Université de Montréal

Integrating MDG Variable Ordering in a VHDL-MDG Design Verification System

par

Yi Feng

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures

en vue de l'obtention du grade de

Philosophiæ Doctor (Ph.D.)

en Informatique

novembre, 2001

Université de Montréal

Faculté des études supérieures

Cette thèse intitulée

Integrating MDG Variable Ordering in a VHDL-MDG Design Verification System

présentée par

Yi Feng

a été évaluée par un jury composé des personnes suivantes:

Professor Marc Feeley — président-rapporteur

Professor Eduard Cerny — directuer de recherche

Professor Hanifa Boucheneb — membre du jury

Professor Xiaoyu Song — examinateur externe

Gary Boucher — représentant du doyen de la FES

Résumé

Les méthodes de vérification formelle consistent en l'utilisation des techniques analytiques pour prouver que l'implantation d'un système se conforme bien à la spécification. Comme une représentation efficace de Machines à États Finis Étendues, les *Multiway Decision Graphs* (MDG) sont appropriés pour la vérification formelle automatique du matériel pour la conception au niveau transfert de registres (*Register Transfer Level*).

Pour réduire l'effet du problème d'explosion d'états dans un système basé sur un MDG, nous présentons des algorithmes statiques et dynamiques de l'ordonnancement automatique des variables. L'ordonnancement des variables sur MDG est plus compliqué que celui sur ROBDDs à cause de la présence des termes du premier ordre dans MDG.

L'algorithme de l'ordonnancement des variables statiques génère un ordre des variables avant qu'un MDGs soit construit et le choix de l'ordre dépend de l'information sur la topologie du circuit en considération. L'algorithme de re-ordonnancement dynamique minimise la taille des MDGs au cours du processus de vérification et permet de terminer en succès une tâche de vérification qui peut échouer avec un ordre fixe. Nous discutons aussi du problème de l'ordonnancement des termes standards causé par l'ordre lexicographique que les MDGs utilisent pour ordonnancer certains termes. Une solution est présentée qui est basée sur le re-étiquetage des fonctions et la reécriture des termes.

Le MDG devient plus efficace avec le développement de l'ordonnancement automatique de variables. Mais, un système MDG n'accepte que le langage MDG-HDL. Nous avons alors développé un traducteur qui accepte un modèle décrit en

VHDL, au niveau RTL synthétisable, et qui produit la représentation requise pour le système MDG.

Nous avons intégré les algorithmes de traduction et ceux de (re)ordonnancement dans le système de vérification de conception basé sur MDG. L'ordonnancement efficace des variables est essentielle pour le bon fonctionnement du système et la traduction automatique du VHDL au MDG-HDL rend possible la vérification des conceptions industrielles décrites en VHDL. Les résultats expérimentaux ont démontré que ce système amélioré est capable de traiter une classe de modèles plus large que celui sans l'introduction de ces améliorations. Donc notre système étend la classe des circuits vérifiables.

Mot clés :

- vérification formelle
- Graphes de Décision Multidirectionnels
- intégration de logiciels
- ordonnancement des variables
- traduction automatique de VHDL
- vérification de propriétés

Abstract

Formal verification methods involve the use of analytical techniques to prove that the implementation of a system conforms to the specification. As an efficient representation of Extended Finite State Machines, Multiway Decision Graphs (MDG) are suitable for automatic hardware formal verification of Register Transfer Level (RTL) designs.

To reduce the effects of the state explosion problem in the MDG-based system, we explore automatic static and dynamic variable ordering algorithms. Compared with ROBDDs, the situation is complicated by the presence of first order terms in MDGs.

The static variable ordering algorithm generates a variable order before an MDG is built and the order is chosen using information about the circuit topology. The dynamic reordering algorithm minimizes the size of the MDG during the verification process and allows a verification task to finish when the task may not complete with a fixed order due to insufficient memory or execution time. We also identify a standard term ordering problem caused by the standard term order used in MDG to order some specific terms. A solution based on function renaming and term rewriting is presented.

The MDG system has become more efficient with the development of automatic variable ordering. However, the system only accepts MDG-HDL and most of designs in industry are described in VHDL or Verilog. We present a translator which can accept a VHDL model given at the synthesizable RTL level and produce the required representation by the MDG system.

We integrated the translation and the (re)ordering algorithms into the MDG design verification system. Efficient variable ordering is essential for good functioning of the verification system and automatic translation from VHDL to MDG-HDL makes possible the verification of industrial designs. Experimental results proved that this updated system can handle a larger class of designs than before, thus alleviating the effects of the state exploration problem and increasing the range of the circuits that can be verified.

