1 Background

1.1.1 Model Checking

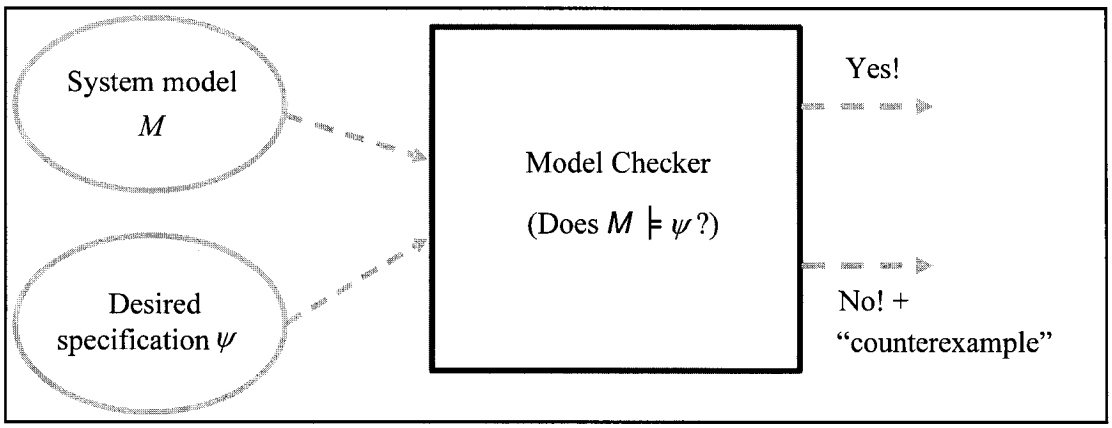


Figure 1. Model checking process

Typically a model checking process involves three major aspects: system modeling, a specification language and a model checking algorithm, shown in Figure 1.

The first step in model checking a design is to develop a formal model, usually expressed as a finite state transition system, for the circuits under study. The desired specifications of the design (properties) are captured by a specification language based on temporal logic. A model-checking tool accepts the system model and specifications. By exhaustively exploring the state space of the state transition system, the tool then returns "yes" if the given model satisfies the given specifications and "no" with a counterexample otherwise. The counterexample demonstrates how the error occurs. The termination of model checking is guaranteed by the finiteness of the model.

1.1.1.1 Modeling

A finite state transition system can be described as a Kripke structure: $M = (S, SI, T, L)$, where

- S : a finite set of states,
- $SI \subseteq S$: the set of initial states,
- $T \subseteq S \times S$: a transition relation with $\forall s \in S\,(\exists\, s' \in S\,((s,s') \in T))$,
- L: $S \rightarrow 2^{AP}$: a labeling function, associating each state with a set of atomic propositions (APs).

Note that every state must have a successor in T, which means that it is always possible to have an infinite sequence of states in the Kripke structure.

A path is an infinite sequence of states such that each state is related to its successor by the transition relation.

Atomic propositions represent the basic properties that hold in the associated states.

1.1.1.2 Specification Languages

The properties of a design are expressed as temporal logic formulas [2][15]. Temporal logic is a kind of logic which views time as a sequence of states. Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) or *branching time logic* are two most commonly used temporal logics in the context of model checking. LTL expresses temporal properties over a linear execution sequence, i.e. a single sequence of states, of the state transition system. CTL, on the other hand, can express properties across several different sequences of states simultaneously.

Temporal logics use atomic propositions as their building blocks and combines theses propositions into formulas using logical operators and temporal operators. Atomic propositions (p, q, ...) are variables which can either be true or false. The logical operators used in temporal logic formulas are the usual connectives: $\neg, \vee, \wedge, \rightarrow$ and $\leftrightarrow$. The temporal operators are classified into two groups: state operators and path operators. State operators are used to select states: Gp ("always p", "globally p"), Fp ("sometime p", "finally p"), Xp ("nexttime p") and pUq ("p until q"). Path operators are used to select paths: A ("all paths"), E ("there exists a path").

Note that path operators are only applied to CTL formulas since in LTL there is no concept of branching and hence no need for selecting paths. We can say that the absence of A and E path operators in LTL formulas which reflects the linear-time paradigm (as opposed to branching-time paradigm in CTL) is the major difference between LTL and CTL.

The model M defined in previous section can be viewed as a temporal model where each state represents a point in time. Within each state, atomic propositions are true or false; hence a temporal logic formula can be evaluated to true or false from its subformulas in a recursive fashion until reaching atomic propositions. Note that a temporal logic formula that is true in some states might not hold in other states for a given model.

A temporal formula p is satisfied by a model $M = (S, SI, T, L)$ if it is true for all the initial states SI of the model, i.e. $SI \subseteq \{ s \in S \mid M, s \models p\}$. The recursive definition of $\models$ for a CTL formula is as following:

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

- M, $s_0 \models \neg p$ iff not M, $s_0 \models p$.
- M, $s_0 \models (p_1 \wedge p_2)$ iff M, $s_0 \models p_1$ and M, $s_0 \models p_2$.
- M, $s_0 \models AXp$ iff for all states s_0' with $(s_0, s_0') \in T$, M, $s_0' \models p$.
- M, $s_0 \models EXp$ iff for some state s_0' with $(s_0, s_0') \in T$, M, $s_0' \models p$.
- M, $s_0 \models A[p_1 \, U \, p_2]$ iff for all paths $(s_0, s_1, \ldots)$, there exists an $j \geq 0$ with M, $s_j \models p_2$, and M, $s_i \models p_1$ holds for all $0 \leq i < j$.
- M, $s_0 \models E[p_1 \, U \, p_2]$ iff for some path $(s_0, s_1, \ldots)$, there exists an $j \geq 0$ with M, $s_j \models p_2$, and M, si $\models p_1$ holds for all $0 \leq i < j$.

1.1.1.3 Model Checking Algorithms

A Model checking algorithm is used to decide if a system satisfies a temporal property and generates a counterexample otherwise. Different temporal logic model checking algorithms have been devised to target LTL model checking and CTL model checking.

The complexity of model checking algorithms with temporal logics have been studied since early 1980's and several important results have been established.

In 1985 Pnueli and Lichtenstein [18] presented a model checking algorithm with linear time temporal logic formulas and the complexity of this algorithm was shown to be exponential in the length of the formula but linear in the size of the transition system.

Clarke, Emerson and Sistla [8] devised an algorithm for CTL model checking and the complexity of the algorithm was proved to be linear in the length of the formula and also linear in the size of the transition system.

Another type of branching-time logic is CTL*, introduced by Clarke, Emmeson and Sistla[8], which combined CTL and LTL and could be checked with the same time complexity as the LTL formulas.

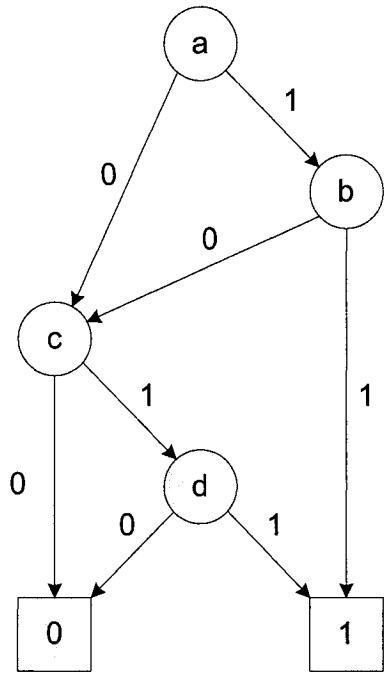
These early model checking algorithms are so-called state exploration algorithms which require explicitly constructing the state graph of the circuit under study and a complete exploration of the state space. They were attractive because of their high automation and little requirement for human effort to guide the proof process, whereas they usually suffered from the state explosion problem and the size of the transition systems that could be verified by them were very limited.

Several techniques were developed to overcome this problem in certain aspects, among which symbolic algorithms have shown great success.

1.1.2 BDDs

A Binary Decision Diagram is a data structure which allows us to represent a Boolean function as a rooted acyclic-directed graph where each non-terminal vertex is labeled by a variable and has two directed edges, labeled 0 and 1, respectively. Terminal vertices are labeled either 0 or 1. Figure 2 shows a BDD for the function f(a, b, c, d) = (a∧b)∨(c∧d) with a truth table representing this function at the right.

To evaluate the represented Boolean function for a given valuation of the function arguments, a path is traced in the BDD from the root vertex down to a terminal by taking at each vertex the edge labeled with the value of the labeling variable of this vertex. The value of the labeling variable of the terminal reached by this path defines the value of the Boolean function under the current valuation. For example, to evaluate f(a=0, b=1, c=1, d=1), start at the root a, traverse down the edge labeled 0 to c, then down two edges labeled 1 until reach the terminal labeled 1, which means that the value of f is 1 with respect to the valuation (a=0, b=1, c=1, d=1).



a	b	c	d	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Figure 2. BDD and true table for function f(a, b, c, d) = (a∧b)∨(c∧d)

As shown by Bryant [20], a reduced ordered BDD offers a canonical representation of a given Boolean function, in other words, every Boolean function can be represented by a unique reduced BDD for a given ordering of the input variables. By using reduced ordered BDD's, a set of algorithms can be developed for manipulating Boolean functions with a high degree of efficiency.

1.1.3 Symbolic Model Checking

The basic idea of symbolic model checking is to represent the state space symbolically. Burch, Clarke, McMillan, and Dill [14] presented a symbolic CTL model checking algorithm to verify sequential circuits, where the transition relation for the entire system is represented symbolically as a characteristic function of all of state variables in the system. By manipulating the BDD representations of the state space and the temporal

formula, model checking can be performed with efficient algorithms existing for BDD-Based Boolean Manipulation.

The strength of this algorithm stems from the fact when this symbolic representation captures the right structural uniformities in the graph, it is much smaller than an explicit table of all of the states [13], and thus it can be applied to verify some very large sequential circuits.

However, this method can not be generally applied to verify all the large circuits with complex data paths, and in many cases it will still have the state explosion problem. Another drawback of the algorithm is that it can be very computationally consuming to generate this characteristic function [23].

1.1.4 MDGs

MDGs can be viewed as a generalization of BDDs. BDDs offer representations of Boolean formulas. Graph D in Figure 3 depicts the BDD for 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 D_0 and D_1 respectively.

Alternatively, graph D can be viewed as representing a formula $((x = 0) \wedge F_0) \vee ((x = 1) \wedge F_1)$ in a many-sorted first-order logic. More generally, node a can range over a larger set of values than {0, 1} and can even range over abstract terms. It is also possible that a cross operator can be a decision node in a generalized decision graph. The definitions of an abstract term and a cross operator can be found in Chapter 4 where more details about the many-sorted first-order logic and MDGs are presented.

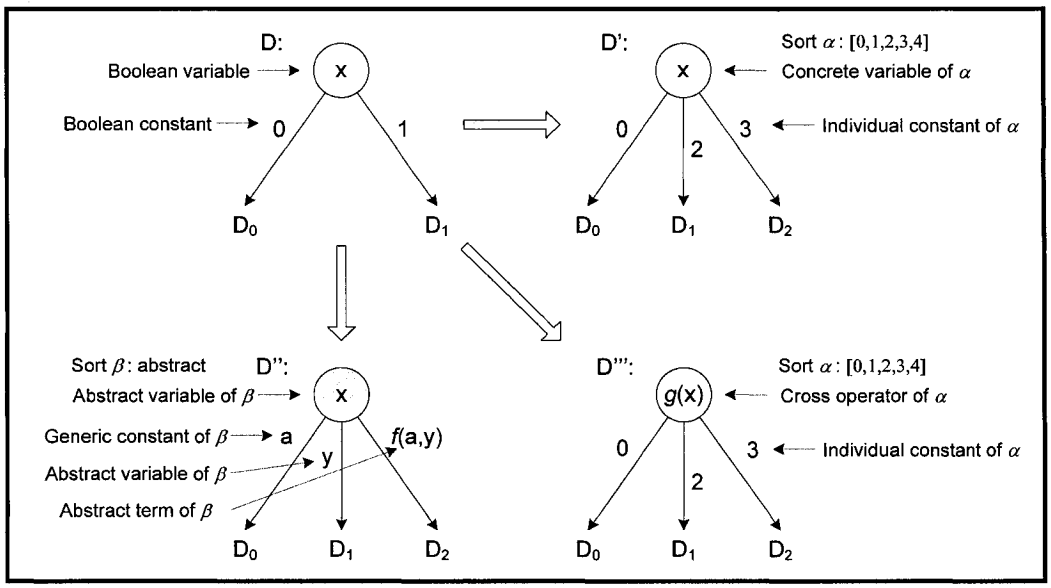


Figure 3. From BDDs to MDGs

Three possible generalizations of BDD D and the corresponding formulas are shown in Figure 3 and below, where F_0, F_1 and F_2 are first-order formulas represented by the sub-graphs D_0, D_1 and D_2 respectively:

- From D to D′: $x \in \{0,1\} \rightarrow x \in \{0,2,3\}$, and Graph D′ represents the formula

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

- From D to D″: $x \in \{0,1\} \rightarrow x \in \{a, y, f(a,y)\}$, and Graph D″ represents the formula

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

- From D to D‴: $x \in \{0,1\} \rightarrow g(x) \in \{0,2,3\}$, and Graph D‴ represents the formula

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

The above generalized decision graph D, D′ and D‴ are examples of Multiway Decision Graphs (MDGs).

1.2 Thesis Contributions

The main contributions of the thesis are as follows:

- We investigated the underlying theory and methodology of Symbolic Trajectory Evaluation (STE) and MDG-based model checking techniques by a detailed case study for each of them.
- We proposed a hybrid approach of performing first-order Symbolic Trajectory Evaluation using MDGs, which can not only increase the scale of circuits that can be verified using STE but also improve the performance of STE by raising the level of abstraction.

1.3 Scope of Thesis

Symbolic Trajectory Evaluation (STE) and MDG-based model checking are two model checking techniques which improve the traditional symbolic model checking approaches in two different ways. The aim of this thesis is to investigate the possibility of using MDGs to perform STE. The motivation of combining these two techniques is to develop a more powerful model checking technique which will take the advantages of both of them.

The rest of the thesis is organized as follows:

In Chapter 2, we study the underlying theory and methodology of Symbolic Trajectory Evaluation, provide an illustrative example of this approach, and describe an STE-based verification tool.

Chapter 3 provides a case study of STE. We first make a brief introduction to the Look-Aside Interface (LA-1), and then, after the discussion of some related work including a previous RTL design for the LA-1 interface, present a modified RTL design for the LA-1 interface. Finally, the verification processes of both the designs using STE are illustrated followed by experimental results.

Chapter 4 introduces the theoretical foundations and methodology of MDG-based model checking followed by an illustrative example, and discusses MDG-based model checking tools.

In Chapter 5, we use the MDG tools to verify the same properties of the LA-1 Interface. Our goal is to compare the two methods and prepare the ground for our proposal to define a symbolic trajectory evaluation in MDG. This experiment involves a mapping from STE assertions to L_{MDG} properties. We will provide a method to perform this mapping. Experimental results are given at the end of this chapter.

Based on the exploration of STE and MDG-based model checking through case studies, in Chapter 6, we describe two attempts to combine these two model checking techniques: one in the STE verification environment and the other in the MDG tools. We focus on the second attempt and propose a hybrid approach of performing first-order Symbolic Trajectory Evaluation in the MDG tools.

Finally, in Chapter 7, we conclude this thesis and present the future work.

Chapter 2

Symbolic Trajectory Evaluation

This chapter describes the underlying theory and methodology of Symbolic Trajectory Evaluation (STE), a symbolic simulation based bounded model checking technique. Firstly, lattice and symbolic simulation are described, which are the theoretical foundations of STE. Next, the chapter discusses in detail the modeling, specification language and verification methodology of STE, followed by an illustrative example. An STE-based verification tool is described at the end.

2.1 Basics

Symbolic Trajectory Evaluation is a symbolic simulation based bounded model checking approach devised by Bryant and Seger [23], which relates most closely to the symbolic model checking algorithm proposed by Bose and Fisher [24]. In Bose and Fisher's algorithm, an explicit representation of the next state function for every state variable in the system is extracted using symbolic simulation. In one simulation run, each state variable and each input signal is represented by a distinct Boolean variable, and a Boolean representation of the next state behavior is computed. A temporal logic formula can then be checked using symbolic Boolean manipulation. This extraction process for the explicit next state function can be quite costly [23].

What distinguishes STE from Bose and Fisher's method and other symbolic model checking algorithm is its representation of state space and the next state behavior.

- The state space is represented using a lattice-based model.
- The next state function is represented implicitly as a result of combining the circuit structure and the simulation algorithm, and the next state behavior is computed only for the particular patterns required for the verification of a given assertion.

These particular patterns involve far fewer variables than is required in Bose and Fisher's method [23]. The strength of STE comes largely from the fact that the complexity of an STE run depends only on the complexity of the STE assertion itself rather than that of the circuit being checked. STE offers an effective alternative to classical symbolic model checking techniques which often suffer from state explosion and hence can verify much larger circuits, although it has its own limitation on the kind of properties it can handle.

2.1.1 Lattice

In this thesis, we view a lattice as a partially ordered set. The discussion of other uses of lattice is outside the scope of this thesis and will not be included here.

A partial order is a binary relation on a set which is reflexive, antisymmetric, and transitive. Given a partial order R on a set S, for all $a, b, c \in S$, we should have:

- $a \; R \; a$ (reflexivity),
- $(a \; R \; b \text{ and } b \; R \; a) \rightarrow a = b$ (antisymmetry), and
- $(a \; R \; b \text{ and } b \; R \; c) \rightarrow a \; R \; c$ (transitivity).

This partial order relation formalizes the intuitive concept of an ordering of the set elements, which represents a hierarchy of information or knowledge. More specifically, the higher order an element has, the more information it contains. The 'partial' here

indicates that such an ordering does not necessarily need to be total, that is, not all pairs of elements in the set are mutually comparable.

We call a set with a partial order a partially ordered set.

Definition 2.1.1.1: A partially ordered set $(L, \leq)$ [9] is a lattice if for any elements a and b of L, the set $\{a, b\}$ has both a least upper bound (join) and a greatest lower bound (meet), where L is the so called ground set and $\leq$ is the partial order.

The join of a and b is denoted by $a \vee b$, and the meet is denoted by $a \wedge b$, where $\wedge$ and $\vee$ are binary operations.

Definition 2.1.1.2: A complete lattice is a partially ordered set $(L, \leq)$ which has both a greatest lower bound (meet) and a least upper bound (join) for every subset A of L, denoted by $glb(A)$ and $lub(A)$, respectively.

In other words, a complete lattice is a complete relation with a bound on every subset. Note that each complete lattice has a unique greatest element (often called universal upper bound) and a unique least element (often called universal lower bound). A complete lattice is a special case of lattices.

The power set (the collection of all subsets) of a given set S forms a complete lattice using "subset of" as the ordering relation $\leq$. Meet and join of subsets can be obtained by the set operations intersection and union, respectively. In this class of lattices, the empty set is least element, and S is greatest element. A lattice based on the power set of $\{a,b,c\}$ is shown in Figure 4. A diagram of a partial order that leaves out the transitive relations is referred as a Hasse diagram. The ground set L of this lattice is $\{\varnothing, S_1, S_2, S_3, S_4, S_5, S_6, S\}$. The ordering of these elements is listed below:

- $\varnothing \leq S_1, \varnothing \leq S_2, \varnothing \leq S_3$;

- $S_1 \leq S_4$, $S_1 \leq S_5$;
- $S_2 \leq S_4$, $S_2 \leq S_6$;
- $S_3 \leq S_5$, $S_3 \leq S_6$;
- $S_4 \leq S$;
- $S_5 \leq S$;
- $S_6 \leq S$;

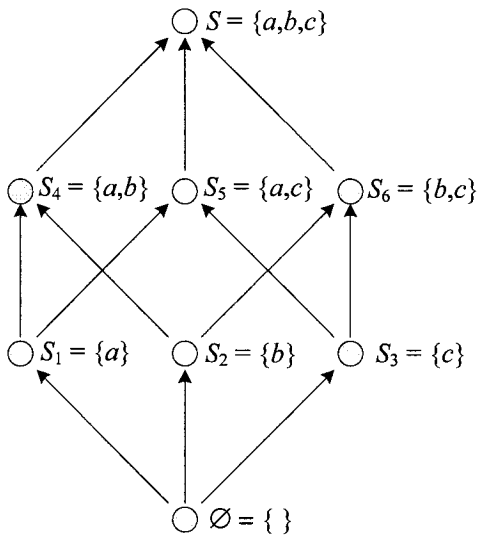


Figure 4. Lattice structure for the power set of {a,b,c}

The ordering relation is represented by a directed edge pointing from the element with lower order to the one with higher order. Note that there is no ordering relation applied to the element pairs of $\{S_1, S_2\}$, $\{S_1, S_3\}$, $\{S_2, S_3\}$, $\{S_4, S_5\}$, $\{S_4, S_6\}$ and $\{S_5, S_6\}$, since the two elements in each of the above pairs are not mutually comparable under the relation "subset of", which reflects the "partial" feature of this ordering.

It is not hard to tell from the diagram that each of the 2^8 subsets of L has a unique least upper bound and a unique greatest lower bound.

2.1.2 Symbolic Simulation

In digital hardware verification, the term simulation is referred to a modeling technique which describes the state transitions, inputs and outputs of a digital system. Simulators are often used to test logic designs before constructing the real hardware.

Figure 5 illustrates a simulator example of a 5-input OR gate. During simulation, a sequence of input patterns 01100, 10011, ... are fed into the input ports of the simulator which models the behavior of the circuit, and the corresponding output response patterns are sampled and checked against the expected values at the output ports of the simulator. A single run of this simulator can only determine the behavior of the OR gate, that is the output response in this case, for a single input pattern. In order to verify this 5-input OR gate exhaustively, we need 32 (2^5) test patterns to cover all the possibilities of the input signals and, therefore, need 32 simulation runs

The number of the required test patterns for exhaustive verification will grow exponentially with the number of input signals. In this case of sequential circuit simulation, the situation will be even worse. We need to take into consideration not only the input sequence but also the initial state of the system. Thus, simulation is only applicable for verifying very small systems, the limitation of which comes from the stimulus generation and simulation runtime.

Symbolic simulation is a promising method to generalize the traditional simulation technique and make it feasible to larger systems. A symbolic simulator resembles a traditional simulator, except that it simulates the design using Boolean variables instead of constant binary values at the inputs of the circuit model.

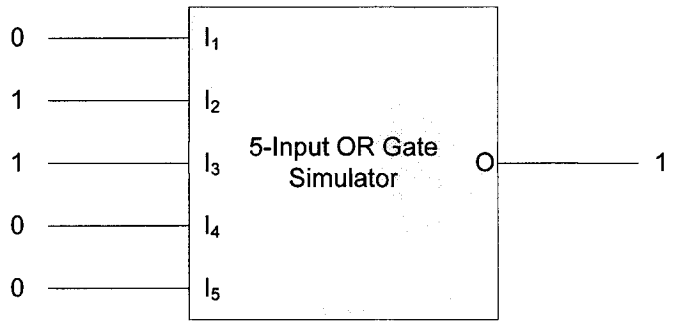


Figure 5. Simulator for a 5-input OR gate

During simulation the values of the circuit state and the circuit output are represented as Boolean functions over these initial variables. At the same time, logic operations, such as AND, OR and NOT, should be refined to operate over Boolean functions rather than over the constants 0 and 1. At the end of each simulation run, a set of Boolean functions representing implicitly all set of states that are reachable by the current circuit state in one clock cycle for the input variables can obtained by manipulating Boolean operations, and so can the Boolean functions for the outputs. This method allows all the next state behaviors of a circuit in a specific state under all possible inputs to be verified with a single simulation run simultaneously. In other words, a single symbolic simulation run can compute information that would otherwise need to be obtained by multiple traditional simulation runs.