Key words:

- formal verification
- Multiway Decision Graphs
- software integration
- variable ordering
- automatic translation of VHDL
- property checking

Table of Contents

List of Tables

List of Figures

List of Initials and Abbreviations

ASM	Abstract State Machine
ATM	Asynchronous Transfer Mode
BTTL	Branching Time Temporal Logic
CDFG	Control/Data Flow Graphs
CTL	Computation Tree Logic
DAG	Directed Acyclic Graph
DF	Directed Formula
DNF	Disjunctive Normal Form
EFSM	Extended Finite State Machine
FSM	Finite State Machine
HDL	Hardware Description Language
ITC	Island Tunnel Controller
LTL	Linear Temporal Logic
MDG	Multiway Decision Graph
MDG-HDL	Multiway Decision Graph-Hardware Description Language
PbyS	Pruning by subsumption
ROBDD	Reduced Ordered Binary Decision Graph
RTL	Register Transfer Level
VHDL	VHSIC Hardware Description Language

Acknowledgements

First and foremost, I would like to express my heartfelt thanks to my supervisor, Dr. Eduard Cerny, for his extensive time, extreme patience, valuable suggestions and constant encouragement during my entire doctoral studies. It is he who looks after me as an international student, academically, financially and socially, always with great responsibility, which I appreciate so much and will always remain deeply in my memory.

I would also like to thank Michel Reid for his technical and administrative support, which is invaluable in working for this thesis. Many thanks to my friends and fellow graduate students in Lab Lasso, Jin Hou, Ying Zhang, and Mohammad-Sadegh Jahanpour for their help.

Last, but not least, I would like to thank all my family members: my parents, my sister, brother, and my husband for their full support and encouragement for my studies.

Chapter 1 Introduction

Technological advances in microelectronics have increased the complexity of digital hardware designs. Their correctness thus becomes a major concern, especially in critical applications where failure is unacceptable. Traditionally, the task of design validation is carried out by means of simulation. In a simulation-based approach, the designer needs to create a set of test vectors that represents the possible inputs to the system. The output for each of these test vectors is compared with the expected response. This method is very costly and incomplete because of the large number of input sequences to consider. In almost all practical situations it is infeasible to exhaustively simulate a design to guarantee its correctness.

Complement to simulation is the use of formal verification. Formal verification methods intend to establish that an implementation satisfies a specification by mathematical reasoning [33]. In this thesis, we refer to an implementation as the hardware design to be verified, and a specification as the property with respect to which the correctness of the implementation is to be determined. Formal verification conducts an exhaustive exploration of all possible behaviors. Thus, when a design is pronounced correct by a formal verification method, it implies that all behaviors relative to the property have been explored [14].

In this introduction, we first present the background for this thesis. The related research is introduced next. We then summarize the scope of this thesis in Section 1.3, and give an outline of the thesis in Section 1.4.

1.1 Background

Formal verification methods can be classified in two main categories: interactive verification with a theorem prover and automated Finite State Machine (FSM) verification based on state enumeration [33].

Interactive verification with a theorem prover uses a powerful formalism such as higher-order logic that allows the verification problem to be stated at many levels of abstraction. This approach has achieved significant successes in verifying microprocessor designs. However, interactive verification has the drawback that the user is responsible for coming up with the proof of correctness and feeding it to the theorem prover, which requires great expertise.

Automated FSM verification based on state enumeration techniques provides automation for behavior comparison and model checking. Model checking works on a finite-state model of the system to be verified, and the logical specification of the desired behavior of the system model. Since model checking can be completely automatic and has been used successfully to verify complex sequential circuit designs and communication protocols, it is beginning to be used in industry.

A model checking approach to formal verification is based on exploring the reachable state space of its model. Finite state models of concurrent systems grow exponentially as the number of components of the system increases. This is known as the *state explosion* problem in automatic verification. The main challenge in model checking is dealing with this problem.

The most promising approach to the state explosion problem has been the application of ROBDD (Reduced Ordered Binary Decision Diagrams) to the representation of state graphs [7]. ROBDD can encode sets of states as well as transition and output relations, and perform an implicit enumeration of the state

space, thus making it possible to verify finite state machines with a larger number of states. Since the size of an ROBDD is largely influenced by the choice of the variable order, several approaches have been researched to find a good variable order [9, 21, 28, 29, 45, 55, 57].

For automated hardware verification, ROBDDs have proved to be a powerful tool. However, because they require a Boolean representation of the circuit, the size of an ROBDD grows, sometimes exponentially, with the number of Boolean variables. Therefore, ROBDD-based verification cannot be directly applied to circuits with complex and large datapaths.

To overcome this limitation, the verification group at the University of Montreal has proposed a new class of decision graphs called Multiway Decision Graphs (MDG). MDGs efficiently represent a class of formulas of a many-sorted first-order logic with a distinction of abstract and concrete sorts [16]. In an MDG, a data signal is represented by a single variable of abstract sort rather than by a vector of Boolean variables, and a data operation is represented by an uninterpreted function symbol. MDGs compactly encode sets of (abstract) states and transition/output relations for abstract description of state machines. The implicit enumeration technique is lifted from the Boolean level to the abstract level and referred to as implicit abstract enumeration. MDGs are thus much more compact than ROBDDs for circuits having complex and large datapaths. This greatly increases the range of circuits that can be verified.

However, both MDG-based and ROBDD-based verification systems still suffer from the problem of state explosion when handling realistic systems. To reduce the effects of this problem, one of the most important approaches is to select a good variable order. Like ROBDD, the size of the MDG heavily depends on its variable order. A good variable order can keep the size of MDG as small as possible and reduce the memory requirement and processing time.

The concept of ordering in MDG concerns two orders: the standard term order and the custom symbol order. The standard term order is a total order of all the terms of the logic. The custom symbol order is a total order of a set of symbols that includes the concrete variables, abstract variables, and some but not necessarily all of the operators. The custom symbol order does not need to be compatible with the standard term order.

In this thesis, automatic static and dynamic variable ordering (custom symbol ordering) algorithms for MDG are explored. They are much more complicated than ROBDD because of the presence of first order terms in MDG. A standard term ordering problem is also identified and a solution is proposed. These algorithms and solutions are integrated into a VHDL-MDG design verification system.

1.2 Related Research

Many researchers have been working on the state explosion problem in automatic formal verification. In this section we describe another approach called model reduction. It has been implemented on MDG [34]. The research on variable ordering on ROBDD will also be presented.

1.2.1 Model Reduction

A reduction method reduces the correctness problem to a similar problem with a smaller state space. This is generally done by replacing processes in the model by smaller processes that have similar or identical communication behaviors. The common reduction methods on ROBDDs include homomorphic reduction in

language containment tests [43, 65], structural symmetry exploration [23] and partition refinement [18, 19].

Hou and Cerny have presented a model reduction algorithm for property checking [34]. For the property to be verified, the algorithm first constructs a property dependency graph that represents the function dependency of the property on the state variables. Starting from the set of state variables appearing in the property, the algorithm searches through the dependency graph and adds a non-correlated set of state variables to the current set of state variables to construct a more detailed model at each reduction iteration step [35]. This reduction algorithm is completely automatic and has been implemented on MDG. Experiments show that this method can achieve efficient reduction on benchmark circuits and has significantly increased the useful domain of MDG.

Model reductions and variable ordering both reduce the state space to alleviate state explosion and increase the number of circuits that can be verified. However, they each contribute in their own ways. Model reduction tries to build a smaller model based on the property to be verified. Variable ordering reduces the size of a decision graph by choosing a good variable order.

1.2.2 Variable Ordering on ROBDD

Bryant first presented the variable ordering problem in his landmark paper on ROBDD in 1986 [7]. Since then, there have been many researchers working on this problem. It is usually classified into three categories: static variable ordering to find an appropriate order before generating an ROBDD by using the logic circuit information that is the source of the Boolean function to be represented; dynamic variable ordering to reduce the ROBDD size by permuting the variables of a given ROBDD starting with the initial static order; and finally optimal variable ordering to find the best order for the ROBDD. It has been proven that

finding the optimal order is co-NP complete [3]. Many variable ordering algorithms produce acceptable results, so it is not necessary to find the optimal order. The most successful static ordering algorithm was proposed by Fujita in 1988 by minimizing the number of crosspoints of nets when the circuit graph is drawn [28]. Rudel first introduced a dynamic algorithm based on variable exchanges, called sifting [57]. Sifting allows many ROBDD operation sequences to succeed, when they would have failed with a fixed variable order. However, sifting is extremely expensive in both time and space.

1.3 Scope of the Thesis

This thesis explores static and dynamic variable ordering algorithms on MDG. Compared to ROBDD, ordering on MDG has to deal with the constraints caused by abstract variables and uninterpreted function symbols. In this thesis an efficient static ordering method for combinational and sequential circuits and a dynamic ordering method for choosing an order in the middle of the verification process are proposed. Moreover, because some special decision nodes adopt standard term ordering, this situation may cause state explosion for circuits which have certain specific structures. This is called the standard term ordering problem. We propose a solution using function renaming and rewriting rules. All these methods make MDG work more efficiently.