The symbolic simulator of the same 5-input OR gate is shown in Figure 6. The input signals of the simulator are represented by Boolean variables a, b, c, d and e, respectively. The output of the simulator is a Boolean function over these five variables. In this case, we only need 1 symbolic test pattern, abcde, to verify this 5-input OR gate exhaustively.

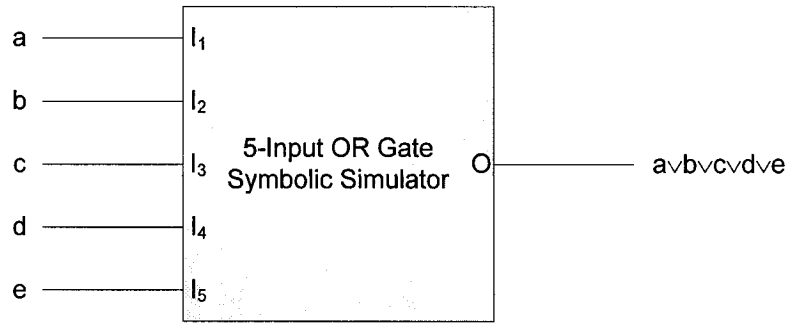


Figure 6. Symbolic simulator for a 5-input OR gate

One problem with symbolic simulation is that it needs to exhaustively manipulate the circuit functionality and requires extensive manipulation of Boolean expressions.

Ternary simulation [21] is another generalized simulation approach, in which three-valued logic is used instead of two-valued logic. Three-valued logic extends the existing 1 (true) and 0 (false) values in two-valued logic with an unknown or "don't care" value X. In order to perform ternary simulation, functions defined over Boolean values {0, 1} need to be extended to ones defined over ternary values {0, 1, X}. The extensions should obey the following rule: if a circuit node is computed to be either 0 or 1, this result will not change if the X's contained in the stimulus pattern are replaced partially or completely by 0 or 1. This extension rule guarantees that the simulator of a circuit will produce the same response to a certain input pattern even if some bits of it are set to X. As an example, the truth tables of some Boolean operations and their ternary extensions are shown in Figure 7.

Since each use of value X covers two cases of using 0 and 1 in ternary simulation, the number of test patterns required to verify a circuit will be reduced dramatically by introducing X's when applicable. Take the same 5-input OR gate discussed above as an

example. The ternary simulator model and the corresponding truth table of it are given in Figure 8. Compared with the 32 test patterns required by the traditional simulation, only six test patterns, XXXX1, XXX1X, ..., 1XXXX and 00000 are needed to do the exhaustive verification of this 5-input OR gate using ternary simulation. The strength of this approach stems from its computational efficiency.

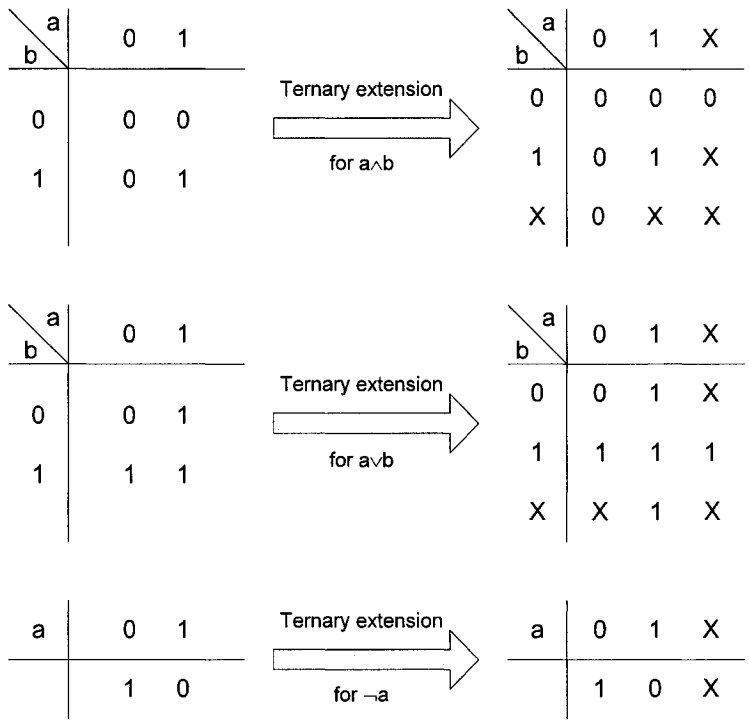


Figure 7. True tables of Boolean operations and the ternary extensions

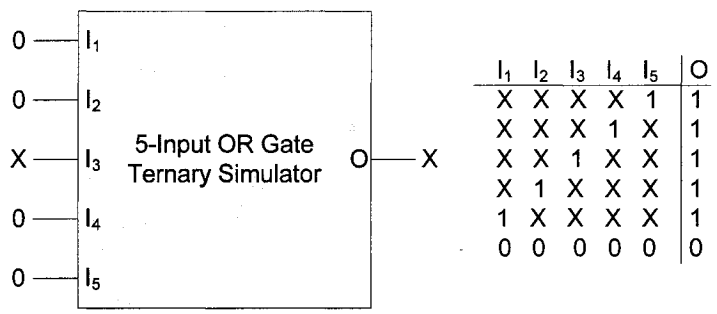


Figure 8. Ternary simulator model for a 5-input OR gate

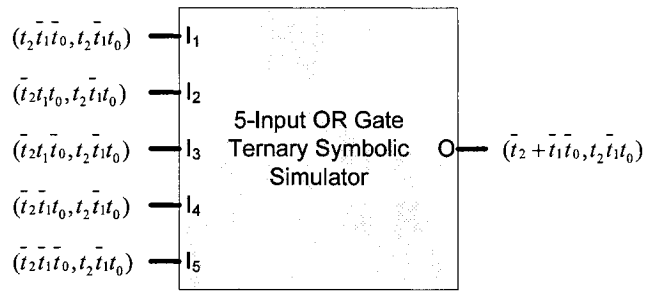
Note that the use of X value may cause information loss of the circuit. Let us take a look at the 5-input OR gate model in Figure 8 again. When four of the inputs are set to 0, the attempt to set the left input to X will lead to an X value at the output of the simulator, a meaningless result in verification. To avoid this, we have to be careful with our choice of using X values in the simulation.

Symbolic simulation and ternary simulation improve the performance of traditional simulation technique in two different ways. Symbolic simulation can dramatically cut down the number of required stimulus patterns and, therefore, the number of simulation runs at the price of increasing the computation complexity and the memory usage. Ternary simulation, on the other hand, can significantly enhance the computational efficiency, but may have the problem of information loss in some cases. If we can combine these two techniques, it is possible for us to take the advantages of both of them. A successful attempt was made by Beatty, Bryant and Seger [5], whose approach is named ternary symbolic simulation.

The key idea of this ternary symbolic simulation approach is to parameterize ternary values by Boolean variables, which can further reduce the number of required test patterns. Figure 9 illustrates how the six test patterns required by the 5-input OR gate ternary simulator are compressed to one symbolic pattern in ternary symbolic simulation.

First, we index the six scalar patterns with numbers from 0 to 5, which are then encoded with three ($\lceil \log(5+1) \rceil$) Boolean variables t_0, t_1 and t_2. Next, for each input of the 5-input OR gate, we represent the six ternary values gathering from the corresponding bits of the six test patterns as a function pair (high, low) over these three variables. Function "high" and "low" indicate the positions of value 1 and 0, respectively, and the

unrepresented positions are of value X. Thus, in our example, we have one symbolic test pattern consisting of five function pairs over three variables to replace all the six scalar test patterns. The output of the ternary symbolic simulator is also a (high, low) function pair.



	scalar patterns						symbolic pattern	
	0	1	2	3	4	5	high	low
I_1	X	X	X	X	1	0	$t_2\bar{t}_1\bar{t}_0$	$t_2\bar{t}_1t_0$
I_2	X	X	X	1	X	0	$\bar{t}_2t_1t_0$	$t_2\bar{t}_1t_0$
I_3	X	X	1	X	X	0	$\bar{t}_2t_1\bar{t}_0$	$t_2\bar{t}_1t_0$
I_4	X	1	X	X	X	0	$\bar{t}_2\bar{t}_1t_0$	$t_2\bar{t}_1t_0$
I_5	1	X	X	X	X	0	$\bar{t}_2\bar{t}_1\bar{t}_0$	$t_2\bar{t}_1t_0$
O	1	1	1	1	1	0	$\bar{t}_2 + \bar{t}_1\bar{t}_0$	$t_2\bar{t}_1t_0$

Figure 9. Ternary symbolic simulator for a 5-input OR gate

Recalling the 32 test patterns required for the traditional simulator of 5-input OR gate, 1 symbolic test pattern over 5 variables for the symbolic simulator and 6 ternary test patterns for the ternary symbolic simulator, our ternary symbolic simulator has the best performance among all these simulators.

Note that reduced ordered BDDs can also be applied in ternary symbolic stimulation, since ternary values are manipulated implicitly via binary encodings mentioned above.

Note also that the above ternary symbolic simulation approach just shows us one of the ways to combine ternary modeling technique with symbolic simulation and in practice we may have our own ways to do the combination depending on the applications.

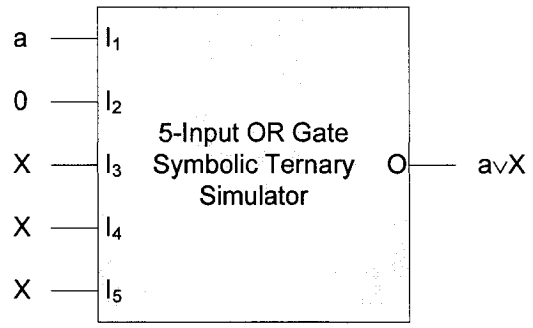


Figure 10. Symbolic tenary simulator for a 5-input OR gate

Another possible way to symbolize the ternary simulation is to partially symbolize the test patterns using ternary variables and ternary constant. An example of a partially symbolized test pattern for the 5-input OR gate is shown in Figure 10, where input signal I_1 is represented by variable 'a' over ternary values {0, 1, X}, input I_2 is represented by ternary constant 0, and input I_3, I_4 and I_5 are all represented by ternary constant X. This partially symbolized test pattern covers three possible ternary test patterns: X0XXX, 00XXX and 10XXX, and the corresponding outputs of the circuit will be of values X, X and 1 respectively.

Note that for some special simulators, the "don't care" value X appearing at the output of a gate may have specific meaning for the verification and should not be deemed as meaningless.

2.2 Modeling

Symbolic trajectory evaluation (STE) extends symbolic simulation with some of the analytic capability of temporal logic model checking techniques [6]. As in a model checking approach, STE also develops a formal model for the circuit under verification,

but different from the temporal logic model checkers, it uses a lattice-based model instead of a Kripke structure.

The lattice-based model in STE is a tuple $M = [(S, \leq), Suc]$, where:

- S is a set of finite states,
- $\leq$ is a partial order over S,
- $(S, \leq)$ is a complete lattice,
- Suc: $S \rightarrow S$ is the next state function, monotone with respect to $\leq$.

A function between ordered sets is monotone if it preserves the given order. For function Suc, whenever $s_i \leq s_j$ and $s_i, s_j \in S$, then $Suc(s_i) \leq Suc(s_j)$.

The state space $S = \{0, 1, X\}^n$ is a set of n-length vectors over ternary values for some natural number n. The partial order $\leq$ is defined over $\{0, 1, X\}$ first and extended to $\{0, 1, X\}^n$. Figure 11 illustrates the partial orders over $\{0, 1, X\}$ and $\{0, 1, X\}^2$.

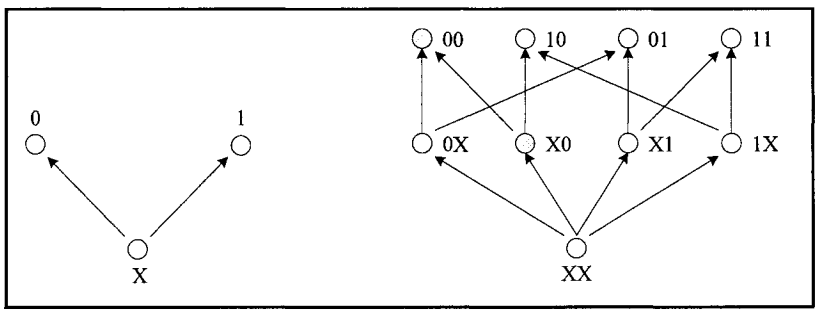


Figure 11. Partial orders over $\{0, 1, X\}$ and $\{0, 1, X\}^2$

According to the definition of complete lattice, $(\{0, 1, X\}^n, \leq)$ is not a complete lattice, since not every subset of $\{0, 1, X\}^n$ has a least upper bound. In order to make $(S, \leq)$ a complete lattice, we introduce a top element $\top$ to the state set S. We use $\top$ to

represent a unique "overconstrained" state [4], where some node of the state vector is set to both 0 and 1 at the same time. Thus, the state set $S = \{0, 1, X\}^n \cup \{\top\}$ and the partial order $\leq$ form a complete lattice with $\top$ as the universal upper bound and $\perp = X, \ldots, X$ as the universal lower bound. The Hasse diagrams of complete lattices $(\{0, 1, X\} \cup \{\top\}, \leq)$ and $(\{0, 1, X\}^2 \cup \{\top\}, \leq)$ are shown in Figure 12.

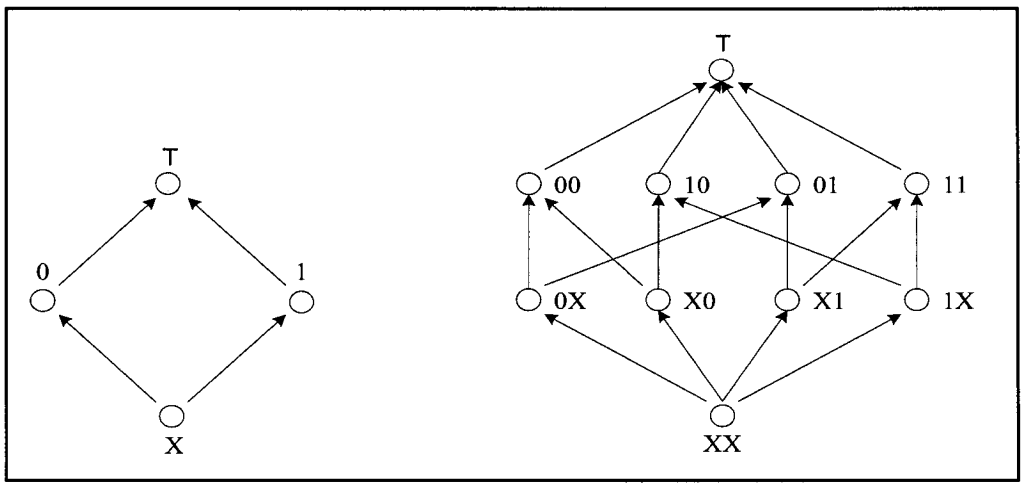


Figure 12. Hasse diagrams of complete lattices $(\{0, 1, X\} \cup \{\top\}, \leq)$ and $(\{0, 1, X\}^2 \cup \{\top\}, \leq)$

The next state function *Suc*: $\{0, 1, X\}^n \cup \{\top\} \rightarrow \{0, 1, X\}^n \cup \{\top\}$ is used to compute constraints on the possible values of the successor states of the current states. In other words, for a given state $s \in S$, function $Suc(s)$ computes the least specified (most general) successor state the system can be in one time step later. The "least specified (most general)" is defined by the partial order $\leq$ [4]. As mentioned in previous section, in a partially ordered set, the lower order an element has, the less specified it is. In this sense, for example, value X is less specified than values 0, 1 and $\top$ in the complete lattice $(\{0, 1, X\} \cup \{\top\}, \leq)$. Note that the constraints computed by the next state function are the ones

imposed by the circuit itself and are irrelevant to the circuit inputs which are controlled externally.

For a given state vector $s = c_0 c_1 \ldots c_{n-2} c_{n-1} \in S$, the next state function $Suc(s)$ is actually a vector of next state functions for each node (component) of s, i.e., $Suc(s) = t_0(c_0)t_1(c_1)\ldots t_{n-2}(c_{n-2})t_{n-1}(c_{n-1})$, where c_i: $\{0, 1, X\}^n \rightarrow \{0, 1, X\}$ for $1 \leq i \leq n\text{-}1$. If node c_i is associated with an input of the circuit, the next state function $t_i(c_i) = X$, and otherwise $t_i(c_i)$ is determined by the circuit structure. A constraint of value X indicates that no constraint is imposed on that node. For the state vector T, the next function $Suc(\mathsf{T})$ equals T.

The next state function works like a filter that can filter out irrelevant sequences of states which are not useful for reasoning about model behaviors. We call these useful sequences trajectories. Formally, given a model $M = [(S, \leq), Suc]$, an infinite sequence $(s_0, s_1, s_2, \ldots)$ of states of S is a *trajectory* iff

$$Suc(s_i) \leq s_{i+1} \text{ for } i \geq 0.$$

The set of all trajectories of model M is denoted as $J(M)$.

2.3 Specification Language

In STE, a design specification is expressed as a trajectory assertion in a restricted temporal logic. A trajectory assertion is of the form [*Ante* $\rightarrow$ *Cons*], where both *Ante* and *Cons* are trajectory formulas. The main verification task is to evaluate whether or not every trajectory satisfying *Ante* (called the antecedent) also satisfies *Cons* (called the consequent).

The basic component of a trajectory formula is a simple predicate. Given a model M = [(S, $\leq$), *Suc*], a predicate over S is a function that maps S to a special complete lattice containing only two elements *false* and *true*, with element *false* as the universal lower bound and element *true* as the universal upper bound. A predicate ρ over S is called simple if it is monotone and there exists a unique element d_ρ in S such that for all $s \in S$ with $d_\rho \leq s$, $\rho(s)$ = *true*. The d_ρ here is called the defining value of predicate ρ.

Definition 2.3.1: If we denote a set of simple predicates over S by P, a trajectory formula of model M is then defined inductively as below:

- A simple predicate $\rho \in$ P is a trajectory formula.
- The conjunction $(f_1 \wedge f_2)$ is a trajectory formula if both f_1 and f_2 are trajectory formulas.
- The domain restriction $(b \rightarrow f)$ is a trajectory formula if b is Boolean constant 0 or 1 and f is a trajectory formula.
- The next time expression (**N***f*) is a trajectory formula if f is a trajectory formula and **N** is the next-time operator.

If a trajectory formula f of model M is satisfied by a trajectory $a = (a_0, a_1, a_2, \ldots)$ of the same model, we write $a \models_M f$. The satisfaction relation $\models_M$ is defined as follows:

- $a \models_M \rho$ iff $\rho(a_0)$ = true.
- $a \models_M (f_1 \wedge f_2)$ iff $a \models_M f_1$ and $a \models_M f_2$.
- $a \models_M (1 \rightarrow f)$ iff $a \models_M f$;
- $a \models_M (0 \rightarrow f)$ always holds.
- $a \models_M (\mathbf{N}f)$ iff $(a_1, a_2, \ldots) \models_M f$

2.4 Verification Methodology

The definition of simplicity can be extended directly from predicates to formulas. A formula f of model M is called simple if it is monotone and there exists a unique trajectory α_f in M, called defining trajectory of formula f, such that for all $a \in J(M)$ with $\alpha_f \leq a$, $a \models_M f$ holds. We will see later that trajectory formulas are simple and we can construct the defining trajectory for every trajectory formula. Thus, the main verification task that checks whether or not every trajectory satisfying *Ante* also satisfies *Cons* can be implemented in this way: compute the defining trajectory for the trajectory formula *Ante* first, and then verify that this defining trajectory satisfies trajectory formula *Cons*.

Before constructing a defining trajectory for a given trajectory formula f in M, we will first show how to construct its defining sequence β_f. This sequence should be the least possible sequence in M that satisfies f, i.e., for all sequence a with $\beta_f \leq a$, $a \models_M f$ holds.

Definition 2.4.1: Given a model $M = [(S, \leq), Suc]$ and a set P of simple predicates over S, the recursive definition of the defining sequence β_f of a trajectory formula f in M is as following:

- $\beta_\rho = d_\rho \perp\perp\ldots$ if d_ρ is the defining value of $\rho \in \mathrm{P}$.
- $\beta_{f_1 \wedge f_2} = \mathrm{lub}(\beta_{f_1}, \beta_{f_2})$, where *lub* is the lowest upper bound function.
- $\beta_{(b \to f)} = b ? \beta_f$, where b is a Boolean constant, '?' is an infix "multiplexing" function, and $b ? \beta_f = \begin{cases} \beta_f & \text{if } b = 1 \\ \perp\perp\ldots & \text{otherwise} \end{cases}$.

- $\beta_{Nf} =\perp \beta_f$.

It can be proved [4] that for any given trajectory formula f in M and its defining sequence β_f constructed as above,

$$a \models_M f \Leftrightarrow \beta_f \leq a, \text{ for all } a \in J(M).$$

Note that β_f is not necessary a trajectory of M whereas our goal is to construct the defining trajectory, that is the least possible trajectory that satisfies f, and therefore we need to go one step further. One possible way is to combine the constraints on a sequence imposed by β_f and those from the next state function *Suc* to obtain the required trajectory.

Definition 2.4.2: Given any trajectory formula f of model M = [(S, $\leq$), *Suc*], assuming that $\beta_f = \beta_f^0 \beta_f^1 \ldots$ is the defining sequence for f, a sequence $\chi_f = \chi_f^0 \chi_f^1 \ldots$ can be defined inductively as follows:

$$\chi_f^i = \begin{cases} \beta_f^0 & \text{if } i = 0 \\ \text{lub}(\beta_f^i, Suc(\chi_f^{i-1})) & \text{otherwise} \end{cases}.$$

For the sequence χ_f constructed as above, it can be proved [4] that,

- $\chi_f \in J(M)$,
- $\chi_f \models_M f$, and
- $a \models_M f \Leftrightarrow \chi_f \leq a$, for all $a \in J(M)$.

Thus, we can view χ_f as the defining trajectory of f and it is also safe to say that every trajectory formula is simple.

More precisely, a trajectory assertion is defined as [*Ante* $\rightarrow$ *Cons*] with *dep*(*Ante*) = *dep*(*Cons*), where *Ante* and *Cons* are trajectory formulas and *dep*(*Ante*) and *dep*(*Cons*) are the depths of formulas *Ante* and *Cons* Respectively. Generally, the depth of a formula *f*, denoted as *dep*(*f*), can be defined recursively as below:

- $dep(\rho) = 1$ if $\rho \in \mathrm{P}$ is a simple predicate,
- $dep(f_1 \wedge f_2) = \max(dep(f_1), dep(f_2))$,
- $dep(b \rightarrow f) = dep(f)$, and
- $dep(\mathrm{N}f) = 1 + dep(f)$.

If all the trajectories of model *M* satisfy a trajectory assertion [*Ante* $\rightarrow$ *Cons*] of the same model, we write $\models_M$ [*Ante* $\rightarrow$ *Cons*], where the satisfaction relation is defined as follows:

$\models_M$ [*Ante* $\rightarrow$ *Cons*] holds iff $a \models_M$ *Ante* implies $a \models_M$ *Cons* for all $a \in J(M)$.

Finally, with the methods established for constructing the defining sequence and the defining trajectory for a given trajectory formula, we can apply the theorem below [4] to achieve our verification goal:

Theorem 1: Given χ_{Ante} and β_{Cons} as the defining trajectory and the defining sequence of trajectory formulas *Ante* and *Cons* in model *M*, respectively,

$\models_M$ [*Ante* $\rightarrow$ *Cons*] holds iff $\beta_{Cons} \leq \chi_{Ante}$.

Note that although both the defining sequence and the defining trajectory are infinite by definition, we need only to compute the bounded prefix of them, since it is easy to show that for a given trajectory f with the defining sequence $\beta_f = \beta_f^0 \beta_f^1 \ldots$ we have $\beta_f^i = \perp$ for $i \geq dep(f)$.

2.5 Symbolic Formulation

In this section, we will first introduce symbolic methods for representing trajectory formulas and trajectory assertions and then we will see how to verify these assertions using symbolic simulation. With these symbolic extensions, we can effectively reduce the number of required test cases and simulation runs.

Several ways can be used to realize the symbolization of trajectory formulas and what we present here is the one described in [4].

Definition 2.5.1: Given a model $M = [(S, \leq), Suc]$, a set U of Boolean variables, and a set P of simple predicates over S, the recursive definition of a symbolic trajectory formula of M is as follows:

- A simple predicate $\rho \in \mathrm{P}$ is a symbolic trajectory formula of M.
- The conjunction $(f_1^s \wedge f_2^s)$ is a symbolic trajectory formula if both f_1^s and f_2^s are symbolic trajectory formulas of M.
- The domain restriction $(B \rightarrow f^s)$ is a symbolic trajectory formula if B is a Boolean function over U and f^s is a symbolic trajectory formula of M.
- The next time expression $(\mathbf{N} f^s)$ is a symbolic trajectory formula if f^s is a symbolic trajectory formula of M and **N** is the next-time operator.

As we can see from the above definition, the only modification from the original definition of trajectory formula is in the domain restriction part, where the domain constraint is generalized from a Boolean constant to a Boolean function.

The definition of a symbolic trajectory assertion (shown as below) can then be easily developed by simply replacing the trajectory formulas with the symbolized ones:

A symbolic trajectory assertion is of form $[Ante^s \rightarrow Cons^s]$ with $dep(Ante^s) = dep(Cons^s)$, where $Ante^s$ and $Cons^s$ are symbolic trajectory formulas and $dep(f^s)$ denotes the depth of a symbolic formula f^s.

For an assignment $\eta : U \rightarrow \{0, 1\}$ to the Boolean variables in a given symbolic trajectory formula f^s of model $M = [(S, \leq), Suc]$, the evaluation of f^s denoted by $f^s(\eta)$ is defined recursively as follows:

- $\rho(\eta) = \rho$, if $\rho \in \mathrm{P}$ is a simple predicate over S.
- $(f_1^s \wedge f_2^s)(\eta) = f_1^s(\eta) \wedge f_2^s(\eta)$, if both f_1^s and f_2^s are symbolic trajectory formulas of M.
- $(B \rightarrow f^s)(\eta) = B(\eta) \rightarrow f^s(\eta)$, if B is a Boolean function over U and f^s is a symbolic trajectory formula of M.
- $(\mathrm{N}f^s)(\eta) = (\mathrm{N}(f^s(\eta)))$, if f^s is a symbolic trajectory formula of M and **N** is the next-time operator.

Accordingly, the evaluation of a given symbolic trajectory assertion $[Ante^s \rightarrow Cons^s]$ of the same model for the assignment $\eta : U \rightarrow \{0, 1\}$, denoted by $[Ante^s \rightarrow Cons^s](\eta)$, is defined as:

$$[Ante^s \rightarrow Cons^s](\eta) = [Ante^s(\eta) \rightarrow Cons^s(\eta)].$$

In the rest of this section, we will show how to verify these symbolic trajectory assertions using symbolic simulation. A symbolic trajectory evaluation algorithm can be easily developed by symbolically extending the functions and relations used in the scalar trajectory evaluation algorithm discussed in the previous section.

Let H be the set of all assignments to the Boolean variable set U, i.e., $\mathrm{H} = \{\eta \mid \eta : U \rightarrow \{0,1\}\}$. The state set $S = \{0, 1, \mathrm{X}\}^n \cup \{\mathrm{T}\}$ is extended to a symbolic state set $S(U) = \{g \mid g : \mathrm{H} \rightarrow S\}$. Each symbolic state in $S(U)$ is a function mapping a Boolean assignment $\eta \in \mathrm{H}$ to a vector of ternary values (a scalar state) in S. For any given state $s \in S$, we let s^c denotes the constant function that has $s^c(\eta) = s$ for any assignment $\eta \in \mathrm{H}$. Particularly, $\perp^c$ denotes the constant function for the state $\perp$. The next state function *Suc*: $S \rightarrow S$ is then extended to the symbolic next state function $Suc^s : S(U) \rightarrow S(U)$. The lowest upper bound function *lub*, and the partial order $\leq$ are extended to their symbolic counterparts lub^s and $\leq^s$, respectively.

Note that in order to apply BDD technique in the symbolic trajectory evaluation algorithm, we need to represent the state space as Boolean functions. Thus, we treat the symbolic state in $S(U)$ bit by bit. Let X^c, 1^c, and 0^c be the constant functions for value X, 1 and 0, respectively. Each bit of the symbolic state can be a constant function $X^c/1^c/0^c$, a Boolean function, or a "mux" function taking either of them as the result by the control of another Boolean function. Actually this "mux" function can also be viewed as a Boolean function with a constant $X^c/1^c/0^c$.

Definition 2.5.2: Given a model $M = [(S, \leq), Suc]$ and a set P of simple predicates over S, the defining symbolic sequence of a given symbolic trajectory formula f^s of model M, denoted by $\beta^s_{f^s}$, can be defined as follows:

- $\beta^s_\rho = d^c_\rho \perp^c \perp^c \ldots$ if d_ρ is the defining value of $\rho \in \mathrm{P}$.
- $\beta^s_{f_1^s \wedge f_2^s} = \mathrm{lub}^s(\beta^s_{f_1^s}, \beta^s_{f_2^s})$, where lub^s is the symbolic extension of lub.

- $\beta^s_{B\to f^s} = B?^s \beta^s_{f^s}$, where B is a Boolean function and $?^s$ is the symbolic extension of $?$.
- $\beta^s_{\mathrm{N}f^s} = \perp^c \beta^s_{f^s}$.

The evaluation of the defining symbolic sequence $\beta^s_{f^s}$ for any assignment $\eta \in \mathrm{H}$, denoted by $\beta^s_{f^s}(\eta)$, is $\beta^s_{f^s}(\eta) = \beta^s_{f^s(\eta)}$.

Definition 2.5.3: Given any symbolic trajectory formula f^s of model $M = [(S, \leq), Suc]$, assuming that $\beta^s_{f^s} = \beta^{s0}_{f^s}\beta^{s1}_{f^s}\ldots$ is the defining sequence for f^s, the defining symbolic trajectory $\chi^s_{f^s} = \chi^{s0}_{f^s}\chi^{s1}_{f^s}\ldots$ can be defined inductively as follows:

$$\chi^{si}_{f^s} = \begin{cases} \beta^{s0}_{f^s} & \text{if } i = 0 \\ \mathrm{lub}^s(\beta^{si}_{f^s}, Suc^s(\chi^{s(i-1)}_{f^s})) & \text{otherwise} \end{cases}.$$

The evaluation of the defining symbolic trajectory $\chi^s_{f^s}$ for any assignment $\eta \in \mathrm{H}$, denoted by $\chi^s_{f^s}(\eta)$, is as below:

$$\chi^s_{f^s}(\eta) = \chi^s_{f^s(\eta)}.$$

We can also extend the satisfaction relations symbolically for a symbolic trajectory formula and for a symbolic trajectory assertion, respectively.

Given a symbolic trajectory formula f^s of model M, which is satisfied by a trajectory $a = (a_0, a_1, a_2, \ldots)$ of the same model, we define the symbolic satisfaction relation $\models^s_M$ for the assignment $\eta : U \to \{0, 1\}$ as:

$$(a \models^s_M f^s)(\eta) = 1 \text{ iff } a \models_M f^s(\eta).$$

Similarly, for a symbolic trajectory assertion, we have:

$$(\models^{s}_{M}[Ante^{s} \rightarrow Cons^{s}])(\eta) = 1 \text{ iff } \models_{M}([Ante^{s} \rightarrow Cons^{s}](\eta)).$$

At last, we can apply the theorem below [4] to verify if a symbolic trajectory assertion is satisfied by a model M:

Theorem 2: Given $\chi^{s}_{Ante^{s}}$ and $\beta^{s}_{Cons^{s}}$ as the defining symbolic trajectory and the defining symbolic sequence of symbolic trajectory formulas $Ante^{s}$ and $Cons^{s}$ in model M, respectively, for every $\eta \in \mathrm{H}$,

$$\models^{s}_{M}[Ante^{s} \rightarrow Cons^{s}](\eta) = 1 \text{ iff } \beta^{s}_{Cons^{s}}(\eta) \leq \chi^{s}_{Ante^{s}}(\eta).$$

For a given model M and a symbolic trajectory assertion of it, the symbolic evaluation algorithm yields a Boolean function denoting the set of assignments under which the assertion is satisfied as compared with the simple yes/no answer from the scalar algorithm. Since the verification task requires that the assertion should hold under all variable assignments, this Boolean function should simply be the constant function 1^{c}, i.e., the function that returns 1 for all assignments.

2.6 Illustrative Example

In this section, we will present an illustrative example for Symbolic Trajectory Evaluation.

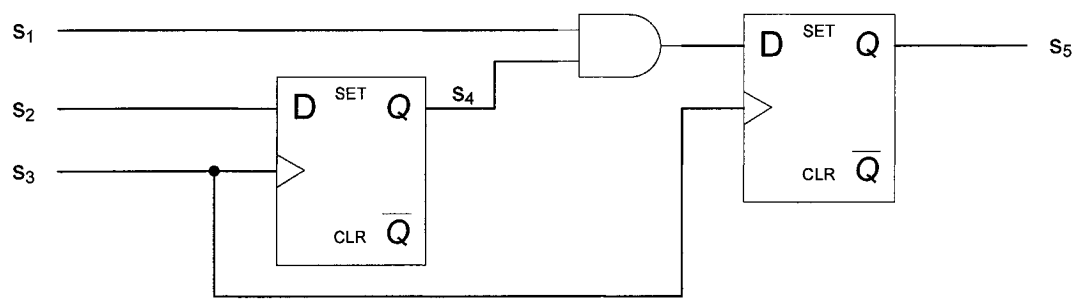


Figure 13. Diagram for a Verilog model of a sequential circuit

Consider the Verilog model of a sequential circuit shown in Figure 13. Signals s_1 and s_2 are two inputs to the circuit. The other input s_3 severs as a clock signal. Signals s_4 registers s_2 at the positive edge of clock s_3. Finally, signal s_5 is the registered output of the circuit, the value of which equals $s_1 \wedge s_4$. The time unit we use here is half clock cycle and we assume there is no time delay at the and gate.

We represent the circuit state as a 5-bit vector $s = s_1 s_2 s_3 s_4 s_5$. The next state function $Suc^s = t_1 t_2 t_3 t_4 t_5$ is defined as follows:

$$t_1(s_1) = X^c, t_2(s_2) = X^c, t_3(s_3) = X^c, t_4(s_4) = s_3 s_2 + \overline{s_3} s_4, t_5(s_5) = s_3 s_1 s_4 + \overline{s_3} s_5 .$$

As mentioned before, no next state constraint is imposed on a state vector component associated with an input signal, that is, its next state function should be the constant function for value X.

Assume that we want to verify the following symbolic trajectory assertion for the above circuit model:

$$\begin{array}{l}((s_1 = a)^{\{4\}} \wedge (s_2 = b)^{\{2\}} \wedge (s_3 = 0) \wedge \mathrm{N}(s_3 = 1) \wedge \mathrm{N}^2(s_3 = 0) \wedge \mathrm{N}^3(s_3 = 1)) \\ \rightarrow (\mathrm{N}^4(s_5 = ab))\end{array},$$

where the expression $(s_1 = a)$ stands for the symbolic trajectory formula $(a \rightarrow (s_1 = 1)) \wedge (\overline{a} \rightarrow (s_1 = 0))$, $(s_2 = b)^{\{2\}}$ denotes $(s_2 = b) \wedge \mathrm{N}(s_2 = b)$, and N^i denotes i next time operators. Generally, we use $(s_i = B)$ as the shorthand for the symbolic formula $(B \rightarrow (s_i = 1)) \wedge (\overline{B} \rightarrow (s_i = 0))$, and represent the symbolic formula $f^s \wedge \mathrm{N}f^s \wedge \mathrm{N}^2 f^s \ldots \wedge \mathrm{N}^{i-1} f^s$ as $(f^s)^{\{i\}}$, where f^s is a symbolic trajectory formula.

The simple predicates involved in the above assertion and the corresponding defining values are listed below:

- $s_1 = 0$ with defining value $\langle 0, X, X, X, X\rangle$,
- $s_1 = 1$ with defining value $\langle 1, X, X, X, X\rangle$,
- $s_2 = 0$ with defining value $\langle X, 0, X, X, X\rangle$,
- $s_2 = 1$ with defining value $\langle X, 1, X, X, X\rangle$,
- $s_3 = 0$ with defining value $\langle X, X, 0, X, X\rangle$,
- $s_3 = 1$ with defining value $\langle X, X, 1, X, X\rangle$,
- $s_5 = 0$ with defining value $\langle X, X, X, X, 0\rangle$,
- $s_5 = 1$ with defining value $\langle X, X, X, X, 1\rangle$.

From Definition 7, we get the defining symbolic sequence of symbolic trajectory formula $(s_1 = a)$, a shorthand expression for $(a \rightarrow (s_1 = 1)) \wedge (\bar{a} \rightarrow (s_1 = 0))$, as:

$$\beta^s_{((a\rightarrow(s_1=1))\wedge(\bar{a}\rightarrow(s_1=0)))} = \langle a, X^c, X^c, X^c, X^c\rangle\langle X^c, X^c, X^c, X^c, X^c\rangle\langle X^c, X^c, X^c, X^c, X^c\rangle\ldots .$$

Similarly, we get the defining symbolic sequence of $(s_2 = b)$ as:

$$\beta^s_{((b\rightarrow(s_2=1))\wedge(\bar{b}\rightarrow(s_2=0)))} = \langle X^c, b, X^c, X^c, X^c\rangle\langle X^c, X^c, X^c, X^c, X^c\rangle\langle X^c, X^c, X^c, X^c, X^c\rangle\ldots .$$

Then the defining symbolic sequence of $(s_1 = a) \wedge (s_2 = b)$ is computed as:

$$\begin{aligned}&\beta^s_{(s_1=a)\wedge(s_2=b)} = \mathrm{lub}^s(\beta^s_{(s_1=a)}, \beta^s_{(s_2=b)})\\ &= \langle a, b, X^c, X^c, X^c\rangle\langle X^c, X^c, X^c, X^c, X^c\rangle\langle X^c, X^c, X^c, X^c, X^c\rangle\ldots\end{aligned}$$

By recursively applying Definition 7, we can get the defining symbolic sequence of the antecedent:

$$Ante^s = ((s_1 = a)^{\{4\}} \wedge (s_2 = b)^{\{2\}} \wedge (s_3 = 0^c) \wedge \mathrm{N}(s_3 = 1^c) \wedge \mathrm{N}^2(s_3 = 0^c) \wedge \mathrm{N}^3(s_3 = 1^c)) \quad ,$$

and then compute its defining symbolic trajectory according to Definition 8, the procedure of which are shown in Table I.

TABLE I. DEFINING SYMBOLIC TRAJECTORY OF THE ANTECEDENT

i	$\beta^{si}_{Ante^s}$					$Suc^s(\chi^{s(i-1)}_{Ante^s})$						$\chi^{si}_{Ante^s}$				
	s_1	s_2	s_3	s_4	s_5	s_1	s_2	s_3	s_4	s_5		s_1	s_2	s_3	s_4	s_5
0	a	b	0^c	X^c	X^c							a	b	0^c	X^c	X^c
1	a	b	1^c	X^c	X^c	X^c	X^c	X^c	X^c	X^c	lub^s	a	b	1^c	X^c	X^c
2	a	X^c	0^c	X^c	X^c	X^c	X^c	X^c	b	aX^c	lub^s	a	X^c	0^c	b	aX^c
3	a	X^c	1^c	X^c	X^c	X^c	X^c	X^c	b	aX^c	lub^s	a	X^c	1^c	b	aX^c
4	X^c	X^c	X^c	X^c	X^c	X^c	X^c	X^c	X^c	ab	lub^s	X^c	X^c	X^c	X^c	ab
≥5	X^c	X^c	X^c	X^c	X^c	X^c	X^c	X^c	X^c	abX^c	lub^s	X^c	X^c	X^c	X^c	abX^c

TABLE II. SYMBOLIC DEFINING SEQUENCE OF THE CONSEQUENT AND COMPARISON

i	$\beta^{si}_{Cons^s}$						$\chi^{si}_{Ante^s}$				
	s_1	s_2	s_3	s_4	s_5		s_1	s_2	s_3	s_4	s_5
0	X^c	X^c	X^c	X^c	X^c	$\leq^s$	a	b	0^c	X^c	X^c
1	X^c	X^c	X^c	X^c	X^c	$\leq^s$	a	b	1^c	X^c	X^c
2	X^c	X^c	X^c	X^c	X^c	$\leq^s$	a	X^c	0^c	b	aX^c
3	X^c	X^c	X^c	X^c	X^c	$\leq^s$	a	X^c	1^c	b	aX^c
4	X^c	X^c	X^c	X^c	ab	$\leq^s$	X^c	X^c	X^c	X^c	ab
≥5	X^c	X^c	X^c	X^c	X^c	$\leq^s$	X^c	X^c	X^c	X^c	abX^c

Similarly, we can also obtain the symbolic defining sequence of the consequent $Cons^s = (\mathrm{N}^4(s_5 = ab))$ shown in Table II, compared with the result for the defining trajectory of the antecedent. We can easily see from the table that: $\beta^s_{Cons^s} \leq^s \chi^s_{Ante^s} = 1^c$, i.e., the symbolic trajectory assertion is satisfied by the circuit model under all variable assignments.

2.7 STE Based Verification Tool and FL Language

Forte is Intel's custom-built verification environment, evolved from Carl Seger's VOSS formal hardware verification system. Forte integrates model-checking engines (STE), BDDs, circuit manipulation functions, theorem proving, and a functional programming language called FL. FL is used in Forte as a programming language for application development and fast prototyping, and also as an extension language for users to enable writing flows and applications [32].

Devised by Carl Seger during the years 1990-1995, FL is a strongly typed, lazy, functional programming language [33]. Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions, which are often defined by separation into various cases, each of which is separately defined by appealing (possibly recursively) to function applications [19]. In contrast to imperative programming, functional programming emphasizes the evaluation of functional expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values. Functional programming languages have no variables, no assignment statements, and no iterative constructs. The oldest example of a functional language is Lisp, though not a purely functional programming language, which introduced many of the features now found in modern functional programming languages. The modern canonical examples are Haskell and members of the ML family including SML and Ocaml. FL is syntactically very similar to Edinburgh ML (Meta Language), and semantically closely related to lazy-ML and Haskell.

What distinguish FL from other functional languages are the following VLSI CAD-related features [32]:

- BDDs fully integrated into the language with every object of type 'bool' represented as a BDD,
- VLSI modeling capability, and
- STE, a C based symbolic simulator, integrated into the language.

FL provides a flexible interface for invoking and orchestrating model checking runs and serves as an extensible 'macro language' for expressing specifications, which makes Forte a generic, open framework where solutions can be tailored to individual verification problems [32].

The model to be verified in Forte must be in Exlif format [31], where the RTL design is flattened to the gate level netlist. It was necessary to develop a converter which can translate the Verilog RTL to Exlif format. The high level description of this translation is illustrated in Figure 14. The Verilog code is first translated to Blif-mv format using the VIS tool, and then we used a translator to convert the Blif-mv format file to an Exlif one. This Blif-mv to Exlif translator was developed using Perl script by our group.

The converting is a straight forward process since the two formats Exlif and Blif-mv are similar modulo certain syntactic differences. Therefore, it is safe to say that the correctness of this translation is guaranteed by the VIS tool.

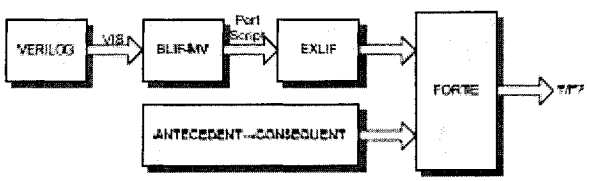


Figure 14. Verification using Forte

The properties of the design are captured by STE assertions generated in FL codes.

In the Forte environment, the syntax for STE invocation is:

STE <model> <weak> <antecedent> <consequence> <trace>

where the description of each object in the above syntax is given below:

• <model>: This is the model to be simulated. The model is an object of type fsm.

• <weak>: This is a list of 4-tuples of the following format: (<guard>, <node>, <from>, <to>), where <guard> is of type bool, <node> is a node name, <from> and <to> are integers. The semantics of such a tuple is such that if the condition <guard> holds, the given <node> should be disconnected from the logic driving it in the model from time <from> to time <to>. The <weak> list is usually used to solve the contradiction in node assignment.

• <antecedent>: The antecedent is the input vectors to the simulator. It is a list of 5-tuples of the following format: (<guard>, <node>, <value>, <from>, <to>), where <value> is of type bool. The semantics of such a tuple is that if the condition <guard> holds, the given <node> is assigned the value <value> from time <from> to time <to>.

• <consequence>: The <consequence> describes the expected result (consequence) of the simulation. It is a list of 5-tuples of following format: (<guard>, <node>, <value>, <from>, <to>). The semantics of such a tuple is that when the condition <guard> is T then after running STE, the given node is expected to be equal to the given value from time <from> to time <to>.

• <trace>: This is a list of triples of the following format: (<node>, <from>, <to>). The semantics of such a triple is that STE records the value of the given node from time <from> to time <to>.

After running the STE simulation, STE returns Boolean value T if the consequence is realized, and returns the Boolean condition under which the consequence is realized otherwise. If the consequence is never realized, STE returns Boolean value F.

Time-frame specified in <weak>, <antecedent>, <consequence> and <trace> is in terms of the internal clock of STE that progresses tick by tick. Model clocks and model timing should be translated in terms of the STE clock.

2.8 Summary

In this chapter, we presented the theory and methodology of Symbolic Trajectory Evaluation and introduce the verification tool for STE. Then, in the following Chapter, we will further investigate this technique by a case study of verifying the Look-Aside Interface using STE.

Chapter 3

Verifying Look-Aside Interface using STE

In this chapter, we first present a brief introduction to the Look-Aside Interface (LA-1). We then discuss some related work including a previous RTL design for the LA-1 interface. A modified RTL design for the LA-1 interface is described in detail after that. Finally, the verification processes of both the previous RTL design and the modified LA-1 interface RTL design using STE are illustrated.