Our MDG-based verification system only accepts a Prolog-style HDL, MDG-HDL, which allows the use of abstract variables for representing data signals. We developed a software to translate a VHDL model to MDG-HDL. The translator accepts a VHDL model as input and produces the representation for the MDG system. The VHDL model must be given at the synthesizable Register Transfer Level (RTL). Each VHDL process is transformed into a Directed Acyclic Graph (DAG) [58]. Then, MDG-HDL is generated from this DAG. This translator and

variable ordering make the MDG verification system more suitable for a large class of problems than before the introduction of these improvements.

My contributions in this thesis are as follows:

1. The development and implementation of automatic static variable ordering algorithms on MDG.
2. The development and implementation of dynamic variable ordering algorithms on MDG.
3. The identification of and a solution to the standard term ordering problem.
4. Automatic translation from DAG to MDG-HDL.
5. Integration of automatic variable ordering with the MDG package.
6. Experiments on benchmark circuits.

1.4 Outline of the Thesis

This thesis discusses formal verification techniques, MDG-based verification approaches, variable ordering algorithms on MDG and a translation between VHDL and MDG-HDL. The thesis is organized as follows:

In Chapter 2, we review several formal hardware verification techniques.

In Chapter 3, we begin with describing the basic concepts of MDG. We then present MDG-based verification approaches.

In Chapter 4, we discuss the static variable ordering algorithms on MDG and the constraints on variable ordering that MDGs impose. It is difficult to derive a method that always yields the best order to minimize MDGs, but with our static algorithms, we can find a fairly good order in most cases.

In Chapter 5, we present dynamic variable ordering algorithms on MDG. When using a fixed static variable order, some MDG operations may run out of memory. Dynamic variable ordering allows these operation sequences to succeed when a new order is chosen mid-stream.

In Chapter 6, we start by explaining what the standard term ordering problem is. We then propose a solution based on function renaming and rewriting.

In Chapter 7, a translation method from VHDL to MDG-HDL is presented.

Finally, conclusions and future directions of research are stated in Chapter 8.

Chapter 2 Formal Hardware Verification

Several approaches to formal verification have been proposed over the years. This chapter concentrates on the method called model checking by which a desired behavioral property of a reactive system is verified through exhaustive enumeration (explicit or implicit) of all the states reachable by the system and the behaviors that traverse though them [14]. We first review several modeling languages of design systems. We then describe model checking property specifications and systems. Finally, we discuss how to represent finite state reactive systems symbolically using Reduced Ordered Binary Decision Diagrams (ROBDDs) and review variable ordering on ROBDDs. This chapter introduces the related research and provides the theoretical basis for the subsequent chapters.

2.1 Modeling Languages

At their most detailed level, digital systems may consist of millions of elements, as would be the case if we view a system as a collection of logic gates or transistors. From a more abstract viewpoint, these elements may be grouped into a handful of functional components such as cache memories, floating-point units, signal processors, or real-time controllers. Hardware description languages have evolved to aid in the design of systems with this large number of elements with a wide range of electronic and logical abstractions [63]. Different design and verification systems may have different system description languages. Here we will review some of them, including Verilog, VHDL, SMV input language, Synchronous Verilog, and MDG-HDL.

2.1.1 Verilog and VHDL

Verilog and VHDL are hardware description languages used to design and document electronic systems. They allow a designer to describe designs at a high level of abstraction such as at the architectural or behavioral level as well as the lower implementation levels (gate and transistor switch) [12].

Verilog and VHDL describe a digital system as a set of modules. Each of these modules has an interface to other modules as well as a description of its contents. A module represents a logical unit that can be described either by specifying its internal logical structure - for instance describing the interconnection of the actual logic gates it is comprised of, or by describing its behavior in a program-like manner - in this case focusing on what the module does rather than on its logical implementation. These modules can be interconnected with signals, allowing them to communicate.

The notion of a process plays a central role in VHDL and Verilog. All time-dependent behavior is defined in terms of process statements. A process can be thought of as an independent thread of control, which may be quite simple, involving only one repeated action, or very complex, resembling a software program. It might be implemented as a sequential state machine, as a microcoded controller, as an asynchronous clearing of a register, or as a combinational circuit.