3.1 LA-1 Interface Specifications

The LA-1 interface [34], developed by the Network Processor Forum, is a memory-mapped interface based on QDR (Quad Data Rate) SRAM, targeted at devices (memories or coprocessors) that offload certain tasks from a network processing unit (NPU). The major features of the LA-1 interface include:

- Concurrent read and write.
- Separate unidirectional read and write data buses.
- Single address bus.
- 18-bit DDR data output bus transfers 32 bits plus 4 bits of data parity per read.
- 18-bit DDR data input bus transfers 32 bits plus 4 bits of data parity per write.

- Byte write control for writes.

3.1.1 Signal Descriptions

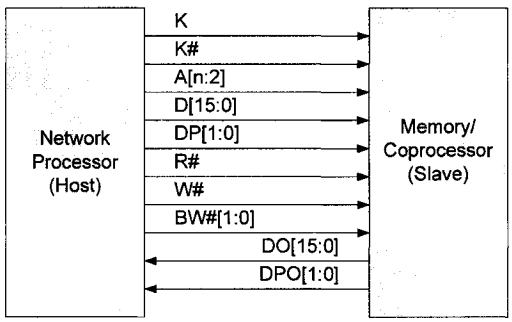


Figure 15. LA-1 Interface buses

The LA-1 interface transfers data between an NPU and memory or coprocessors. Figure 15 shows the LA-1 interface bus signals. One LA-1 port consists of two input clocks (K and K#), which are rising-edge active and should be ideally 180 degrees out of phase with each other, one active-low write select input W#, one active-low read select input R#, 2-bit active-low byte-write inputs BW#[1:0], single address bus A, 16-bit synchronous data inputs D[15:0] plus 2-bit synchronous data parity inputs DP[1:0] for write operations, and 16-bit synchronous data outputs DO[15:0] plus 2-bit synchronous data parity outputs DPO[1:0] for reads.

3.1.2 Port Operation specifications

3.1.2.1 Write Operations

A write cycle is initiated by asserting W# low at the rising edge of clock K.

The write address should be ready at the following rising edge of K# and data is captured at the rising edge of K and K# in the same cycle.

3.1.2.2 Read Operations

A read cycle is initiated by asserting R# low at the rising edge of K and the read address is captured at the same edge. Output data is delivered after the next rising edge of K.

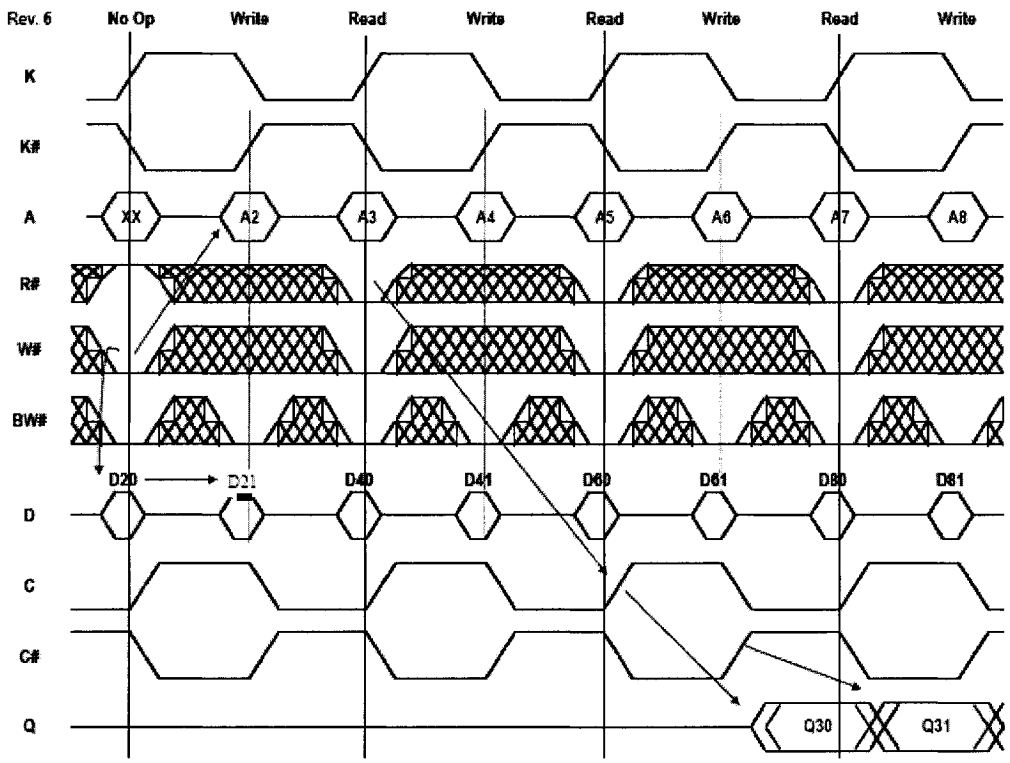


Figure 16. LA-1 port operation timing diagram

The timing diagram for the port operation of the LA-1 interface is shown in Figure 16 [34].

3.2 Related Work

The LA-1 interface was first verified by A.Habibi et al. [1] at both the behavioral level and the RTL level. That work, to our knowledge, includes behavioral designs in Abstract State Machine (ASM) and SystemC, a RTL design in Verilog and the corresponding verification approaches. The ASM level LA-1 design was verified using the AsmL tool to model check a set of properties in Property Specification Language (PSL). The SystemC level model was verified using simulation to perform Assertion-Based Verification (ABV) of properties expressed in C# assertions. The verification of the Verilog RTL LA-1 design was performed using the RuleBase tool to model check PSL properties.

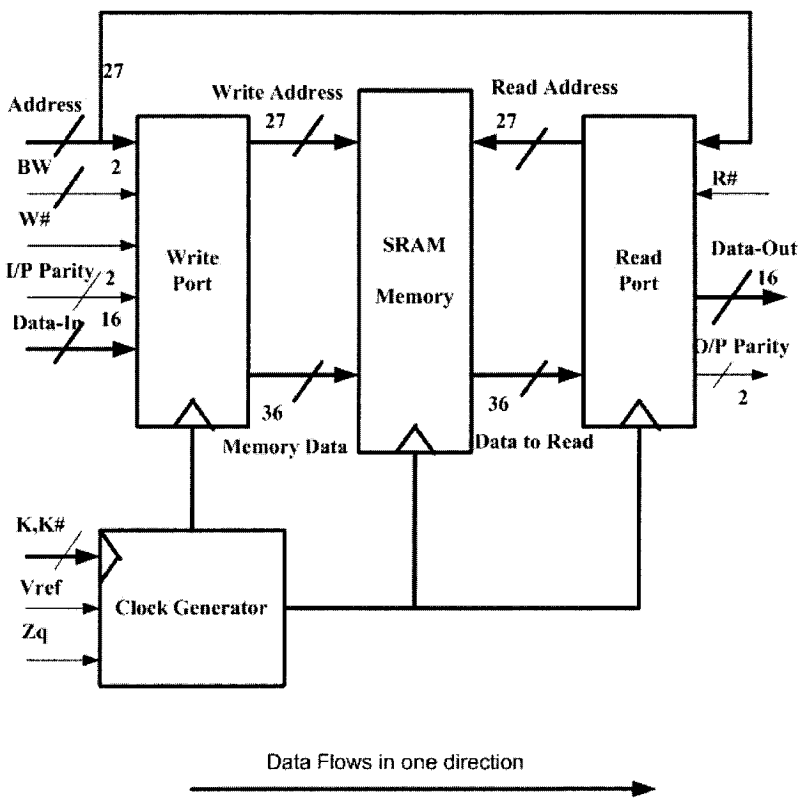


Figure 17. Archtecture of Ahmed's LA-1 RTL design

3.3 Verifying Ahmed's LA-1 RTL Design using STE

3.3.1 Design

A synthesizable RTL design for the LA-1 interface was implemented in Verilog by A. I. Ahmed et al. [3] conforming to the above specifications. The architecture of the LA-1 interface RTL design is shown in Figure 17 [3]. Three main building blocks are used in this LA-1 RTL design: Write port, Read port and Memory. Notice that the memory data bus width is 36 bits. The timing diagrams for each of the three blocks are shown in Figure 18, 19 and 20 [3], respectively. Note that clock K and K# in the LA-1 interface specifications are represented by CLK_K and CLK_K1 respectively in the design.

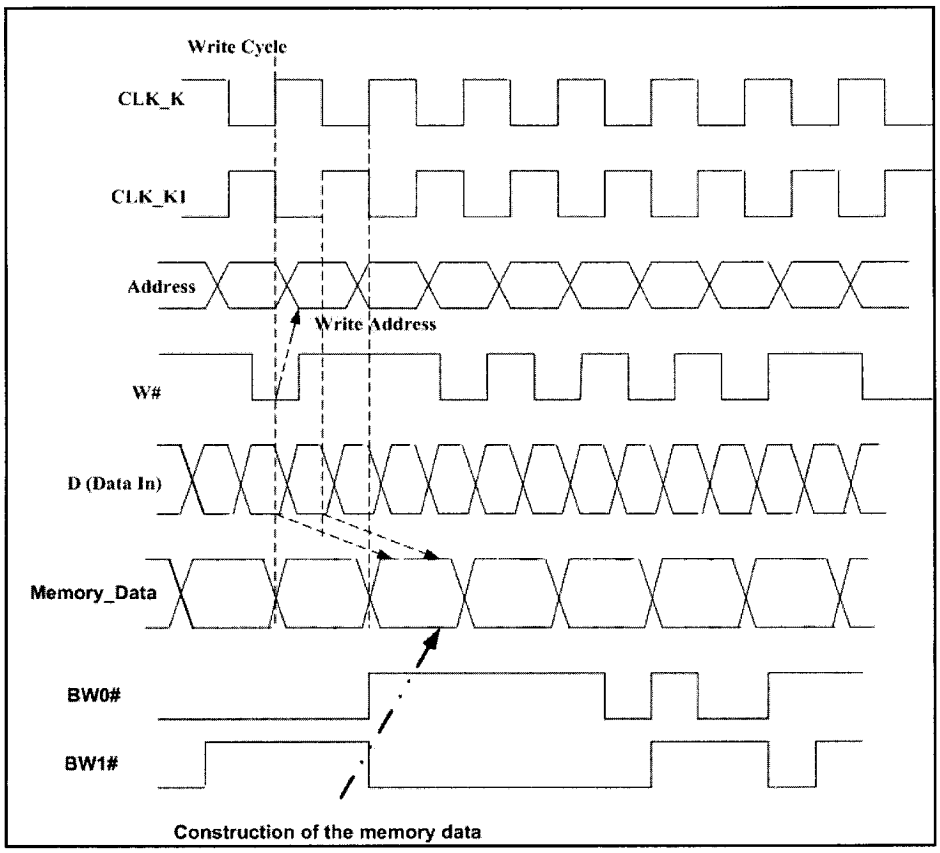


Figure 18. Timing diagram for the LA-1 Interface Write Port

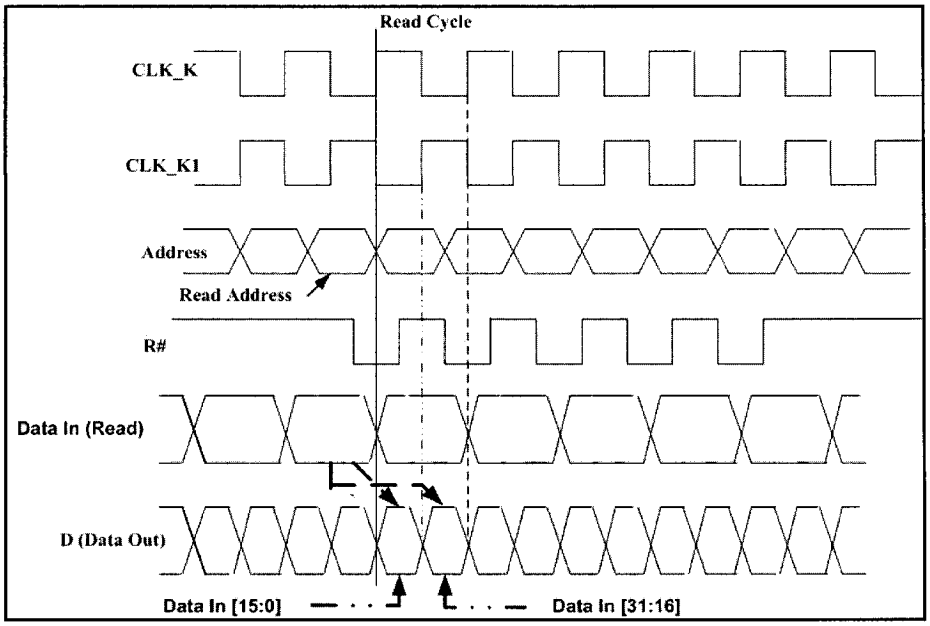


Figure 19. Timing diagram for the LA-1 Interface Read Port

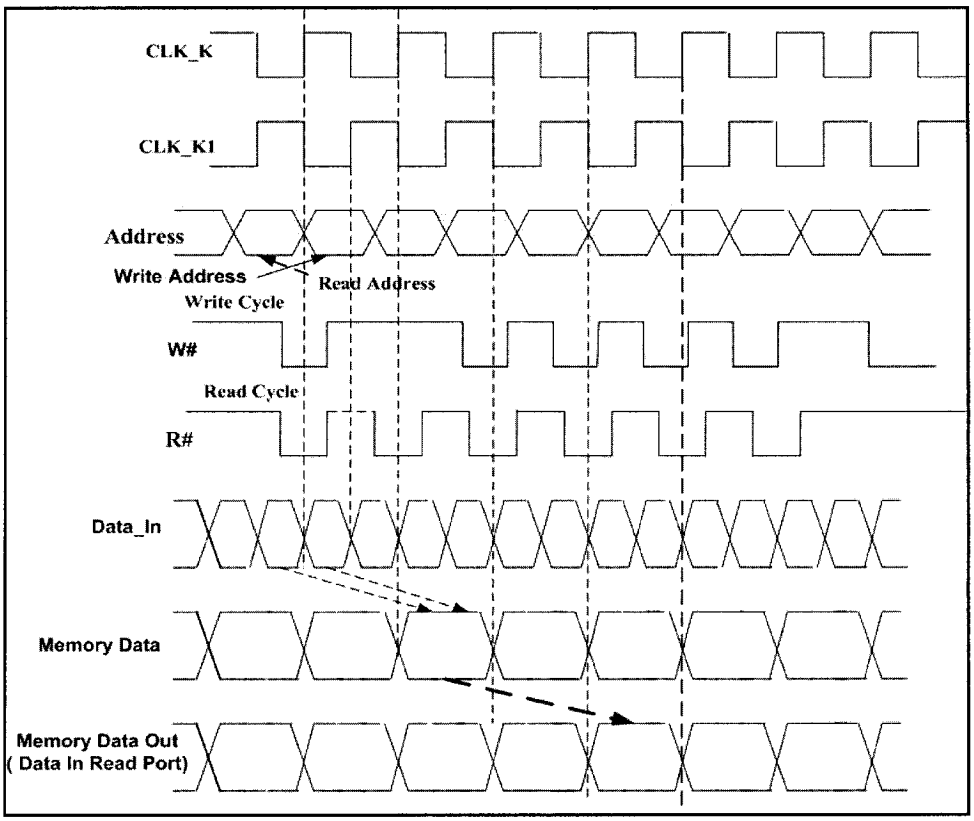


Figure 20. Timing diagram for the LA-1 Interface Memory

The following properties are extracted of the design specifications:

- **Property 1** (Write Port): by asserting *W#* low at the rising edge of *CLK_K*, if the byte-write control inputs *BW#1* and *BW0#* are set to low, the full data input *D* will be captured, at the same cycle, at the rising edge of *CLK_K* and *CLK_K1* and sent to the memory through *Memory_data* at the next rising edge of *CLK_K*.
- **Property 2** (Write Port): by asserting *W#* low at the rising edge of *CLK_K*, the active-low memory enable signal will be set to low at the next rising edge of *CLK_K*.
- **Property 3** (Read Port): by asserting R# low at the rising edge of *CLK_K*, the data from the memory *Data_In* will be sent out sequentially two times with half a clock cycle in between through *D* after the next rising edge of *CLK_K* .
- **Property 4** (Memory Port): the data written to a specific address of the memory by a previous write operation can be read out properly by a read operation to the same memory location, provided that there is at least a two clock cycle delay between the write operation and the read operation.

3.3.2 Verification

Now, we apply the STE model checking technique presented in a previous chapter to verify the LA-1 RTL Design of A. I. Ahmed et al. using Forte. As mentioned previously, we need the VIS tool and a Perl script to convert the RTL design from Verilog to Exlif, the only format accepted by Forte. One limitation of this converting process is that VIS does not support multiple clocks. Hence, one of the two clocks must be removed from the design. We solved the problem by substituting the use of the rising edge of clock

CLK_K1 (K#) for the use of the negative edge of the clock CLK_K (K), based on the fact that CLK_K1 and CLK_K are ideally 180 degrees out of phase with each other. Thus we have only one clock (CLK_K) left in the resulting Exlif format model.

The STE formulations of the four properties described above are given below:

- **STE Assertion 1** for **Property 1**:

$$
\begin{aligned}
&((CLK\_K = 0) \wedge (W\# = 0) \wedge (BW0\# = 0) \wedge (BW1\# = 0) \wedge (DI[15:0] = d1[15:0])) \\
&\wedge \mathrm{N}((CLK\_K = 1) \wedge (DI[15:0] = d2[15:0])) \\
&\wedge \mathrm{N}^2(CLK\_K = 0) \wedge \mathrm{N}^3(CLK\_K = 1) \\
&\rightarrow \mathrm{N}^3((Memory\_Data[31:16] = d1[15:0]) \wedge (Memory\_Data[15:0] = d2[15:0])).
\end{aligned}
$$

- **STE Assertion 2** for **Property 2**:

$$
\begin{aligned}
&((CLK\_K = 0) \wedge (W\# = 0) \\
&\wedge \mathrm{N}(CLK\_K = 1) \wedge \mathrm{N}^2(CLK\_K = 0) \wedge \mathrm{N}^3(CLK\_K = 1) \\
&\rightarrow \mathrm{N}^3(me = 1).
\end{aligned}
$$

- **STE Assertion 3** for **Property 3**:

$$
\begin{aligned}
&((CLK\_K = 0) \wedge (R\# = 0)) \\
&\wedge \mathrm{N}(CLK\_K = 1) \wedge \mathrm{N}^2(CLK\_K = 0) \\
&\wedge \mathrm{N}^3((CLK\_K = 1) \wedge (Data\_In[35:0] = d1[35:0])) \wedge \mathrm{N}^4(CLK\_K = 0) \\
&\wedge \mathrm{N}^5(CLK\_K = 1) \\
&\rightarrow \mathrm{N}^4(DO[15:0] = d1[15:0]) \wedge \mathrm{N}^5(DO[15:0] = d1[31:16]).
\end{aligned}
$$

- **STE Assertion 4** for **Property 4**:

$$
\begin{aligned}
&((CLK\_K = 0) \wedge (W\# = 0) \wedge (R\# = 0) \wedge (BW0\# = 0) \wedge (BW1\# = 0) \\
&\wedge (DI[15:0] = d1[15:0]) \wedge (Address[26:0] = a1[26:0])) \\
&\wedge \mathrm{N}((CLK\_K = 1) \wedge (DI[15:0] = d2[15:0])) \\
&\wedge \mathrm{N}^2(CLK\_K = 0) \wedge \mathrm{N}^3(CLK\_K = 1) \wedge \mathrm{N}^4(CLK\_K = 0) \wedge \mathrm{N}^5(CLK\_K = 1) \\
&\wedge \mathrm{N}^6(CLK\_K = 0) \wedge \mathrm{N}^7(CLK\_K = 1) \wedge \mathrm{N}^8(CLK\_K = 0) \wedge \mathrm{N}^9(CLK\_K = 1) \\
&\rightarrow \mathrm{N}^8(DO[31:16] = d1[15:0]). \wedge \mathrm{N}^9(DO[15:0] = d2[15:0]).
\end{aligned}
$$

The FL code used to invocate the STE simulation to verify STE assertion 1 is as follows:

```
1.  let my_ckt = load_exe "LA1 INTERFACE TOP.exe";
2.  let weak = [ ];
3.  let write ant =
4.      (gen_clock_cycles clk F (0 upto 1)) @
5.      (ws is F in_cycles (0 upto 2)) @
6.      (vassign_v_t1_t2 bwe F 0 5) @
7.      (vassign_v_w_t1_t2 din "d1" 15 1 2)
8.  ;
9.  let write cons =
10.     (vassign_v_w_t1_t2 write nodes h "d1" 15 3 4)
11. ;
12. let write tr =
13.     let watch n = (n, 0, 4) in
14.         map watch write_nodes
15. ;
16. STE " " my_ckt weak write_ant write_cons write_tr;
```

The first line of the code is used to load the model of our LA-1 interface design into the Forte system. In line 2, we leave the <weak> list in the STE assertion blank since we suppose that there is no node with contradiction assignment in our model and thus no need to do any disconnection. Line 3 to line 8 specifies the antecedent of the property. Two STE clock cycles, that is four STE clock ticks, are generated by line 4. W#, BW0#

and BW1#, the control inputs, are asserted low respectively for all those two clock cycles in line 5 and 6. The data input is provided in the line right after. The consequence of the property is defined in line 9 to line 11, which gives the expected results of the simulation. Lines 12 to 15 specify a list of nodes which will be traced within specified time ranges. With all these STE invocation object definitions ready, we can run the STE simulation by calling the last line of the above code. This STE simulation will end up with a value T/F to indicate the success/fail of the simulation.

3.3.3 Experimental Results

In this section, we describe our results on the verification of Ahmed's LA-1 RTL design using STE. Table III shows the statistics of verifying the LA-1 Interface design for 4 bits, 6 bits, and 8 bits data width combined with 12 bits and 16 bits address width using Forte. The experiments were done on 2 X UltraSPARC-III+ machine with 2 900Mhz processors and 4096M of RAM. In Figure 21, we can see that the memory usage grows pretty nicely, though not linearly with respect to the width of the address bus. The BDD complexity grows also quite acceptably. The time complexity is not an issue since all the runs for the three different cases took less than one second. However, we could not perform the verification for the full design with 27 address bits and 16 data bits.

TABLE III. STATISTICS FOR AHMED'S LA-1 RTL DESIGN VERIFICATION USING STE

Address Width SRAM Memory	Data Width SRAM Memory	Memory (in MB)	Number of BDD nodes
12	4	18	7491
	8	20	14266
	16	25	29489
16	4	20	12697
	8	22	21578
	16	27	43381

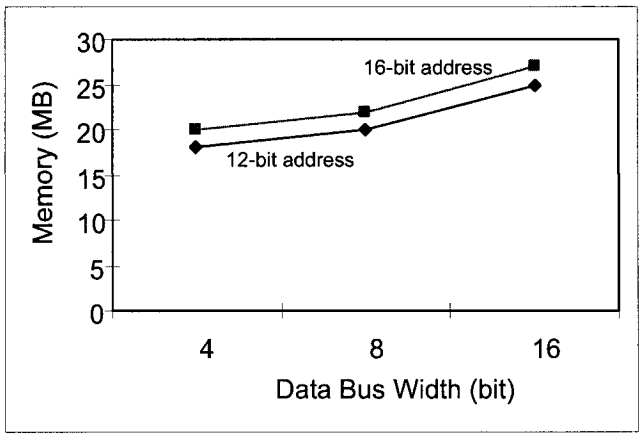


Figure 21. Memory Usage for Ahmed's LA-1 RTL design verification using STE

Our verification did successfully find some bugs for the design which were against the LA-1 Interface specification. Those bugs were fixed in our RTL design. Then the design was converted to Exlif, loaded and checked against all the STE properties again in Forte. Finally all the properties passed for the updated design. The bugs for the read port, for example, are listed below:

- If signal READ_SEL is asserted low all the time and never goes to high, then no data is delivered at all. More specifically, only after signal READ_SEL is asserted high at least once at the rising edge of CLK_K before it is asserted low at the rising edge of CLK_K, this read operation can be recognized and executed.
- Data is delivered one clock cycle earlier than specified with respect to CLK_K.

When an STE run returns a value other than T, it indicates that the consequence is not realized and there should be some unexpected results of the simulation. By looking into the STE return value which may give the Boolean condition under which the consequence is realized or by checking the records of traced nodes, we may find out

where the problem exists and furthermore try to figure out how it can be solved. We can also get some warnings in Forte when something unexpected happens and those warnings can be helpful in problem solving. Here is an example of such a warning message from the STE engine in Forte:

> *Warning: Consequent failure at time 6 on node*
> *DATA_WRITE_MEM_OUT_w< 16 >*
> *Current value:da0*
> *Expected value:db16*

This warning message alerts the user of a consequent failure happened on node *DATA_WRITE_MEM_OUT_w<16>* at time tick 6 during the simulation and the reason for this failure is shown in the second and the third lines. Finally, a Boolean condition is provided under which this consequent failure will hold.

Finally, all the STE assertions listed in the previous sections have been verified in Forte.

3.4 Verifying Modified LA-1 RTL Design using STE

3.4.1 Modifications

The purpose of the modification is to correct some misunderstandings of the design specifications in Ahmed's design presented above and to make the design more general and adaptable to different verification tools.

Modification 1 (Architecture)

After reexamining the LA-1 interface specifications described in Look-Aside (LA-1) Interface Implementation Agreement [34], we found that the Memory unit itself should

not be included as part of the LA-1 interface which works as the interconnection between a network processor (host) and a memory/coprocessor (slave). Thus, we removed the Memory block from our modified LA-1 RTL design.

Modification 2 (Memory interface)

As mentioned in section 4.4, the memory data bus width in the previous design is 36 bits which means that not only the 32 bits of data but also the 4 bits of data parity are stored in the memory. However, from [34] we can see that, during a write operation, the 4 bits of data parity from the host should be only used by the interface to check against the internal generated ones from the 32-bit data from the host and should not be put into the memory; while during a read operation, the 4 bits of data parity to the host should be generated from the 32-bit data stored in the memory by the interface but not be read out directly from the memory. Hence, we reduced the memory data bus width from 36 bits to 32 bits and only the 32-bit data will be written into and read out of the memory.

Modification 3 (Clock frequency)

Two clocks CLK_K (K) and CLK_K1 (K#) are used in the previous design as shown in section 4.4. Then in section 4.5, when verifying the design using STE, we met a problem for the clocks because the VIS tool did not support multiple clocks and we solved the problem by using the double edges of clock CLK_K instead of using both the rising edges of the two clocks and CLK_K became the only clock in the design. Another solution is to generate an internal double-frequency clock *clk_2x* from clock CLK_K and *clk_2x* is also used as the only clock for the LA-1 interface circuit. In this case, the rising edge of CLK_K and CLK_K1 can be obtained by combing *clk_2x* and a control signal *pflag* which is used to indicate the positive edge and negative edge of clock CLK_K. The

second solution is a better choice for our verification purposes since it can not only avoid the use of multiple clocks but also avoid the use of double edges of the clock which is not supported by some verification tools. In fact, we will present another verification methodology in a later chapter for the LA-1 RTL design where the use of double edges of the clock is not allowed. Therefore, we will apply the second solution, that is, the double-frequency clock scheme in our modified design.

3.4.2 Modified Design

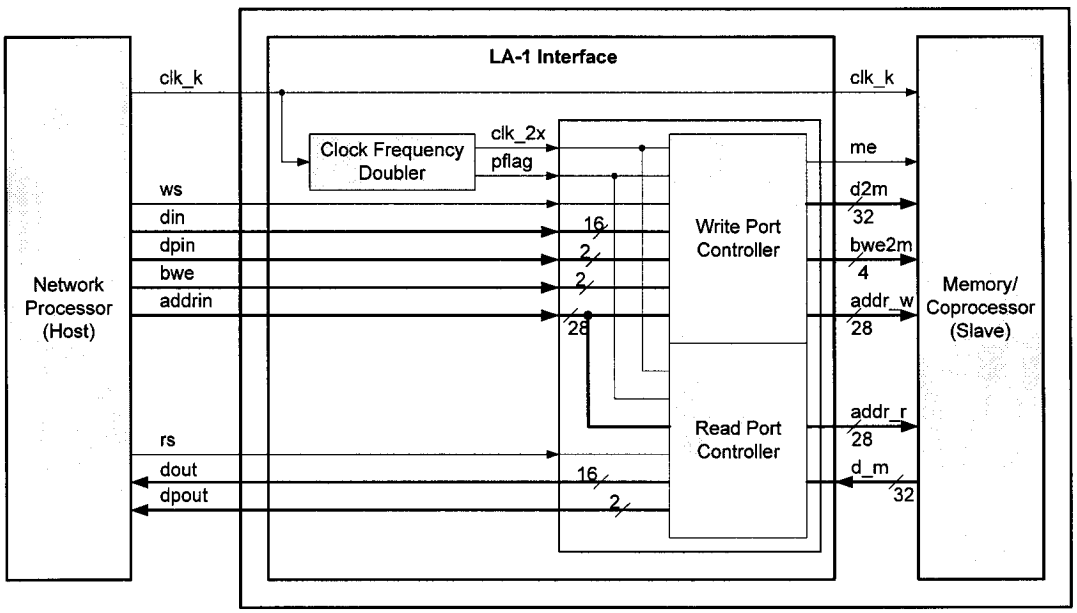


Figure 22. Modified LA-1 RTL design

By applying the modifications presented above to the previous LA-1 RTL design, we got our modified design shown in Figure 22 which is also implemented in Verilog. Note that we also increased the address bus width from 27 bits to 28 bits in the modified design. According to the LA-1 interface specifications [34], the address bus width range is from

22 bits to 28 bits. We used the 28 bits because we wanted to verify the design in the extreme situation. In this design, we use a Clock Frequency Doubler to take the clock input clk_k (CLK_K) and generate an internal double-frequency clock *clk_2x* which is used as the only clock for the LA-1 interface circuit and a control signal *pflag* denoting the positive edge and the negative edge of clock clk_k. The timing for the Write Port and the Read Port are shown in Figure 23 and 24.

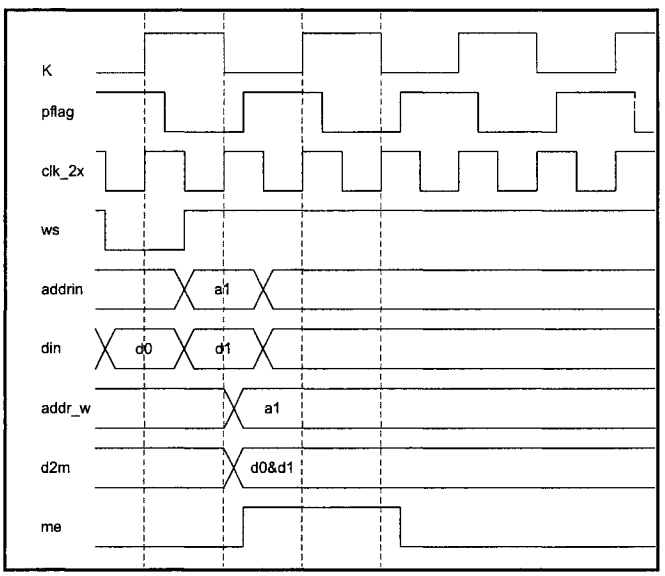


Figure 23. Timing diagrm for Write Port Controller

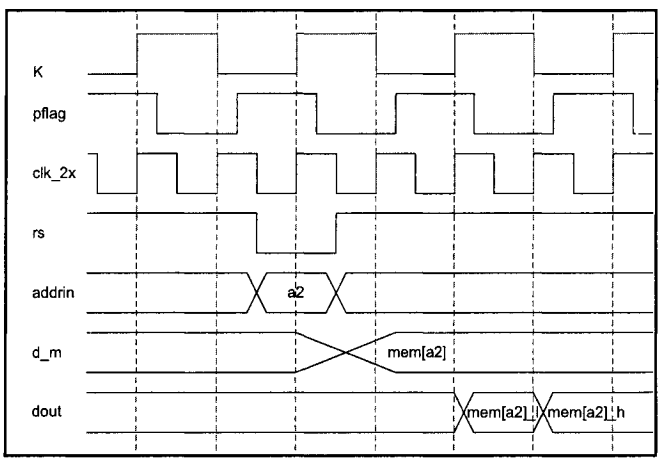


Figure 24. Timing diagram for Read Port Controller

The Clock Frequency Doubler is implemented using the built-in digital delay-locked loop (DLL) of Xilinx Virtex devices. Xilinx Virtex Series DLLs provide precise clock edges through frequency multiplication [37]. The diagram for Virtex DLL is shown in Figure 25 [37]. The Verilog code for the Clock Frequency Doubler is as follows:

```
IBUFG CLK_ibufg_A (
        .I (CLK_K),
        .O(CLK_ibufg) );

BUFG CLK_2x_bufg (
        .I (CLK_2x_t),
        .O (CLK_2x) );

CLKDLL CLK_2x (
        .CLKIN(CLK_ibufg),
        .CLKFB(CLK_2x),
        .RST(1'b0),
        .CLK2X(CLK_2x_t),
        .CLK0(),
        .CLK90(),
        .CLK180(),
        .CLK270(),
        .CLKDV(),
        .LOCKED() );

FLAG_pos = ~CLK_ibufg;
```

where CLKDLL is the DLL component, IBUFG and BUFG are Xilinx global buffer components, signal CLK_K is the input clock, CLK_2x is the output double-frequency clock and FLAG_pos is the output control signal used to denote the edges of CLK_K. The use of global clock buffers can take advantage of the low-skew, high-drive capabilities of the dedicated global buffer tree of Xilinx devices [36].

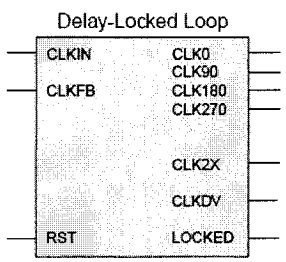


Figure 25. Virtex DLL Block Diagram

Based on the design specification we draw out the following properties:

- **Property 1** (Write Port): by asserting *ws* (W#) low at the rising edge of *clk_2x* when *pflag* is high, if the byte-write control inputs *bwe[1:0]* (BW#[1:0]) are set to low, the full input *data din[15:0]* will be captured at the current and the next rising edges of *clk_2x* and sent to the memory through *d2m[35:0]* (data to memory) at the next rising edge of *clk_2x*.

- **Property 2** (Write Port): by asserting *ws* (W#) low at the rising edge of *clk_2x* when *pflag* is high, the active-low memory enable signal *me* will be set to low at the next rising edge of *clk_2x*.

- **Property 3** (Read Port): by asserting *rs* (R#) low at the rising edge of *clk_2x* when *pflag* is high, the data from the memory *d_m[35:0]* will be sent out through *dout[15:0]* (DO[15:0]) after the next rising edge of *clk_2x*.

3.4.3 Verification

We also use STE to verify the modified LA-1 RTL design in Forte.

The STE formulations of the three properties described above are given below:

- **STE Assertion 1** for **Property 1**:

$$
\begin{aligned}
&((clk\_2x = 0) \wedge (pflag = 1) \wedge (ws = 0) \wedge (bwe[1:0] = 0) \wedge (din[15:0] = d1[15:0])) \\
&\wedge \mathrm{N}(clk\_2x = 1) \wedge \mathrm{N}^2((clk\_2x = 0) \wedge (pflag = 0) \wedge (din[15:0] = d2[15:0])) \\
&\wedge \mathrm{N}^3(clk\_2x = 1) \wedge \mathrm{N}^4((clk\_2x = 0) \wedge (pflag = 1)) \\
&\rightarrow \mathrm{N}^4((d2m[31:16] = d1[15:0]) \wedge (d2m[15:0] = d2[15:0])).
\end{aligned}
$$

- **STE Assertion 2** for **Property 2**:

$$((clk\_2x = 0) \wedge (pflag = 1) \wedge (ws = 0)$$
$$\wedge \mathrm{N}(clk\_2x = 1) \wedge \mathrm{N}^2((clk\_2x = 0) \wedge (pflag = 0))$$
$$\wedge \mathrm{N}^3(clk\_2x = 1) \wedge \mathrm{N}^4((clk\_2x = 0) \wedge (pflag = 1))$$
$$\rightarrow \mathrm{N}^4(me = 1).$$

- **STE Assertion 3** for **Property 3**:

$$((clk\_2x = 0) \wedge (pflag = 1) \wedge (rs = 0))$$
$$\wedge \mathrm{N}(clk\_2x = 1) \wedge \mathrm{N}^2((clk\_2x = 0) \wedge (pflag = 0))$$
$$\wedge \mathrm{N}^3(clk\_2x = 1) \wedge \mathrm{N}^4((clk\_2x = 0) \wedge (pflag = 1) \wedge (d\_m[35:0] = d1[35:0]))$$
$$\wedge \mathrm{N}^5(clk\_2x = 1) \wedge \mathrm{N}^6((clk\_2x = 0) \wedge (pflag = 0))$$
$$\wedge \mathrm{N}^7(clk\_2x = 1) \wedge \mathrm{N}^8((clk\_2x = 0) \wedge (pflag = 1))$$
$$\rightarrow \mathrm{N}^6(dout[15:0] = d1[15:0]) \wedge \mathrm{N}^8(dout[15:0] = d1[31:16]).$$

3.4.4 Experimental Results

All the three STE assertions of the modified design were verified using Forte. Besides those properties targeted for the 28 bits address width, we also verified the same three STE assertions for the 4 bits address width in Forte. Firstly, in both cases, the memory usages for each of the three STE assertions were almost the same. Secondly, it turned out that the memory usage did not grow dramatically with the growth of the address bus width but almost remained the same. In fact, all the memory usages for the three plus three runs were less than 1M. Thirdly, the time complexity is also not an issue since all the runs for the six different assertions took less than one second.

Compared with the experimental results of Ahmed's design, in which the memory usage were much larger and grew almost linearly with the width of the address bus, these results were not surprising since we removed the Memory block from the modified design which took most of the resources.

3.5 Summary

In this chapter, we performed a case study of verifying the Look-Aside Interface using STE, through which we obtain in-depth understanding and practical experience of this symbolic model checking approach. In the next two chapters, we will investigate the MDG-based model checking approach in the same way: describing the underlying theory and methodology in one chapter and presenting a case study of it in the next chapter.

Chapter 4

MDG-based Model Checking

To deal with the state explosion problem of traditional BDD-based symbolic model checking methods, a new MDG-based model checking approach is proposed by Corella et al. [10]. In this chapter, we first introduce the theoretical foundations of MDG-based model checking in Section 4.1 and 4.2. After that, the modeling, specification language and verification methodology of this approach will be described in detail in the following sections. Finally, after providing an illustrative example of this approach, MDG-based model checking tools are discussed.

4.1 Many-sorted First-order Logic

Whereas Boolean propositional logic is used in BDD-based model checking approaches to model circuits at the bit level, a more expressive logic is needed in the MDG-based method in order to represent the circuits at higher level of abstraction. A modified many-sorted first-order logic is then proposed for this purpose.

Standard many-sorted first-order logic [11] is a very powerful language in terms of expressiveness and it can be viewed as a unifying framework for all other logics, including higher-order logic. By adding the notion of type (or sort) to the formalism of first-order logic, it gains modeling flexibility and retains the tractability of first-order

logic, such as completeness, compactness, structural induction over terms and formulas, and efficient matching and unification algorithms.

The logic used in the MDG-based verification modified the standard many-sorted first-order logic by separating the set S of sorts into two classes: the set S_c of concrete (also called enumerated) sorts and the set S_a of abstract sorts, which makes possible the distinction between data path and control path in hardware verification.

Concrete sorts have enumerations, while abstract ones do not. The enumeration of a concrete sort $s \in S_c$ is a list of constants of sort s, called individual constants. The constants that do not show in any enumeration are generic constants of abstract sorts. Constants or variables may be of concrete sort or abstract sort. As a special case, the Boolean logic may be included in this logic as a concrete sort with an enumeration over $\{0, 1\}$.

Function symbols are classified into three categories according to the sorts of its arguments and the result:

- concrete function symbol (with a concrete result and concrete arguments),
- abstract function symbol (with an abstract result),
- or cross-operator (with a concrete result and at least one abstract argument).

Both abstract function symbols and cross-operators are uninterpreted and they are used to model data operations and feedback from data path to control, respectively. Concrete function symbols are used to denote control path operations.

Terms and formulas are defined inductively in the similar way as in standard many-sorted first-order logic. In short, terms are formed of sorts, constants, variables, and

function symbols, and formulas are defined using equations of terms, logical connectives and quantifiers.

A term is said to be concrete/abstract if it is of concrete/abstract sort. A term is concretely reduced if and only if it has no concrete sub-terms other than individual constants, i.e., a concretely-reduced term is formed of either abstract terms or individual constants. A cross-operator $f(t_1,t_2,\ldots,t_n)$ is called a cross-term if all the arguments $t_1,t_2,\ldots,t_n$ are concretely-reduced terms.

An interpretation, δ, is a mapping that assigns a denotation (a non-empty set) to each abstract sort. A concrete sort or a constant is itself a denotation. Let V be a set of variables. A δ-compatible assignment with domain V, ϕ_V^δ, is a function that maps each variable in V of sort s to an element of the denotation of the sort s. Let Φ_V^δ be the set of all possible δ-compatible assignments to the variables in V.

The truth semantics of a formula is defined relative to an interpretation and an assignment compatible to it. More precisely, given an interpretation δ and a δ-compatible assignment ϕ to the variables that occur free in F, the truth of a formula F, denoted by $\delta,\phi \models F$, is defined recursively as follows:

- $\delta,\phi \models t_1 = t_2$, iff t_1 and t_2 are terms of the same denotation.
- $\delta,\phi \models \neg F$ iff not $\delta,\phi \models F$.
- $\delta,\phi \models F_1 \wedge F_2$ iff $\delta,\phi \models F_1$ and $\delta,\phi \models F_2$.
- $\delta,\phi \models F_1 \vee F_2$ iff $\delta,\phi \models F_1$ or $\delta,\phi \models F_2$.
- $\delta,\phi \models (\exists x)F$ iff there exists an assignment ϕ' which is ϕ extended with an assignment to variable x such that $\delta,\phi' \models F$

- $\delta,\phi \models (\forall x)F$ iff for any assignment ϕ' which is ϕ extended with an assignment to variable x, $\delta,\phi' \models F$ holds.

We use $\models F$ to denote the case where $\delta,\phi \models F$ holds for all δ and $\phi \in \Phi_V^\delta$ with variables in V occur free in F.

4.2 Multiway Decision Graphs

Multiway Decision Graph (MDG) is a data structure representing a formula in the many-sorted first-order logic described in the previous section.

Definition 4.2.1: Let X and A be two sets of variables such that X ∩ A = ∅. An MDG of type X → A is a finite rooted directed acyclic graph (DAG) G, where

- Each non-root leaf node is labeled by formula ⊤ (truth), and a root leaf node (in which case G has only one node) may be labeled by formula ⊤ or ⊥ (falsity).
- For each internal node n, either
  - n is labeled by a cross-term of concrete sort α with variables in X, and the outgoing edged of n are labeled by individual constants of α, or
  - n is labeled by a variable in X of concrete sort α, and the outgoing edges of n are labeled by individual constants of α, or
  - n is labeled by a variable in A of concrete sort α, and the outgoing edges of n are labeled by individual constants of α, or
  - n is labeled by a variable in A of abstract sort β, and the outgoing edges of n are labeled by concretely-reduced terms of β with variables in X.

A well-formed (reduced and ordered) MDG [10] is a canonical graph representation of a quantifier-free and negation-free many-sorted first-order formula, called Directed Formula (DF) [7].

Note that a BDD is a special case of an MDG. More precisely, a BDD can be transformed into an MDG by

- replacing the label 0 or 1 of a leaf node with T or ⊥, and
- removing all the non-root leaf nodes labeled ⊥ and all the related incoming edges.

4.3 Modeling

In MDG-based model checking, digital systems under verification are modeled by abstract descriptions of state machines (ASMs), where both sets of states and relations are encoded by MDGs.

An abstract description of a finite state machine M is a structure $A = (V_I, V_S, V_O, \nu, S_I, R_T, R_O)$, where

- V_I, V_S and V_O are pairwise disjoint vectors of input, state and output variables respectively.
- ν is the function that maps each state variable in V_S to the corresponding next state variable. Thus $V_S^{'} = \nu(V_s)$ is the next state variable set, which is disjoint from V_I, V_S and V_O.
- S_I is the abstract description of the set of initial states encoded by an MDG of type $X \to V_S$, where X is a set of abstract variables disjoint from V_I, V_S, V_O and $V_S^{'}$.

- R_T is the abstract description of the transition relation encoded by an MDG of type $V_I \cup V_S \rightarrow V_S'$.
- R_O is the abstract description of the output relation encoded by an MDG of type $V_I \cup V_S \rightarrow V_O$.

Abstract descriptions of state machines describe state machines at a higher level of abstraction. For each interpretation δ, one and only one state machine M can be obtained by applying δ to the abstract description A, which is of the form $M = (\Phi_{V_I}^{\delta}, \Phi_{V_S}^{\delta}, \Phi_{V_O}^{\delta}, Set_{V_S}^{\delta}(S_I), R_T{}^{\delta}, R_O{}^{\delta})$ such that

- $\Phi_{V_I}^{\delta}$, $\Phi_{V_S}^{\delta}$ and $\Phi_{V_O}^{\delta}$ are the sets of all possible δ-compatible assignments to the variables in V_I, V_S and V_O respectively, i.e., the set of input vectors, the set of states and the set of output vectors respectively.
- $Set_{V_S}^{\delta}(S_I) = \{\phi \in \Phi_{V_S}^{\delta} \mid \delta, \phi \models (\exists X) S_I\}$ is the set of initial states.
- $R_T{}^{\delta} = \{(\phi, \phi', \phi'') \in \Phi_{V_I}^{\delta} \times \Phi_{V_S}^{\delta} \times \Phi_{V_S}^{\delta} \mid \delta, \phi \cup \phi' \cup (\phi'' \circ \nu) \models R_T\}$ is the transition relation.
- $R_O{}^{\delta} = \{(\phi, \phi', \phi'') \in \Phi_{V_I}^{\delta} \times \Phi_{V_S}^{\delta} \times \Phi_{V_O}^{\delta} \mid \delta, \phi \cup \phi' \cup \phi'' \models R_O\}$ is the output relation.

4.4 Specification Language

A specification language called L_{MDG} is used to express the properties to be verified in the MDG-based model checking approach.

L_{MDG} [29] is a CTL-like specification language based on many-sorted first-order logic, which is used to describe properties for abstract description of state machines.