2.1.2 SMV Input Language and Synchronous Verilog

The SMV system is a formal verification tool for checking finite state systems against specifications in the temporal logic CTL [47]. The input language of SMV is designed to allow the description of finite state systems that range from completely synchronous to completely asynchronous, and from the detailed to the abstract. One can readily specify a system as a synchronous Mealy machine, or as

an asynchronous network of abstract, nondeterministic processes. The language provides for modular hierarchical descriptions, and for the definition of reusable components. Because it is intended to describe finite state machines, the only data types in the language are finite ones: Booleans, scalars and fixed arrays. Static, structured data types can also be constructed. The primary purpose of the SMV input language is to describe the transition relation of a finite Kripke structure. Any expression in the propositional calculus can be used to describe this relation. The logic CTL allows a rich class of temporal properties, including safety, liveness, fairness constraints and absence of deadlock, to be specified using a concise syntax.

Those who are familiar with the Verilog modeling language may find it easier to write models for SMV in Synchronous Verilog (SV). This language is syntactically only a slight variation of the Verilog language. However, its semantics is not based on an event queue model, as in Verilog. Rather, SV is a synchronous language, in the same family as SMV [49]. Because SV provides a functional description of a design rather than an operational description of how to simulate it, SV is better suited than Verilog to such applications as hardware synthesis, cycle-based (functional) simulation and model checking [48].

2.1.3 MDG-HDL

Synchronous RT (Register-Transfer) level hardware designs can be suitably represented by Multiway Decision Graphs (MDGs), a class of decision graphs that subsumes the class of Reduced Ordered Binary Decision Graphs (ROBDDs) while accommodating abstract sorts and uninterpreted function symbols. The MDG tools are a prototype implemented in Prolog for the verification of RTL designs. They are intended for the verification of abstract descriptions of state machines rather than Finite State Machines (FSM). An abstract description of a state machine, called abstract state machine (ASM), is obtained by letting some data input, state

or output variables be of an abstract sort, and the operations on them be uninterpreted function symbols.

The hardware description language that MDG tools accept is a Prolog-style HDL, MDG-HDL, which allows the use of abstract variables for representing data signals. The MDG-HDL description is then compiled into the ASM synchronous model in internal MDG data structures. MDG-HDL supports structural descriptions, behavioral ASM descriptions, or a mixture of structural and behavioral descriptions. A structural description is usually a netlist of components (predefined in MDG-HDL) connected by signals. A behavioral description is given by a tabular representation of the transition/output relation or by a truth table [70].

2.2 Model Checking Property Specifications and Systems

Model checking is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. Specifications are expressed in a propositional temporal logic, and the reactive system is modeled as a state-transition graph. An efficient search procedure is used to determine whether or not the state-transition graph satisfies the specifications. A model checking system is described in Figure 2.1. This technique was originally developed by Clarke and Emerson in 1981 [14].

Model checking has several important advantages over mechanical theorem provers for verification of circuits and protocols. The most important is that the procedure is highly automatic. In general, a model checker builds or accepts a finite-automaton model of the system and checks whether or not the specified property holds on the model. If it does not, the model checker returns a failure trace. Normally, the property is expressed in a temporal logic which we describe next.

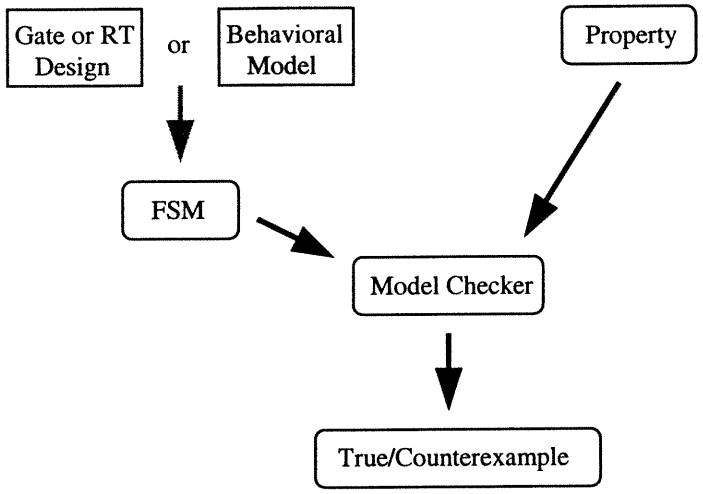


Figure 2.1 A model checking system

2.2.1 Temporal Logics

A temporal logic is a formalism for describing sequences of transitions between states in a reactive system. It provides a formal system for qualitatively describing and reasoning about how the truth values of assertions change over time [33]. There are four basic operators in temporal logic:

- $\Box P$ is true in state s, if P is true in all future states from s (including s).
- $\Diamond P$ is true in state s, if P is true in some future states from s.
- OP is true in state s, if P is true in the next state from s.
- PUQ is true in state s, if either Q is true in s itself, or it is true in some future state of s, and until then P is true at every intermediate state.