ASMs lift the system modeling in BDD-based approaches from the propositional level to the first-order level. Similarly, L_{MDG} lifts CTL (a specification language used in BDD-based approaches) from the propositional level to the first-order level.

A Next_let_formula is the basic building block of a L_{MDG} property. Given an ASM and a set of ordinary variables (not occurring in the ASM), the recursive definition of a Next_let_formula is as follows:

- An equation $u_1 = u_2$ is a Next_let_formula, if u_1 is an ASM variable and u_2 is an ASM variable, an ordinary variable, or a constant.
- $!f$ (not f), $f \& g$ (f and g), $f \mid g$ (f or g) and $f \rightarrow g$ (f implies g) are Next_let_formulas, if both f and g are Next_let_formulas.
- **LET** $(v = u)$ **IN** f is a Next_let_formula, if v is an ordinary variable, u is an ASM variable, and f is a Next_let_formula. We call this type of formulas **LET–IN** formulas.
- X f is a Next_let_formula, if f is a Next_let_formula and X is the next-time operator.

Just like in the symbolic trajectory formulas, finite depth of nesting of the next-time operator is also allowed in the Next_let_formulas.

Let p and q be Next_let_formulas. A L_{MDG} property is defined of either of the following forms:

- A(p),
- AG(p),
- AF(p),

- $A(p)\,U\,(q)$,
- $AG((p) \Rightarrow (F(q)))$,
- $AG((p) \Rightarrow ((p)\,U\,(q)))$.

The truth semantics of a L_{MDG} property is defined relative to an interpretation δ and a δ-compatible assignment ϕ. A detailed description of the semantics can be found in [28].

4.5 Verification Methodology

4.5.1 Reachability Analysis in MDG-based Model Checking

As mentioned previously, given an abstract description $A = (V_I, V_S, V_O, \nu, S_I, R_T, R_O)$ of finite state machines, for any interpretation δ, one and only one state machine $M = (\Phi_{V_I}^{\delta}, \Phi_{V_S}^{\delta}, \Phi_{V_O}^{\delta}, Set_{V_S}^{\partial}(S_I), R_T{}^{\delta}, R_O{}^{\delta})$ can be obtained by applying δ to the abstract description A. We now show how to perform the reachability analysis of M using some basic MDG algorithms.

The pseudo-code [7] shown in Figure 26 describes the algorithm **ReAn** for reachability analysis, where R, Q, I, P and N are MDG variables representing sets of states, O is an MDG variable representing the set of output vectors, K is the loop counter, **Fresh** is a local function, and **ReIP**, **PbyS** and **Disj** are basic MDG algorithms which are described in detail in [7]. Note that, an invariant condition C represented by an MDG is checked against A during the reachability analysis.

In line 2, before the start of the loop, R that represents the set of reachable states found so far and Q that represents the frontier set, that is, a subset of R containing at least

all those states entering R for the first time in the previous loop iteration, are initialized to the MDG representing the set of initial states, and the loop counter K is reset to zero.

Lines 3 to 14 specify the body of the loop.

In line 5, function **Fresh**(V_I, K) constructs an MDG representing the set of input vectors which are fresh variables related to the value of K.

```
1.    ReAn(A, C)
2.        R := S_I; Q := S_I; K := 0;
3.        loop
4.            K := K + 1;
5.            I := Fresh(V_I, K);
6.            O := ReIP({I, Q, R_O}, V_I ∪ V_S, ∅);
7.            P := PbyS(O, C);
8.            if P ≠ F then return failure;
9.            N := ReIP({I, Q, R_T}, V_I ∪ V_S, η);
10.           Q := PbyS(N, R);
11.           if Q = F then return success;
12.           R := PbyS(R, Q);
13.           R := Disj(R, Q);
14.       end loop;
15.   end ReAn;
```

Figure 26. Reachability analysis algorithm in MDG-based model checking

Lines 6 to 8 are used to check if the invariant C holds and to terminate the algorithm and report failure if the check fails. In line 6, the relational product algorithm **ReIP**

computes an MDG representing the set of output vectors produced by the states in the frontier set Q, which is assigned to O. In line 7, the pruning-by-subsumption algorithm **PbyS** is used to remove from O any path leading to output vectors that satisfy the invariant C, the resulting MDG of which is assigned to P. In line 8, if the set represented by P is not empty which means not all the output vectors produced by the states in the frontier set satisfy the invariant, the algorithm terminates, reports failure and provides a counterexample.

Lines 9 to 11 compute the new frontier set and check if the fixpoint has been reached. Line 9 computes an MDG representing the set states reachable in one transition from the frontier set, which is assigned to N. Line 10 is used to remove from N the current reachable states represented by R, the resulting MDG of which representing the new frontier set is assigned to Q. In line 11, if the new frontier set is empty which means the fixpoint has been reached, the algorithm terminates and returns success.

Lines 12 and 13 are used to compute an MDG representing the new set of reachable states by unioning the new frontier set Q with R. First, in line 12, R is simplified using **PbyS** by removing any path subsumed by Q. Then, in line 13, the disjunction algorithm **Disj** computes an MDG representing the union of sets represented by R and Q, and assign the resulting MDG to R.

Note that, the reachability algorithm described above may produce false negative and may not terminate, the discussion of which is beyond the scope of this thesis and the detail of which can be found in [7].

4.5.2 Model Checking of L_{MDG} Properties

In general, the MDG-based model checking approach is based on abstract implicit state enumeration (the reachability analysis algorithm described in the previous section). Different property checking algorithms [28] are developed for L_{MDG} formulas of various forms. The basic idea is to use an automatic tool to build additional ASMs for the L_{MDG} property to be verified, connect the additional ASMs to the ASM model M representing the design under verification to make a new composite ASM model M', and then set an invariant condition to be checked against M' during the reachability analysis of M'. If the invariant holds in all the reachable states of M', we then prove that model M satisfies the L_{MDG} property.

In this approach, data signals are denoted by abstract variables instead of Boolean variables, and data operators are represented by uninterpreted or partially interpreted function symbols instead of Boolean functions. Thus, the verification based on abstract implicit state enumeration can be carried out independently of data path width, which therefore can effectively alleviate the state explosion problem.

4.6 MDG-based Verification Tools

The MDG tools [30] are implemented in Prolog as our MDG-based verification tools. The MDG tools, targeted to the verification of RTL designs modeled by ASMs, consist of an MDG package, a reachability analysis algorithm, applications for RTL verification, and a model checker for L_{MDG}. The MDG package contains a set of manipulation algorithms for MDGs, the details of which can be found in [10]. The reachability analysis algorithm explores all the reachable states of an ASM and checks whether an invariant

holds in all those states. Four applications for RTL verification are provided in the MDG tools: ASM state exploration, ASM safety property checking, ASM equivalence checking and Combinational verification. The MDG model checker [27] performs checks on properties expressed in L_{MDG} against an ASM model. Our verification for the modified LA-1 RTL design was performed using the MDG model checker in the MDG tools.

The MDG model checker accepts only design models in MDG-HDL [27], a Prolog-style Hardware Description Language which allows the use of abstract variables and uninterpreted function symbols. Therefore, a converter is needed to translate the Verilog RTL into MDG-HDL format. Note that, in this case study, due to time limitation (not technical limitation), we did not build the converter but did the translation for the modified LA-1 RTL design manually. We will put the efforts of developing such a Verilog to MDG-HDL converter as our future work. Besides the MDG-HDL description of the design, a bunch of other information, such as sort and function type definitions and user-defined symbol ordering, is also needed by the MDG model checker in order to perform the verification properly. All the required information is arranged into four input files: the algebraic specification file, the symbol order file, the circuit description file and the invariant specification file. Detailed descriptions of these files can be found in [30].

Design properties are expressed in L_{MDG}. Given a L_{MDG} property, the property parser in MDG model checker will develop an additional MDG-HDL code for the property, merge the additional code with the original MDG-HDL code generated for the design under verification, and set an invariant condition in the invariant specification file.

The new merged MDG-HDL code and the invariant are then sent into the MDG model checker, where the MDG-HDL code is complied into an ASM encoded internally

by MDGs and the invariant is checked against the ASM model during the reachability analysis of this model. If the invariant holds in all the reachable states of the ASM model, we can then prove that the design under verification satisfies the L_{MDG} property. When the checking for the invariant fails at some stage of the reachability analysis procedure, the procedure will be terminated immediately and a counterexample will be generated to indicate the states not satisfying the invariant.

4.7 Summary

In this chapter, we presented the theory and methodology of MDG-based model checking and introduce the verification tool for it. In the next Chapter, we will further investigate this approach by another case study of verifying the same Look-Aside Interface using MDGs.

Chapter 5

Verifying Look-Aside Interface using MDGs

In Chapter 3, we verified several properties of the Look-Aside Interface using STE. In this chapter, we will make another case study of verifying the same properties of the interface using MDGs. By comparing the syntax of STE assertions with that of L_{MDG} properties, we can see that L_{MDG} properties are more powerful than STE assertions in terms of expressiveness. More precisely, the properties that STE assertions can describe are a subset of those that can be expressed by L_{MDG} properties. Thus, we can easily get the L_{MDG} properties to be verified against the interface model by mapping from of the STE assertions. We will provide a method to perform the mapping from STE assertions to L_{MDG} properties.

5.1 Mapping STE Assertions to L_{MDG} Properties

We start from the normalized form of basic STE assertions

$$A_0 \wedge \mathrm{N}(A_1) \wedge \ldots \wedge \mathrm{N}^i(A_i) \rightarrow C_0 \wedge \mathrm{N}(C_1) \wedge \ldots \wedge \mathrm{N}^i(C_i),$$

where A_i and C_i are simple predicates or conjunctions of these or empty, $\wedge$ and $\rightarrow$ are logic connectives "and" and "implication" respectively, N is the next-time operator, and N_i denotes i next time operators. For example, A_i can be $D = d$, a simple predicate which states that node D of a circuit has the value of d at the present time and d can be a

symbolic variable or a constant or a vector of either of them. The antecedent instructs the initialization of signals for the symbolic simulation in STE and the consequent defines the expected response of the circuit which is then checked against the simulation result.

Note that the time unit in STE is half clock cycle, while MDG tools use one clock cycle as the time unit. Thus, only those STE assertions that have no predicates at both edges of the clock can be mapped to L_{MDG} properties.

Before doing the real work, we need to check whether an STE assertion is suitable for the mapping or not. If it is not, we need to redo the RTL design to remove the use of both edges of the clock. One solution is to generate an internal double-frequency clock *clk_2x* from the original clock *clk* and use *clk_2x* as the only clock for the circuit. The rising edge and the falling edge of the original clock *clk* can be obtained by combining *clk_2x* and clk, in which case *clk* is viewed as a control input.

We then remove the predicates related to the clock signal in the STE assertion because in the MDG tools the clock signal is implemented implicitly. The resulting STE assertion should be of the following form:

$$A_0 \wedge \mathrm{N}^2(A_2) \wedge \ldots \wedge \mathrm{N}^j(A_j) \rightarrow C_0 \wedge \mathrm{N}^2(C_2) \wedge \ldots \wedge \mathrm{N}^j(C_j),$$

where j is the greatest even number that is equal or less than k.

In order to ease the mapping process, we first transform the normalized form of an STE assertion mentioned earlier into an equivalent formula by decomposing the implication into a conjunction of sub-implications according to the consequent side as:

$$\begin{aligned}
&A_0 \wedge \mathrm{N}^2(A_2) \wedge \ldots \wedge \mathrm{N}^j(A_j) \rightarrow C_0 \wedge \mathrm{N}^2(C_2) \wedge \ldots \wedge \mathrm{N}^j(C_j) = \\
&(A_0 \wedge \mathrm{N}^2(A_2) \wedge \ldots \wedge \mathrm{N}^j(A_j) \rightarrow C_0) \wedge \\
&(A_0 \wedge \mathrm{N}^2(A_2) \wedge \ldots \wedge \mathrm{N}^j(A_j) \rightarrow \mathrm{N}^2(C_2)) \wedge \\
&\ldots \wedge \\
&(A_0 \wedge \mathrm{N}^2(A_2) \wedge \ldots \wedge \mathrm{N}^j(A_j) \rightarrow \mathrm{N}^j(C_j))
\end{aligned}$$

Next, we do the decomposition again for each of the sub-implications shown above, according to the antecedent side, as below:

$$A_0 \wedge \mathrm{N}^2(A_2) \wedge \ldots \wedge \mathrm{N}^j(A_j) \rightarrow \mathrm{N}^i(C_i) =$$
$$(A_0 \rightarrow \mathrm{N}^i(C_i)) \wedge (\mathrm{N}^2(A_2) \rightarrow \mathrm{N}^i(C_i)) \wedge \ldots \wedge (\mathrm{N}^j(A_j) \rightarrow \mathrm{N}^i(C_i)) =$$
$$(A_0 \rightarrow \mathrm{N}^i(C_i)) \wedge (\mathrm{N}^2(A_2 \rightarrow \mathrm{N}^{i-2}(C_i))) \wedge \ldots \wedge (\mathrm{N}^j(A_j \rightarrow \mathrm{N}^{i-j}(C_i)))$$

Then we start the real mapping process. The first step is to replace a state variable or a vector of state variables with an ASM variable. Then we use the **LET-IN** formulas mentioned above to rewrite the final sub-implications of form $A_l \rightarrow \mathrm{N}^m(C_n)$ as follows:

- Any predicate in A_l of form $v_{ai} = v_{si}$ or $v_{ai} = v_{si}[n:0]$ will be re-written as $v_{oi} = v_{ai}$, where v_{ai}, v_{si}, $v_{si}[n:0]$ and v_{oi} are an ASM variable, a symbolic variable, a vector of symbolic variable and an ordinary variable respectively.
- Any predicate in C_n that uses v_{si} or $v_{si}[n:0]$ will use v_{oi} instead.

The resulting sub-implication will be written as:

LET $((v_{o1}=v_{a1})\&\ \ldots\ \&\ (v_{op}=v_{ap}))$
IN $(((v_{ac1}=c_1)\&\ \ldots\ \&\ (v_{acq}=c_q))\ \ (\mathrm{N}^m(C_n')))$

where v_{aci} is an ASM variable in A_l, c_i is a Boolean constant or a vecotor of Boolean constants, and C_n' is the resulting consequent by replacing v_{si} or $v_{si}[n:0]$ in C_n with v_{oi}.

LET-IN formulas allow us to use ordinary variables to remember the current values of ASM variables in the antecedent which are then used in the consequent. Note that the predicates in the antecedent that have the state variables or vectors of state variables assigned constant values should keep the same structure and not be transformed by **LET-IN** formulas, since constant values do not change with time and therefore there is no need to use **LET-IN** formulas to store the current values for them.

Thirdly, the logic connectives $\rightarrow$ and $\wedge$ should be mapped to their counterparts in L_{MDG} which are $\rightarrow$ and & respectively and each two next time operators N^2 should be replace by one X which is the next time operator in L_{MDG}.

Next, we may do some compositions to get a more compact formula.

The last step is simply to add AG to the front of the resulting formula.

The mapping process can be illustrated by the following example. Assume that we want to map the following STE assertion to a L_{MDG} property:

$$\begin{array}{l}((clk=0)\wedge(en=1)\wedge(din[15:0]=d1[15:0]))\wedge \mathrm{N}(clk=1)\\ \wedge \mathrm{N}^2((clk=0)\wedge(en=0)\wedge(din[15:0]=d2[15:0]))\\ \wedge \mathrm{N}^3(clk=1)\wedge \mathrm{N}^4(clk=0)\\ \rightarrow \mathrm{N}^2(dout=d1)\wedge \mathrm{N}^4(dout=d2)\end{array},$$

where *clk*, *en*, *din*[15:0], and *dout*[15:0] are state variables or vectors of state variables representing the clock signal, the control input, the 16-bit data input and the 16-bit data output respectively.

First, by removing the clock related predicates, we get:

$$\begin{array}{l}((en=1)\wedge(din[15:0]=d1[15:0]))\\ \wedge \mathrm{N}^2((en=0)\wedge(din[15:0]=d2[15:0]))\\ \rightarrow \mathrm{N}^2(dout=d1)\wedge \mathrm{N}^4(dout=d2)\end{array}.$$

Next, we decompose the above formula as:

$$\begin{array}{l}(((en=1)\wedge(din[15:0]=d1[15:0]))\rightarrow \mathrm{N}^2(dout=d1))\\ \wedge \mathrm{N}^2(((en=0)\wedge(din[15:0]=d2[15:0]))\rightarrow(dout=d1))\\ \wedge(((en=1)\wedge(din[15:0]=d1[15:0]))\rightarrow \mathrm{N}^4(dout=d2))\\ \wedge \mathrm{N}^2(((en=0)\wedge(din[15:0]=d2[15:0]))\rightarrow \mathrm{N}^2(dout=d2))\end{array}$$

Finally, by applying the **LET-IN** formulas, we have the mapped L_{MDG} property as:

```
AG(   ( LET (v1=din) IN ( (en=1) → X(dout=v1) ) )
      & X( LET (v2=din) IN ( (en=0) → (dout=v1) ) )
      & ( LET (v1=din) IN ( (en=1) → XX(dout=v2) ) )
      & X( LET (v2=din) IN ( (en=0) → X(dout=v2) ) ));
```

where $v1$ and $v2$ are ordinary variables and the other three variables *din*, *en* and *dout* are ASM variables . Note that $v1$, $v2$, *din* and *dout* should be of the same abstract sort and *en* is of a concrete sort. Thus, vectors of Boolean variables (state variables) in STE assertions are mapped to abstract variables in L_{MDG} properties, which makes the verification in MDG tools independent of data path width.

5.2 Verifying Modified LA-1 RTL Design using MDGs

5.2.1 Modeling

The MDG-HDL model for the Write Port of the modified LA-1 RTL design is shown in Figure 27, where

- input signals *clk_2x*, *pflag*, *ws*, *dpin1*, *dpin0*, *bwe1* and *bwe0* are of type bool,
- input signals *din* and *addrin* are of abstract sort 'wordn',
- output signals *me*, *bwe_m3*, *bwe_m2*, *bwe_m1*, and *bwe_m0* are of type bool,
- output signals *d2m* and *addr_w* are of abstract sort 'wordn', and
- components *make_word*, *parity1*, *parity2*, *parity3* and *parity4* are abstract function symbols.

Note that signals *dpin1* and *dpin0* are mapped from *dpin*[1] and *dpin*[0] in the Verilog design respectively. Similarly, signals *bwe1* and *bwe0* are mapped from *bwe*[1] and *bwe*[0] respectively, and signals *bwe_m3*, *bwe_m2*, *bwe_m1*, and *bwe_m0* are mapped from *bwe2m*[3], *bwe2m*[2], *bwe2m*[1] and *bwe2m*[0] respectively. The function of *make_word* is to merge two input data into one output data. The function of *parity1*, *parity2*, *parity3* or *parity4* is to compute the parity of the input data.

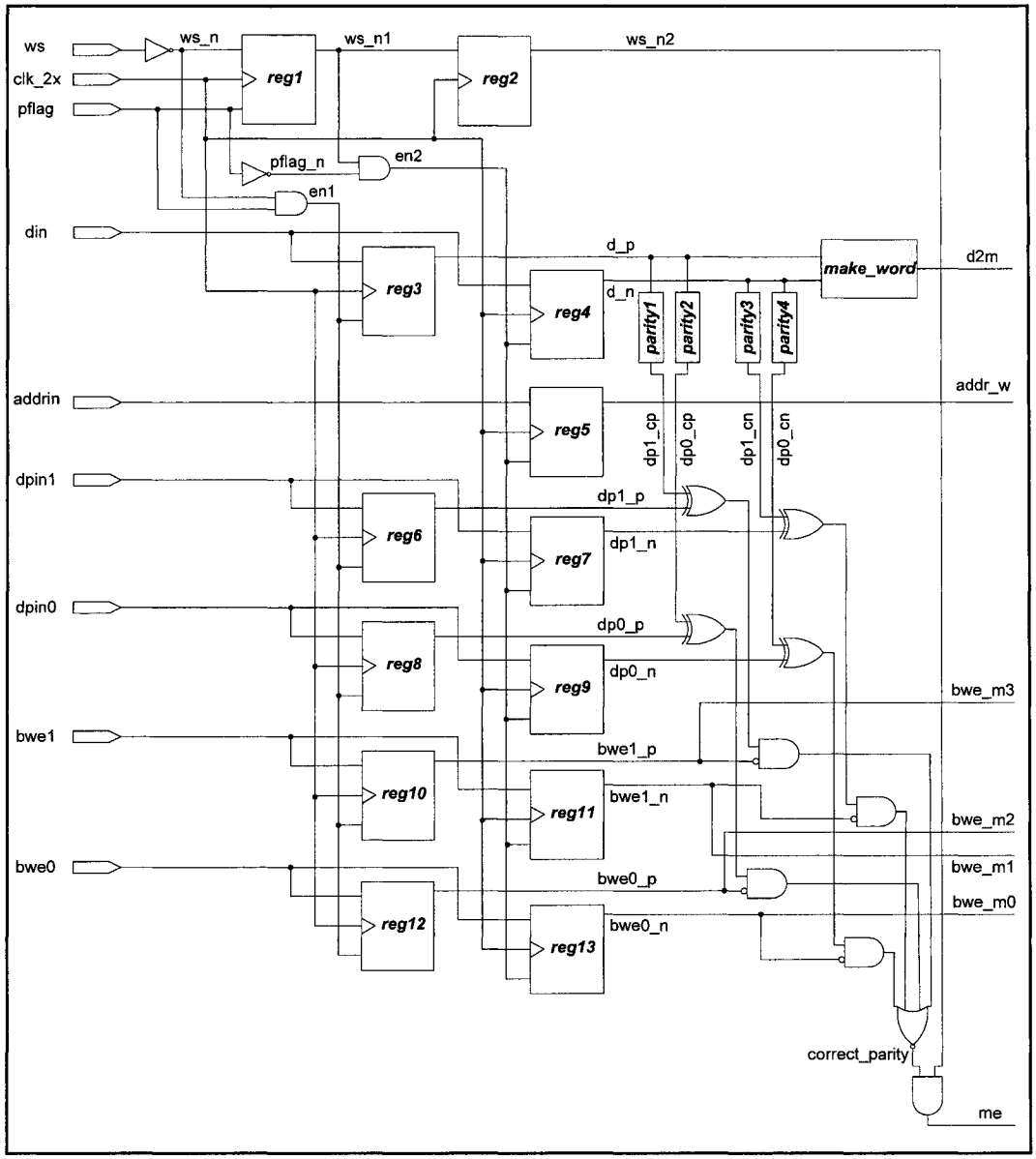


Figure 27. MDG-HDL model for the Write Port of the modified LA-1 RTL design

The MDG-HDL model for the Read Port of the modified LA-1 RTL design is shown in Figure 28, where

- input signals *clk_2x*, *pflag* and *rs* are of type bool,
- input signals *d_m* and *addrin* are of abstract sort 'wordn',

- output signals *dpout1* and *dpout0* are of type bool,
- output signals *dout* and *addr_r* are of abstract sort 'wordn', and
- components *msw*, *lsw*, *parity1*, *parity2*, *parity3* and *parity4* are abstract functions.

Note that signals *dpout1* and *dpout0* are mapped from *dpout*[31] and *dpout*[0] in the Verilog design respectively. The function of *msw* is to strip the most significant word from the input data. The function of *lsw* is to strip the least significant word from the input data. The function of *parity1*, *parity2*, *parity3* and *parity4* is to compute the parity of the input data.

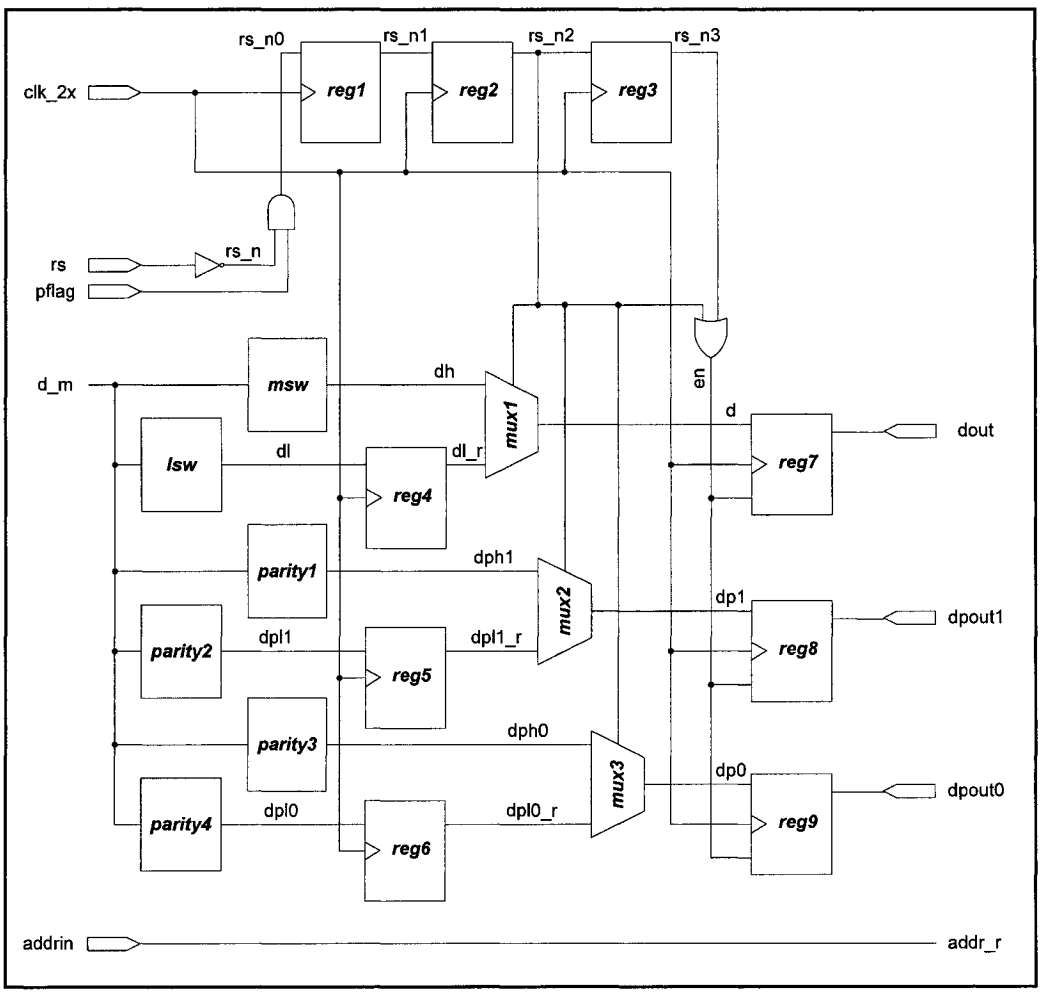


Figure 28. MDG-HDL model for the Read Port of the modified LA-1 RTL design

5.2.2 Properties

The L_{MDG} properties listed below are obtained by mapping from the STE assertions presented in the previous section using the mapping procedure proposed earlier in this chapter:

- **L_{MDG} Property 1** mapped from **STE Assertion 1**:

```
AG(
        (pflag=1 & ws=0 & bwe1=0 & bwe0=0)->
                ( LET (v1=din) IN
                        ( X( LET (v2=din) IN
                                ( X( d2m = fmake_word(v1,v2) ) )
                          )
                        )
                )
);
```

- **L_{MDG} Property 2** mapped from **STE Assertion 2**:

```
AG(
        (pflag=1 & ws=0) ->
                ( XX(me = 1)
                )
);
```

- **L_{MDG} Property 3** mapped from **STE Assertion 3**:

```
AG(
        ((pflag=1)&(rs=0))->
                ( XX ( LET (v1=d_m) IN
                        ( X (   ( dout=fmsw(v1))
                              & X (dout=flsw(v1))
                          )
                        )
                    )
                )
);
```

5.2.3 Experimental Results

During our verification using the MDG tools, we found some bugs in the MDG tools which are listed below:

- The property parser does not support the nested **LET-IN** structure in the L_{MDG} properties and only the last Let_equation will be counted.
- The property parser cannot deal with the cross-terms in the L_{MDG} properties properly. More specifically, when generating the additional MDG-HDL code for a cross-term in the property, the property parser will declare the output signal of the cross-term as an abstract variable which contradicts the fact that the result of a cross-term should be of a concrete sort.
- The property parser cannot build the symbol ordering for the resulting merged MDL-HDL model correctly.

We fixed the problems caused by the above bugs for our modified LA-1 RTL design verification manually.

All the three L_{MDG} properties listed in the previous section have been verified in the MDG tools. Note that MDG-based verification for the LA-1 Interface is independent of address width since the address input is of abstract sort.

Table IV shows the memory usage, runtime and the number of MDG nodes used for verifying each of the three LMDG properties. As can be seen, the memory usages were very small and the time complexity is not an issue since all the runs for the three properties took less than two seconds.

TABLE IV. VERIFICATION STATISTICS FOR THE L_{MDG} PROPERTIES

	Memory (MB)	Runtime (s)	Nodes
L_{MDG} Property 1	1.7	0.8	2300
L_{MDG} Property 2	1.0	0.5	1019
L_{MDG} Property 3	3.0	1.7	5210

5.3 Summary

After exploring the theories and methodologies of STE and MDGs through case studies, it can be concluded that MDG algorithm is a complementary technique to STE, which can lift STE to a higher level of abstraction and can therefore further alleviate the state explosion problem of STE. In the following Chapter, we will discuss the possibility of combining STE and MDGs.

Chapter 6

First-Order Symbolic Trajectory Evaluation using MDGs

In this chapter, we investigate the possibility of using MDGs to perform Symbolic Trajectory Evaluation. Two attempts to combine the Symbolic Trajectory Evaluation with the MDGs are discussed: one in the STE verification environment and the other in the MDG tools. We focus on the second attempt and propose a theory and methodology of performing first-order Symbolic Trajectory Evaluation in the MDG tools. This study may provide direction for further research in the application of MDGs.

6.1 Purpose

Symbolic Trajectory Evaluation technique and MDG-based model checking technique improve the traditional BDD-based symbolic model checking approaches in two different ways. The first one can dramatically reduce the computations for the next state space and enhance the computational efficiency, while the latter one can simplify the data path operations and thus can effectively overcome the state explosion problem. If we can combine these two techniques, it is possible for us to take the advantages of both of them. The basic idea of such a combination is to replace the use of the BDDs with the MDGs

for the encoding of the symbolic expressions and to implement the STE algorithm at a higher level of abstraction which can further alleviate the state explosion problem in STE. We can implement this combination either in the STE environment 'Forte' or in the MDG environment 'MDG tools'.

6.2 Implementing the Combination in Forte

Our goal of the first combination approach is to encode the symbolic expressions in Forte using the MDG package. To achieve this goal, we need to integrate the MDG package into Forte. A BDD package programmed in FL is used to encode symbolic expressions in Forte. As mentioned in previous chapters, the programming language in Forte is FL, while the MDG package that we have in the MDG tools is implemented in Prolog. Hence, the MDG package in Prolog cannot be integrated into Forte system directly to replace the BDD package. In order to solve this problem, we can either program the MDG package again in FL or implement the MDG package in a language other than FL assuming that Forte has a foreign language interface to that language. The procedure of the first approach is somewhat straightforward provided that we have sufficient background knowledge for the MDG package and FL. We now illustrate the basic idea of the latter one using a well known Muddy example.

MuDDy [35] is a SML (Standard ML) interface to BuDDy, a Binary Decision Diagram package written in C language by Jørn Lind-Nielsen [38]. The first usage of MuDDy was in the Hol98 theorem prover to integrate the BuDDy BDD package. BuDDy and a piece of C code 'muddy.c' associating the C functions called in MuDDy with the C functions in Buddy form a new C library, which is then compiled into a dynamically

loadable library. MuDDy is used in the Moscow ML system where Moscow ML's foreign function interface is used to call the C functions in the dynamic library. Moscow ML is a proper extension of SML and every valid SML program should be a valid Moscow ML program [26].

MuDDy makes BuDDy applicable in SML modules via three structures:

- bdd - defining an ML type bdd representing nodes in BuDDy's BDD space, and operations for creating and manipulating ML values representing BDDs,
- fdd - providing support for blocks of BDD variables used to encode values representing elements of finite domains, and
- bvec - providing support for Boolean vectors.

Figure 29 shows an example of how to access BuDDy through MuDDy in the Moscow ML system. When an SML function 'func1' of structure 'bdd' is called from the top level SML file, C function symbol 'mlbdd_func1' in the dynamic library associated to 'func1' by the SML function 'app1' in MuDDy will be accessed. Finally inside the dynamic library, C functions in BuDDy associated to 'mulbdd_func1' in 'muddy.c' will be accessed by the top level SML file.

From the illustration of the above example, we can see that the following steps should be taken for the purpose of integrating the MDG package in Forte through the foreign function interface method:

- Program the MDG package in a language, such as C, to which Forte has a foreign language interface,
- Devise an FL interface to the MDG package, and

- Write a piece of code like 'muddy.c' in the same language as used for the MDG package to associate the functions called in the FL interface with the functions in the MDG package, and compile the code and the MDG package into a dynamically loadable library.

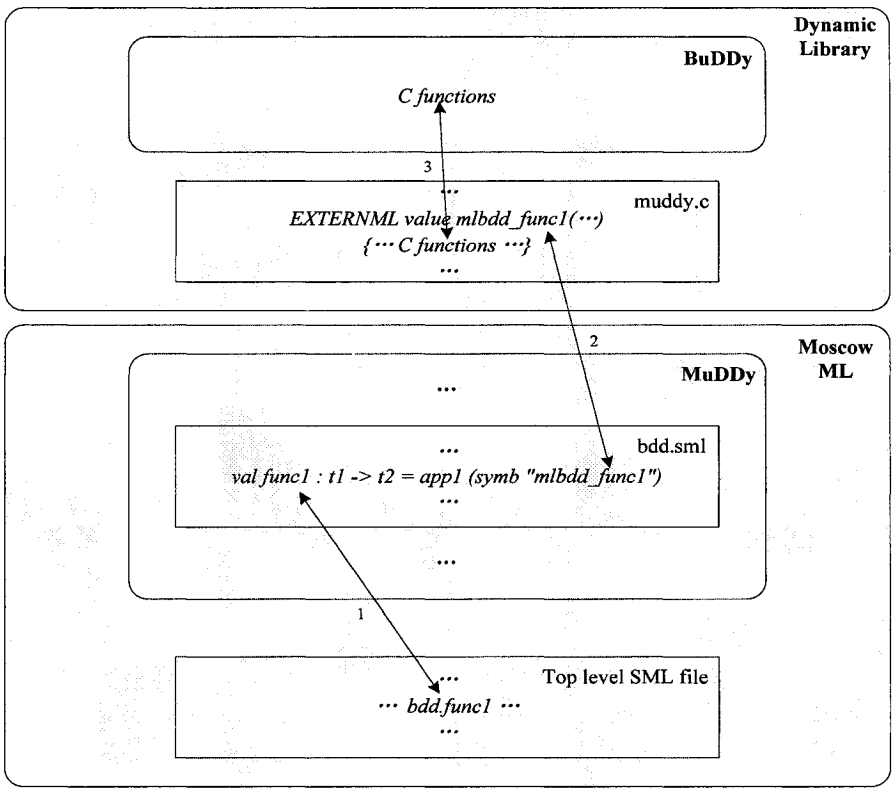


Figure 29. MuDDy in Moscow ML system

Theorically, it is possible to implement both the above methods to integrate the MDG package in Forte either by programming the MDG package directly in FL or by programming the MDG package in a language other than FL and accessing the MDG package using FL's foreign function interface. However, it is practically impossible for us to do so because the FL source code for Forte is not yet open to the public and thus we

can neither modify it nor add more codes to the system. Therefore, we have to make another try at combining these two techniques, detail of which will be discussed in the next section.

6.3 Implementing the Combination in the MDG Tools

In this combination approach, we will develop an MDG-STE engine in the MDG tools where the STE algorithm is implemented using the MDG package.

To do this, we need to first find out the features exclusive to STE and try to implement them in the MDG tools. One such notable feature is the "don't care" value X in the logic of STE. Another important feature of STE is the concept of 'lattice'.

Next, we will discuss respectively the implementations of STE modeling, STE assertions and STE verification methodology using MDGs.

A detailed description of the STE algorithm and related terminology can be found in Chapter 2.

6.3.1 Logic Extension

The underlying logic of STE is three-valued logic which extends the existing 1 (true) and 0 (false) values in two-valued logic with an unknown or "don't care" value X. The X value is essential to the modeling and symbolic simulation in STE, and is absent from the modified many-sorted first-order logic used in MDGs. Hence, for the purpose of implementing the STE algorithm using MDGs, we should also extend the modified many-sorted first-order logic by adding the don't care value X to the denotation of each of the concrete/ abstract sort for each interpretation. For example, if the denotation of a

concrete/abstract sort s is set $\{a_1, a_2, \ldots, a_n\}$ under an interpretation δ, the denotation of s should be extended as $\{a_1, a_2, \ldots, a_n, X\}$.

6.3.2 Implementation of STE Modeling

In STE, a lattice-based tuple $M = [(S, \leq), Suc]$ is used to model the system under verification, where a partial order $\leq$ is defined over the state space $S = \{0, 1, X\}^n \cup \{T\}$ and $(S, \leq)$ forms a complete lattice. Following the same structure, in the MDG-STE engine, we should also define a partial order $\leq_{mdg}$ over the state space S_{mdg} and make $(S_{mdg}, \leq_{mdg})$ a complete lattice.

In MDG tools, an abstract description of the state machine (ASM) is used to model the digital system. An abstract description of an STE model $A_{mdg} = (V_{mdg}, \leq_{mdg}, R_{suc})$ can be built based on the ASM, where

- V_{mdg} is a vector of state variables and each variable could be concrete or abstract,
- $\leq_{mdg}$ is a partial order over the state space, and
- R_{suc} is the abstract description of the next state function encoded by an MDG of type $V_{mdg} \rightarrow V_{mdg}$.

Note that the state variable vector V_{mdg} is actually a combination of variables representing the input signals, the register output signals or the output signals. For a given state variable vector $V_{mdg} = v_0 v_1 \ldots v_{n-2} v_{n-1}$, the next state function $R_{suc}(V_{mdg})$ is actually a vector of next state functions for each element of V_{mdg}, i.e., $R_{suc}(V_{mdg}) = t_0(v_0)t_1(v_1)\ldots t_{n-2}(v_{n-2})t_{n-1}(v_{n-1})$. If element v_i is associated with an input of the circuit,

the next state function $t_i(v_i) = X^c$, and otherwise $t_i(v_i)$ is determined by the circuit structure.

For each interpretation δ, one and only one STE model M_{mdg} can be obtained by applying δ to the abstract description A_{mdg}, which is of the form $M_{mdg} = ((\Phi^{\delta}_{V_{mdg}}, \leq_{mdg}), R^{\delta}_{suc})$ such that

- $\Phi^{\delta}_{V_{mdg}}$ is the set of all possible δ-compatible assignments to the variables in V_{mdg}, i.e., the set of states,
- $\leq_{\text{mdg}}$ is a partial order over $\Phi^{\delta}_{V_{mdg}}$,
- $(\Phi^{\delta}_{V_{mdg}}, \leq_{\text{mdg}})$ is a complete lattice, and
- $R^{\delta}_{suc} = \{(\phi, \phi') \in \Phi^{\delta}_{V_{mdg}} \times \Phi^{\delta}_{V_{mdg}} \mid \delta, \phi \cup \phi' \models R_{suc}\}$ is the next state function, monotone with respect to $\leq_{\text{mdg}}$.

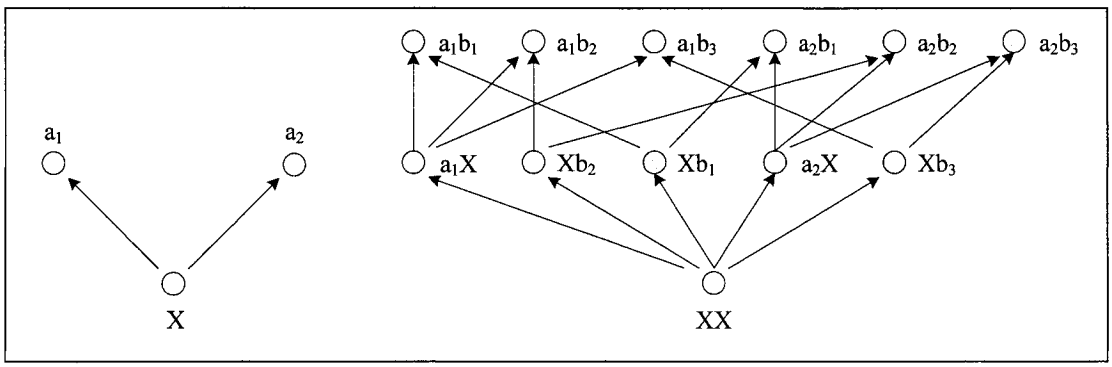


Figure 30. Partial orders over $\{a_1, a_2, X\}$ and $\{a_1, a_2, X\} \cdot \{b_1, b_2, b_3, X\}$

Suppose that the length of the state variable vector V_{mdg} is n and $V_{mdg} = v_1 v_2 \ldots v_n$. The state space $S_{\text{mdg}} = \Phi^{\delta}_{V_{mdg}}$ of the model M_{mdg} can be denoted as $d_1 \cdot d_2 \cdot \ldots \cdot d_n$, where

$d_i (1 \le i \le n)$ is a non-empty set representing the denotation of sort $s_i (1 \le i \le n)$ which is the sort of variable $v_i (1 \le i \le n)$. Note that the don't care value X is an element to each of the denotation, that is, $X \in d_i (1 \le i \le n)$. The partial order $\le_{mdg}$ is defined over $d_1 \cdot d_2 \cdot \ldots \cdot d_n$. Illustrative examples for the partial orders over $\{a_1, a_2, X\}$ and $\{a_1, a_2, X\} \cdot \{b_1, b_2, b_3, X\}$ are shown in Figure 30.

Obviously, $(d_1 \cdot d_2 \cdot \ldots \cdot d_n, \le_{mdg})$ is not a complete lattice since not every subset of $d_1 \cdot d_2 \cdot \ldots \cdot d_n$ has a least upper bound. Therefore, in order to make $(S_{mdg}, \le_{mdg})$ a complete lattice, we introduce the top element T, representing a unique overconstrained state, to the state space S_{mdg}. Thus, the resulting partial order set $d_1 \cdot d_2 \cdot \ldots \cdot d_n \cup \{T\}$, $\le_{mdg}$) forms a complete lattice with T as the universal upper bound and $\perp$ = X, ..., X as the universal lower bound. The complete lattices $(\{a_1, a_2, X\} \cup \{T\}, \le_{mdg})$ and $(\{a_1, a_2, X\} \cdot \{b_1, b_2, b_3, X\} \cup \{T\}, \le_{mdg})$ are shown in Figure 31.

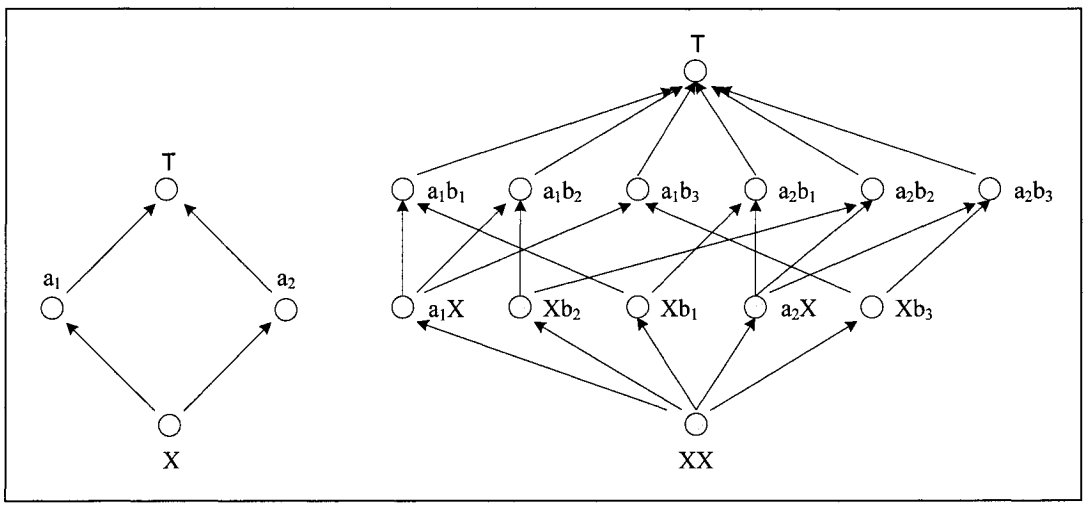


Figure 31. complete lattices $(\{a_1, a_2, X\} \cup \{T\}, \le_{mdg})$ and $(\{a_1, a_2, X\} \cdot \{b_1, b_2, b_3, X\} \cup \{T\}, \le_{mdg})$

6.3.3 Implementation of STE Assertions

In the MDG-STE engine, an STE assertion is of form $[Ante^s_{mdg} \rightarrow Cons^s_{mdg}]$, where both $Ante^s_{mdg}$ and $Cons^s_{mdg}$ are MDG-based symbolic trajectory formulas.

The basic component of a MDG-based symbolic trajectory formula is the simple predicate. Given an STE model $M_{mdg} = ((\Phi^{\delta}_{V_{mdg}}, \leq_{mdg}), R^{\delta}_{suc})$ obtained by applying an interpretation δ to an abstract description $A_{mdg} = (V_{mdg}, \leq_{mdg}, R_{suc})$ of the STE model, a predicate over $\Phi^{\delta}_{V_{mdg}}$ is a function that maps $\Phi^{\delta}_{V_{mdg}}$ to a special complete lattice containing only two elements *false* and *true*, with element *false* as the universal lower bound and element *true* as the universal upper bound. A predicate p_{mdg} over $\Phi^{\delta}_{V_{mdg}}$ is called simple if it is monotone and there exists a unique element $d_{p_{mdg}}$ in $\Phi^{\delta}_{V_{mdg}}$ such that for all $s \in \Phi^{\delta}_{V_{mdg}}$ with $d_{p_{mdg}} \leq_{mdg} s$, $p_{mdg}(s) = true$. The $d_{p_{mdg}}$ here is called the defining value of predicate p_{mdg}. We denote the set of all simple predicates over $\Phi^{\delta}_{V_{mdg}}$ by P_{mdg}. A simple predicate p_{mdg} over $\Phi^{\delta}_{V_{mdg}}$ can be extended symbolically as $p^s_{mdg} : \Phi^{\delta}_{V_{mdg}} \rightarrow \mathrm{P}_{mdg}$, where V is the set of all variables occurring in p^s_{mdg} and $\Phi^{\delta}_{V_{mdg}}$ is the set of all δ-compatible assignments to the variables in V. The symbolic simple predicate p^s_{mdg} maps a δ-compatible assignment to the variables occurring in it to a simple predicate in P_{mdg} and works over V_{mdg}. The symbolic defining value $d^s_{p^s_{mdg}}$ of p^s_{mdg} is the symbolic extension of the defining value $d_{p_{mdg}}$ of p_{mdg}.