The following three classes of properties can be easily expressed in temporal logic:

- *Safety properties* - assert that nothing "bad" happens, typically represented as $\models \Box P$, i.e., P holds at all times in all models;

- *Liveness properties* - assert that eventually something "good" happens, typically represented as $|= P \Rightarrow \Diamond Q$, i.e., in all models, if P is initially true then Q will eventually be true;
- *Precedence properties* - assert the precedence order of events, typically represented as $|= PUQ$, i.e., in all models, P will hold until Q becomes true.

Based on the difference in viewing the notion of time, temporal logics can be classified into two kinds. In one case, time is characterized as a single linear sequence of events, leading to *Linear Time (Temporal) Logic*. In the other case, a branching view of time is taken, such that at any instant there is a branching set of possibilities into the future. This view leads to *Branching Time (Temporal) Logic*.

2.2.2 Propositional Linear Temporal Logic

In a linear temporal logic the underlying structure of time is assumed to be isomorphic to the natural numbers with their usual order (N, <)[22]. Let *AP* be an underlying set of atomic proposition symbols. A linear-time structure *M=(S, x, L)* is defined such that

- *S* is a set of states,
- *x: N-> S* is an infinite sequence of states, and
- *L: S-> $2^{(AP)}$* is a labeling of each state with the set of atomic propositions in *AP* that are true in the state.

Usually, the notation $x = (s_0, s_1, s_2,...) = (x(0), x(1), x(2), ...)$ is employed to denote the *timeline* x, which is also referred to as a *fullpath*, or *computation sequence*, or *computation*.

The basic temporal operators of a Propositional Linear Temporal Logic (PLTL) are **F***p* ("sometimes *p*", also read as "eventually *p*"), **G***p* ("always *p*", also read as "henceforth *p*"), **X***p* ("nexttime *p*"), and *p* **U** *q* ("*p* until *q*") [33].

2.2.3 Computation Tree Logic

Different kinds of Branching Time Temporal Logic (BTTL) have been proposed depending on the exact set of operators allowed. The common feature is that they are interpreted over branching tree-like time structures, where each moment may have many successor moments. The structure of time corresponds to an infinite tree. The usual temporal operators (**F**, **G**, **X**, **U**) are regarded as state quantifiers. Additional quantifier called the path quantifier is provided to represent all path (**A**) and some path (**E**) from a given state. Here we only describe the Computation Tree Logic (CTL), a restricted form of BTTL.

Clarke and Emerson first proposed CTL and presented efficient algorithms for CTL model checking within a larger framework of automatic synthesis of synchronization skeletons from CTL specifications [14].

CTL severely restricts the type of formulas that can appear after a path quantifier - only single linear time operator **F**, **G**, **X**, or **U** can follow a path quantifier and time operators cannot be combined directly with propositional connectives. The syntax of CTL is:

- Every atomic proposition is a CTL formula.
- If *f* and *g* are CTL formulas, then so are $\neg f$, $(f \wedge g)$, **AX***f*, **EX***f*, **A**(*f***U***g*), **E**(*f***U***g*)

The remaining operators are derived from these according to the following rules:

$f \vee g = \neg(\neg f \wedge \neg g)$

AF*g* = **A**(true **U** *g*)

$\mathbf{EF}g = \mathbf{E}(\text{true}\ \mathbf{U}\ g)$

$\mathbf{AG}f = \neg\mathbf{E}(\text{true}\ \mathbf{U}\ \neg f)$

$\mathbf{EG}f = \neg\mathbf{A}(\text{true}\ \mathbf{U}\ \neg f)$

Because all the operators are prefixed by **A** or **E**, the truth or falsehood of a formula depends only on the given state *s*, and not on the particular branch.

Clarke, Emerson and Sistla demonstrated that CTL is an efficient means for verifying finite-state systems. In their approach, a finite-state system is modeled as a labeled transition graph which can be viewed as a *finite Kripke structure* represented as a triple $M=(S, R, P)$, where

- S is a finite set of states,
- R is a total binary relation on states and represents possible transitions, and
- P is a mapping that assigns to each state the set of atomic propositions that are true in the state.

A path within this structure is naturally defined as an infinite sequence of states, with each adjacent pair related by R.