Definition 6.3.3.1: Given an STE model $M_{mdg} = ((\Phi^{\delta}_{V_{mdg}}, \leq_{mdg}), R^{\delta}_{suc})$ and a set P_{mdg} of simple predicates over $\Phi^{\delta}_{V_{mdg}}$, an MDG-based symbolic trajectory formula of model M_{mdg} is defined inductively as below:

- A symbolic simple predicate p^{s}_{mdg} is a MDG-based symbolic trajectory formula, where p^{s}_{mdg} is the symbolic extension of one of the simple predicates in P_{mdg}.
- The conjunction $(f^{s}_{mdg1} \wedge f^{s}_{mdg2})$ is a MDG-based symbolic trajectory formula if both f^{s}_{mdg1} and f^{s}_{mdg2} are MDG-based symbolic trajectory formulas.
- The next time expression $(\mathbf{N} f^{s}_{mdg})$ is a MDG-based symbolic trajectory formula if f^{s}_{mdg} is a MDG-based symbolic trajectory formula and **N** is the next-time operator.

In Forte, STE assertions are provided as an input in FL to the STE engine where the STE algorithm is implemented. The original MDG tools take as input only properties in L_{MDG} format. Therefore, an extra input port for the MDG-based STE assertions should be built for the MDG-STE engine in the MDG tools. As mentioned before, in Forte, the syntax for STE invocation is:

STE <model> <weak> <antecedent> <consequence> <trace>,

where the <antecedent>/<consequence>, representing a symbolic trajectory formula, is both of form a list of 5-tuples of the following format: (<guard>, <node>, <value>, <from>, <to>). The MDG tools may follow the similar format to specify the MDG-based STE assertions under the logic and the internal time-frame (clock) of the MDG-STE engine.

6.3.4 Implementation of STE Verification Methodology

In MDG-STE, the main task of verifying an assertion of form $[Ante^s_{mdg} \rightarrow Cons^s_{mdg}]$ is to check whether or not every MDG-based symbolic trajectory satisfying MDG-based symbolic trajectory formula $Ante^s_{mdg}$ also satisfies MDG-based symbolic trajectory formula $Cons^s_{mdg}$ and it can be implemented in this way:

- first compute the MDG-based defining symbolic trajectory $\chi^s_{Ante^s_{mdg}}$ and the MDG-based defining symbolic sequence $\beta^s_{Cons^s_{mdg}}$ for $Ante^s_{mdg}$ and $Cons^s_{mdg}$ respectively, and
- then check if $\chi^s_{Ante^s_{mdg}}$ is no less than $\beta^s_{Cons^s_{mdg}}$ over the symbolic partial order $\leq^s_{mdg}$ (symbolic extension of $\leq_{mdg}$) for any assignment to the symbolic variables.

Note that the above computation is bounded since it is easy to show that for a given MDG-based symbolic trajectory f^s_{mdg} with the defining sequence $\beta^s_{f^s_{mdg}} = \beta^{s0}_{f^s_{mdg}} \beta^{s1}_{f^s_{mdg}} \ldots$ we have $\beta^{si}_{f^s_{mdg}} = \perp^c$ for $i \geq dep(f^s_{mdg})$.

In STE, function lub^s is used in the definitions of the defining symbolic sequence and the defining symbolic trajectory. Similarly, in the MDG-STE engine, function lub^s_{mdg} is used in the definitions of the MDG-based defining symbolic sequence and the MDG-based defining symbolic trajectory. The function lub^s_{mdg} is the symbolic extension of the MDG-based lower upper bound function lub_{mdg}. Under the definition of the complete lattice $(S_{mdg}, \leq_{mdg})$, it is straight forward to implement the function lub_{mdg}.

Definition 6.3.4.1: Given an STE model $M_{mdg} = ((\Phi^{\delta}_{V_{mdg}}, \leq_{mdg}), R^{\delta}_{suc})$ and a set P_{mdg} of simple predicates over $\Phi^{\delta}_{V_{mdg}}$, the MDG-based defining symbolic sequence $\beta^{s}_{f^{s}_{mdg}}$ of a MDG-based symbolic trajectory formula f^{s}_{mdg} of M_{mdg} can be defined as follows:

- $\beta^{s}_{p^{s}_{mdg}} = d^{s}_{p^{s}_{mdg}} \perp^{c} \perp^{c} \ldots$ if $d^{s}_{p^{s}_{mdg}}$ is the symbolic defining value of p^{s}_{mdg}, where $\perp^{c}$ denote the constant function of $\perp$ and p^{s}_{mdg} is the symbolic extension of one of the simple predicates in P_{mdg},
- $\beta^{s}_{f^{s}_{mdg1} \wedge f^{s}_{mdg2}} = \mathrm{lub}^{s}_{mdg}(\beta^{s}_{f^{s}_{mdg1}}, \beta^{s}_{f^{s}_{mdg2}})$, and
- $\beta^{s}_{\mathrm{N} f^{s}_{mdg}} = \perp^{c} \beta^{s}_{f^{s}_{mdg}}$.

Definition 6.3.4.2: Given any MDG-based symbolic trajectory formula f^{s}_{mdg} of an STE model $M_{mdg} = ((\Phi^{\delta}_{V_{mdg}}, \leq_{mdg}), R^{\delta}_{suc})$, assuming that $\beta^{s}_{f^{s}_{mdg}} = \beta^{s0}_{f^{s}_{mdg}} \beta^{s1}_{f^{s}_{mdg}} \ldots$ is the MDG-based defining symbolic sequence for f^{s}_{mdg}, the MDG-based defining symbolic trajectory $\chi^{s}_{f^{s}_{mdg}} = \chi^{s0}_{f^{s}_{mdg}} \chi^{s1}_{f^{s}_{mdg}} \ldots$ of f^{s}_{mdg} can be defined inductively as follows:

$$\chi^{si}_{f^{s}_{mdg}} = \begin{cases} \beta^{s0}_{f^{s}_{mdg}} & \text{if } i = 0 \\ \mathrm{lub}^{s}_{mdg}(\beta^{si}_{f^{s}_{mdg}}, R_{suc}(\chi^{s(i-1)}_{f^{s}_{mdg}})) & \text{otherwise} \end{cases},$$

where R_{suc}, as defined previously in $A_{mdg} = (V_{mdg}, \leq_{mdg}, R_{suc})$, is the abstract description of the transition relation, i.e., the symbolic extension of R^{δ}_{suc}.

The pseudo-code shown in Figure 32 describes the algorithm **MDG_STE** for implementing the STE algorithm in the MDG-STE engine, where B_a, B_c, C_a, and N are MDG variables representing sets of states, K is the loop counter, **dep** is a function to calculate the depth of a MDG-based symbolic trajectory formula, **lub**, **DSS** and **ParO** are

MDG algorithms developed for MDG-STE, and **ReIP** is a basic MDG algorithm which is described in detail in [7]. The inputs of the **MDG_STE** algorithm are an abstract description $A = (V_{mdg}, \leq_{mdg}, R_{suc})$ of a STE model and an assertion $[Ante^s_{mdg} \rightarrow Cons^s_{mdg}]$, and the algorithm will return success/failure as a result.

```
1.    MDG_STE(A, Ante^s → Cons^s)
2.        loop (K := 0; K ≥ dep(Ante^s); K++)
3.            B_a := DSS(K, Ante^s);
4.            if (K = 0)
5.                then C_a := B_a
6.            else begin
7.                N := ReIP({C_a, R_suc}, V_mdg, ∅);
8.                C_a := lub(B_a, N))
9.            end;
10.           B_c := DSS(K, Cons^s);
11.           P := ParO(B_c, C_a);
12.           if (P = F) then exit and return failure;
13.       end loop;
14.       return success;
15.   end MDG_STE;
```

Figure 32. MDG_STE algorithm in the MDG-STE engine

Lines 2 to 13 specify the body of a loop. The loop will stop when the loop counter K is larger than the depth of the antecedent $Ante^s$. Note that the definition of depth of an MDG-based symbolic trajectory formula is similar to the one defined previously for an

STE formula and the antecedent $Ante^s$ and the consequent $Cons^s$ should be of the same depth.

In line 3, algorithm **DSS** constructs an MDG B_a representing the K-th element of the defining symbolic sequence of the antecedent $Ante^s$.

Lines 4 to 9 are used to compute the K-th element of the defining symbolic trajectory C_a of the antecedent $Ante^s$. The 0-th element of C_a equals B_a, and the other elements of will be computed using algorithm **ReIP** and **lub**. In Line 7, the relational product algorithm **ReIP** computes an MDG N representing the set of states reachable in one transition from C_a. In line 8, function **lub** computes the least upper bound of MDGs B_a and N over the partial order $\leq_{mdg}$ and assigns the result to C_a.

In line 10, algorithm **DSS** constructs an MDG B_c representing the K-th element of the defining symbolic sequence of the consequent $Cons^s$.

Lines 11 to 12 check if the assertion is violated at the K-th stage. In Line 11, algorithm **ParO** is used to compare B_c and C_a over the partial order $\leq_{mdg}$. Then in line 12, if the assertion is not satisfied, the whole algorithm will stop and returns failure.

In line 14, if no violations occur during the loop, the algorithm will return success.

6.4 Illustrative Example

We now present an illustrative example for the above algorithm of First-Order Symbolic Trajectory Evaluation using MDGs.

Consider the sequential circuit modeled in MDG-HDL shown in Figure 33, where

- input signals s_1 and s_2 of abstract sort 'wordn',
- output signal s_4 is of abstract sort 'wordn',

- components *reg1* and *reg2* are registers, and
- component *abs_and* is an abstract function of type '*wordn* * *wordn* → *wordn*'.

Note that in the MDG tools the clock signal is implemented implicitly and no input clock signal is needed.

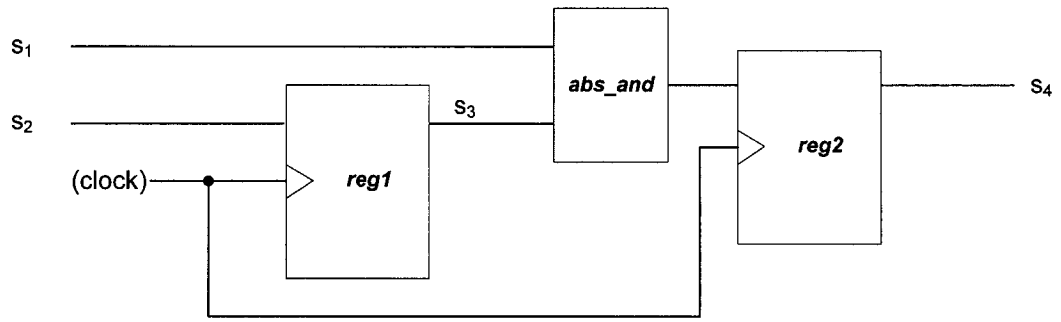


Figure 33. Diagram for a MDG-HDL model of a sequential circuit

We represent the circuit state as a 4-bit vector $s = s_1 s_2 s_3 s_4$. The next state function $R_T = t_1 t_2 t_3 t_4$ is defined as follows:

$$t_1(s_1) = X^c, \quad t_2(s_2) = X^c, \quad t_3(s_3) = s_2, \quad t_4(s_4) = abs\_and(s_1, s_3).$$

As mentioned before, no next state constraint is imposed on a state vector component associated with an input signal, the next state function of which should be X^c.

Assume that we want to verify the following symbolic trajectory assertion for the above circuit model:

$$((s_1 = a) \wedge (s_2 = b)) \wedge \mathrm{N}(s_1 = a) \rightarrow \mathrm{N}^2(s_4 = abs\_and(a,b)).$$

By using Definition 11 and 12 in the previous section, we first show in Table V the computation process for the symbolic defining sequence and the symbolic defining trajectory of the antecedent:

$$Ante^s_{mdg} = ((s_1 = a) \wedge (s_2 = b)) \wedge \mathrm{N}(s_1 = a).$$

TABLE V. SYMBOLIC DEFINING TRAJECTORY OF THE ANTECEDENT

i	$\beta^{si}_{Ante^{s}_{mdg}}$					$R_{suc}(\chi^{s(i-1)}_{Ante^{s}_{mdg}})$					$\chi^{si}_{Ante^{s}_{mdg}}$			
	s_1	s_2	s_3	s_4		s_1	s_2	s_3	s_4		s_1	s_2	s_3	s_4
0	a	b	X^c	X^c							a	b	X^c	X^c
1	a	X^c	X^c	X^c		X^c	X^c	b	abs_and (a, X^c)	lub^{s}_{mdg}	a	X^c	b	abs_and (a, X^c)
2	X^c	X^c	X^c	X^c		X^c	X^c	X^c	abs_and (a, b)	lub^{s}_{mdg}	X^c	X^c	X^c	abs_and (a, b)
≥3	X^c	X^c	X^c	X^c		X^c	X^c	X^c	abs_and (X^c, X^c)	lub^{s}_{mdg}	X^c	X^c	X^c	abs_and (X^c, X^c)

TABLE VI. SYMBOLIC DEFINING SEQUENCE OF THE CONSEQUENT AND COMPARISON

i	$\beta^{si}_{Cons^{s}_{mdg}}$					$\chi^{si}_{Ante^{s}_{mdg}}$			
	s_1	s_2	s_3	s_4		s_1	s_2	s_3	s_4
0	X^c	X^c	X^c	X^c	$\leq^{s}_{mdg}$	a	b	X^c	X^c
1	X^c	X^c	X^c	X^c	$\leq^{s}_{mdg}$	a	X^c	b	abs_and (a, X^c)
2	X^c	X^c	X^c	abs_and (a, b)	$\leq^{s}_{mdg}$	X^c	X^c	X^c	abs_and (a, b)
≥3	X^c	X^c	X^c	X^c	$\leq^{s}_{mdg}$	X^c	X^c	X^c	abs_and (X^c, X^c)

The computation process for the symbolic defining sequence of the consequent $Cons^{s}_{mdg} = (\mathrm{N}^2(s_4 = abs\_and(a,b))$ is then shown in Table VI, compared with the result for the defining trajectory of the antecedent. We can easily see from the table that: $\beta^{s}_{Cons^{s}_{mdg}} \leq^{s}_{mdg} \chi^{s}_{Ante^{s}_{mdg}} = 1^c$, i.e., the symbolic trajectory assertion is satisfied by the circuit model under all variable assignments.

6.5 Summary

In this chapter, we investigated the possibility of integrating Symbolic Trajectory Evaluation and MDG-based model checking. Two attempts to combine these two techniques have been discussed and finally we proposed a hybrid approach of performing first-order Symbolic Trajectory Evaluation using MDGs. In the next and last Chapter of this thesis, we will conclude and discuss our future work.

Chapter 7

Conclusion and Future Work

Traditional BDD-based symbolic model checking techniques are an attractive subset of formal verification methods because of their high automation and little requirement for human effort to guide the proof process, whereas they usually suffer from the state explosion problem. Symbolic Trajectory Evaluation technique and MDG-based model checking technique are an improvement over the traditional BDD-based symbolic model checking approaches in two different ways. The first one can dramatically reduce the computations for the next state space and enhance the computational efficiency, while the latter one can simplify the data path operations and thus can effectively overcome the state explosion problem. If we can combine these two techniques, it is possible for us to take the advantages of both of them.

In this thesis, we investigated the possibility of integrating Symbolic Trajectory Evaluation and MDG-based model checking. For each of the approaches, we studied the underlying theory and methodology, offered an illustrative example, discussed the verification tool, and provided a detailed case study. The main purpose of these two case studies is to obtain an in-depth understanding of the underlying theories and methodologies of these two model checking techniques, which may facilitate the achievement of their combination. Two attempts to combine the Symbolic Trajectory

Evaluation with the MDG were discussed: one in the STE verification environment and the other in the MDG tools.

The goal of the first hybrid approach is to encode the symbolic expressions in Forte using the MDG package. We proposed theorically two methods to integrate the MDG package in Forte, either by programming the MDG package directly in FL or by programming the MDG package in a language other than FL and accessing the MDG package using FL's foreign function interface. However, it was practically impossible for us to implement the above two methods because we didn't have the access to the FL source code.

In the second hybrid approach, we developed an MDG-STE engine in the MDG tools where the STE algorithm were implemented using the MDG package. We first extended the many-sorted first-order logic underlying MDGs by adding the feature of "don't care" value X and then discussed respectively the implementations of STE modeling (including the construction of a complete lattice), STE assertions and STE verification methodology using MDGs. An illustrative example for the algorithm of First-Order Symbolic Trajectory Evaluation using MDGs was given at the end. This proposed hybrid approach can not only increase the scale of circuits verified using STE but also improve the performance of STE by raising the level of abstraction.

As future work, we consider the following research directions:

- fixing the bugs found in the MDG tools during our case study discussed in Chapter 5;
- implementing the algorithm of *First-Order Symbolic Trajectory Evaluation* in the MDG tools;

- proving the correctness of our proposed *first-order Symbolic Trajectory Evaluation* algorithm;
- developing a RTL level Verilog to MDG-HDL converter to facilitate the verification using the MDG model checker;
- performing equivalence checking between the RTL level and the gate level MDG-HDL models of the LA-1 Interface, which may involve the development of a gate level Verilog to MDG-HDL converter.

References

[1] A. Habibi, A.I. Ahmed, O. Ait-Mohamed, and S. Tahar. On the Design and Verification of the Look-Aside Interface. In Proc. IEEE/ACM Design Automation and Test in Europe (DATE'05), pages 649–654, Munich, Germany, March 2005.

[2] A. Pnueli. The temporal logic of programs. In 18th IEEE Symposium on Foundation of Computer Science, pages 46-57, 1977.

[3] A.I. Ahmed and O. Ait-Mohamed. Assertion-Based Verification of Look-Aside Interface (LA-1 Standard). Technical Report TK 7887.5 A36, Concordia University, Department of Electrical and Computer Engineering, June 2004.

[4] C.-J.H. Seger and R.E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. In J. Formal Methods in Syst. Design, vol 6, pages 147-189, Mar. 1995.

[5] D.L. Beatty, R.E. Bryant, and C.-J.H. Seger. Synchronous circuit verification by symbolic simulation: An illustration. In Sixth MIT Conference on Advanced Research in VLSI, pages 98-112,1990.

[6] D.L. Beatty, R.E. Bryant, and C.-J.H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In Proc. 1991 IEEE/ACM Design Automation Cont, pages 397-402, June 1991.

[7] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar and Z. Zhou. Automated Verification with Abstract State Machines using Multiway Decision Graphs. In

Formal Hardware Verification: Methods and Systems in Comparison, T. Kropf (eds.), Springer Verlag, pages 79-113, 1997.

[8] E.M. Clarke, E.A. Emerson and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. In ACM Transactions on Programming Languages and Systems, vol 8(2), pages 244-263, April, 1986.

[9] E.W. Weisstein et al. Partially Ordered Set. From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/PartiallyOrderedSet.html.

[10] F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. In Formal Methods in System Design, vol 10(1), pages 7-46, February 1997.

[11] J.H. Gallier. Logic for Computer Science: Foundations of Automatic Theorem Proving. Harper & Row, 1986.

[12] J.M. Scott. Efficient Verification of Multi-Processor Real-Time Systems Using Symbolic Methods. PhD thesis, Vanderbilt University, Nashville, Tennessee, 2003.

[13] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In Proceedings of the 27th ACM/IEEE Design Automation Conference, pages 46-51, IEEE Computer Society Press, June 1990.

[14] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. In Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, pages 428 – 439, June 1990.

[15] L. Lamport. Sometimes is sometimes ``not never'' - on the temporal logic of programs. In Proceedings of 7th ACM Symposium on Principles of Programming Languages, pages 174-185, 1980.

[16] M. Pandey, R. Raimi, R.E. Bryant, and M.S. Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. In Annual ACM IEEE Design Automation Conference, pages 649–654, Las Vegas, Nevada, United States, 1996.

[17] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, Springer-Verlag, June 1989.

[18] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages, pages 97-107, Association for Computing Machinery, January 1985.

[19] R.A. Finkel. Advanced Programming Language Design. Addison-Wesley Publishing Company, 1996.

[20] R.E. Bryant. Graph-based algorithms for boolean function manipulation. In IEEE Transactions on Computers, pages 677-691, August 1986.

[21] R.E. Bryant. Formal Verification of Memory Circuits by Switch-Level Simulation. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol 10(1), pages 94-102, January 1991.

[22] R.E. Bryant. Symbolic simulation-techniques and applications. In Annual ACM IEEE Design Automation Conference, pages 517–521, Orlando, Florida, United States, 1991.

[23] R.E. Bryant, C.-J.H. Seger. Formal Verification of Digital Circuits Using Symbolic Ternary System Models. In Proceedings of the 2nd International Workshop on Computer Aided Verification, pages 33-43, June 1990.

[24] S. Bose and A. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, 1989.

[25] S. Gnesi. Model checking of embedded systems. http://www.ercim.org, 2003.

[26] S. Romanenko et al. Moscow ML Language Overview, Version 2.00 of June 2000. http://www.dina.kvl.dk/~sestoft/mosml/mosmlref.pdf.

[27] Y. Xu. MDG Model Checker User's Manual, Dept. of Information and Operational Research, University of Montreal, Montreal, Canada, September 1999.

[28] Y. Xu. Model checking for a first-order temporal logic using multiway decision graphs. Ph.D. Thesis, Universite de Montreal (Canada), 1999.

[29] Y. Xu, E. Cerny, X. Song, F. Corella, and O. Ait-Mohamed. Model checking for a first-order temporal logic using multiway decision graphs. In Proc. of the International Conference on Computer-Aided Verification (CAV'98), Lecture Notes in Computer Science 1427, pages 219-231, 1998.

[30] Z. Zhou and N. Boulerice. MDGs Tools (V1.0) User's Manual, D'IRO, University of Montreal, June 1996.

[31] Intel Corporation. EXLIF: Extended Logic Interchange Format Syntax. http://cc.usu.edu/~saraswat/exlif_syntax.pdf, 2003.

[32] Intel Corporation. Forte/FL User Guide, Version 1.0. Intel Technical Publications and Training, Jan 2003.

[33] Intel Corporation. Introduction to Forte Verification Environment. http://www.intel.com/technology/silicon/scl/fortefl.htm.

[34] Network Processing Forum. Look-Aside (LA-1) Interface, Implementation Agreement, Revision 1.1. Kluwer Academic Publishers, April 15, 2004.

[35] IT University of Copenhagen. Research project description of Muddy. http://www.itu.dk/research/muddy/.

[36] Xilinx, Inc. Synthesis and Simulation Design Guide. Xilinx Manuals Online. http://toolbox.xilinx.com/docsan/xilinx4/data/docs/sim/vtex5.html.

[37] Xilinx, Inc. Virtex Delay-Locked Loops. Virtex Tech Topic VTT003 (v1.1). http://www.xilinx.com/products/virtex/techtopic/vtt003.pdf, August 7, 2000.

[38] Webpage for Jørn Lind-Nielsen. http://www.itu.dk/people/jln/.