As its name suggests, CTL interprets temporal formulas over structures that resemble infinite computation trees. In the context defined above, given M and an initial state s_0, it considers the infinite computation tree rooted at s_0, generated by considering all possible nondeterministic transitions at every state. The truth of a CTL formula is defined inductively as follows:

- $(M, s_0) \models p$ iff $p \in P(s_0)$, where p is an atomic proposition
- $(M, s_0) \models \neg f$ iff $(M, s_0) \not\models f$
- $(M, s_0) \models f \wedge g$ iff $(M, s_0) \models f$ and $(M, s_0) \models g$
- $(M, s_0) \models \mathbf{AX}f$ iff for all states t such that $(s_0, t) \in R$, $(M, t) \models f$

- $(M, s_0) \models \mathbf{EX} f$ iff for some states t such that $(s_0, t) \in R$, $(M, t) \models f$
- $(M, s_0) \models \mathbf{A}(f \mathbf{U} g)$ iff for all paths $(s_0, s_1, s_2 \ldots)$, $\exists k \geq 0$ such that $(M, s_k) \models g$, and $\forall i, 0 \leq i < k$, $(M, s_i) \models f$
- $(M, s_0) \models \mathbf{E}(f \mathbf{U} g)$ iff for some paths $(s_0, s_1, s_2 \ldots)$, $\exists k \geq 0$ such that $(M, s_k) \models g$, and $\forall i, 0 \leq i < k$, $(M, s_i) \models f$

Clarke, Emerson and Sistla showed that there is an algorithm for determining whether a CTL formula f is true in state s of the Kripke structure $M = (S, R, P)$ which runs in time $O(\text{length}(f) \times (|S| + |R|))$ [13].

An important consideration in the modeling of concurrency is the notion of *fairness*. Each fairness condition specifies a set of states in the machine, and requires that in any acceptable behavior these states must be traversed infinitely often. Fairness constraints are used to restrict the behavior of the design. Among possible fairness constraints, the following are very common ones [22]:

- Unconditional fairness: an infinite sequence is impartial iff every process is executed infinitely often during the computation.
- Weak fairness: an infinite computation sequence is weakly fair iff every process enabled almost everywhere is executed infinitely often.
- Strong fairness: an infinite computation sequence is strongly fair iff every process enabled infinitely often is executed infinitely often.

Since fairness cannot be expressed in CTL, Clark et al. modified the semantics of CTL to introduce the notion of fairness [13]. The new logic, called CTL^F, has the same syntax as CTL, but the structure is now a 4-tuple (S, R, P, F). S, R, P have the same meaning as in CTL and F is a collection of predicates on S. Fair paths in this context are defined as those along which states occurring infinitely often satisfy each predicate that belongs to F.

Because CTL^F still can not express strong fairness, Emerson and Lei defined Fair CTL by extending the notion of fairness in CTL to consider fairness constraints that are Boolean combinations of **F***p* (infinitely often *p*, same as **GF***p*) and **G***p* (almost always *p*, same as **FG***p*) operators [24]. Combinations of these operators can express strong fairness as well as unconditional and weak fairness.

Clarke and Emerson further extended CTL to CTL^* [14]. CTL^* is sometimes referred to as full branching time logic. It combines both branching time and linear time operators; a path quantifier, either **A** or **E** can prefix an assertion composed of arbitrary combinations of the usual linear time operators **G**, **F**, **X**, and **U**. For example, **EF***p* is a basic modality of CTL; **E(F***p*/**Fq*) is a basic modality of CTL*.

LTL versus BTTL. In linear time logics, temporal operators are provided for describing events along a single future time line, although when a linear formula is used for program specification there is usually an implicit universal quantification over all possible futures. In contrast, in branching time logics the operators usually reflect the branching nature of time by allowing explicit quantification over possible futures. One argument presented by the supporters of branching time logic is that it offers the ability to reason about existential properties in addition to universal properties [33].

2.2.4 ω-automaton based Model Checking

Basically, ω-automata is the same as conventional automata that accept strings, except that the final states of the latter (signaling the end of an accepted string) are replaced by an acceptance condition on the set of states, visited infinitely often. It is useful for modeling non-terminating processes [41].

ω-automaton based model checking considers containment rather than equivalence between the languages representing the implementation and the specification (L(Imp) and L(Spec), respectively). In other words, it determines whether L(Imp) $\subseteq$ L(Spec), thereby verifying that every behavior of the implementation satisfies the property expressed by the specification. This allows easier handling of partial specifications as well as abstractions, thereby facilitating hierarchical verification across different levels of abstraction. Kurshan implemented this approach in COSPAN which is the verification engine in the commercial tool FormalCheck [42].

In his work, Kurshan defined modified versions of finite-state automata and finite-state machines that accept sequences, called L-automata and L-processes, to represent specifications and implementations, respectively. A specification is typically represented by a deterministic L-automaton T (called a "task" in Kurshan's terminology) and an implementation by a nondeterministic L-process. A verification is cast in term of testing for a language containment, i.e., testing if L(A) $\subseteq$ L (T).

One of the greatest strengths of Kurshan approach is its use of reductions both to control the complexity of state-space analysis and to provide a basis for hierarchical verification. Because most techniques based on state-space analysis suffer form the problem of state-explosion, i.e., an exponential increase in the number of states with an increasing number of components, complexity management becomes a critical issue in practice. This is especially so since formal verification is expected to work on large problems that are beyond the reach of traditional simulation methods. Devoting effort to develop an underlying semantics that supports reduction methods has potentially many advantages, as it has been demonstrated by the work with COSPAN. Also, a method of hierarchical verification that includes stepwise refinement of specifications allows much larger systems to be handled in practice.

Though the approach described above has a strong theoretical basis with due regard to complexity issues, its application in practice is sometimes limited by the fact that the burden of providing a reduction transformation lies with the user. In order to realize the full potential of the system, some means of using its reduction mechanism is required. It is not always obvious which transformation works best, though the automated facility to check its validity does help in exploring different options.

2.2.5 Symbolic Model Checking

One of the serious limitations of the model checking approach is its reliance on an explicit state-transition graph representation of the hardware system to be verified. Typically, the number of states in a global graph increases exponentially with the number of gates/processes/elements (parallel components) in the system, resulting in what is popularly called the state explosion problem. This restricts the application of direct state enumeration approaches to small circuits only. Several alternatives have been explored to alleviate this problem.

McMillan presented a method for model checking that reduces the state explosion problem by representing the Kripke model implicitly with a Boolean formula represented in computer memory using Bryant's ROBDDs [7]. This method is called symbolic model checking because symbolic variables are used to represent the components of the system state rather than numeric values. Using symbolic model checking, it is possible to verify automatically some regularly structured systems with literally astronomical numbers of states [47].

The symbolic model checking algorithm is implemented by a procedure *Check* that takes the CTL formula to be checked as its argument and returns an ROBDD that represents exactly those states of the system that satisfy the formula.

McMillan et. al. developed the Symbolic Model Verifier (SMV) to check finite state systems against specification in CTL [47, 48, 49]. McMillan and Schwalbe successfully applied SMV to the verification of the Encore Gigamax cache consistency protocol and found some critical design errors, thus demonstrating the effectiveness of symbolic model checking techniques for industrial applications.

2.2.6 Existing Model Checkers

Several model checking tools have been developed over the last 10 years. The well-known ones are as follows:

- SMV (Symbolic Model Verifier): a symbolic model checking system developed by McMillan at Carnegie-Mellon University [48, 49]. This system permits the automatic verification of programs written in a specialized language for describing concurrent finite state systems and protocols.

- VIS (Verification Interacting with Synthesis): an integrated tool for verification, simulation and synthesis of finite state systems, developed at University of California at Berkeley. It contains a Fair CTL Model Checker and a behavioral equivalence checker for sequential circuits, language emptiness check for Büchi automata and combinational verification [6].

- CVE: an industrial verification environment developed at Siemens. It supports model checking of designs described in VHDL or EDIF against specifications given in a temporal logic called CIL [60].

- FormalCheck: an ω-automata based model checker based on Cospan developed at Bell Labs Design Automation, Lucent Technologies. The

reduction algorithms and refinement methodologies embedded in FormalCheck make the tool applicable to industrial-size designs [75].

The advantage of model checking techniques is that they can be made completely automatic. The major obstacle for model checking to be widely used in an industrial design flow is the state explosion problem. The most promising approach to this problem is the application of Reduced Ordered Binary Decision Diagrams (ROBDDs) to the representation of state graphs [7]. In the next section, we introduce ROBDDs and variable ordering on them.

2.3 Reduced Ordered Binary Decision Graphs

Model checking verification systems have been considerably improved by the application of BDDs to the representation of Boolean functions. In this approach, the explicit construction of state graphs is avoided. Instead, the state graph is implicitly represented by means of Boolean functions from sets of states to sets of states (predicate transformers).

2.3.1 Binary Decision Diagrams

"BDD" stands for "Binary Decision Diagram". A BDD over a set of $X_n = \{x_1, ..., x_n\}$ of Boolean variables is a directed acyclic graph with one source and at most two sinks labeled by 0 and 1 [1]. Each non-sink (internal) node is labeled by a variable in X_n and has two outgoing edges, corresponding to where the variable evaluates to a 0 and to a 1, respectively. For a given assignment to the variables, the function value is evaluated by tracing a path from the root to the terminal. For a given input $m = (m_1, ..., m_n)$, the evaluation starts at the root and at an internal node with label x_i the outgoing edge with label m_i is chosen [7].

Although BDDs have been researched for about four decades, their widespread use occurred only after Bryant showed that under two restrictions such graphs are canonical and can be easily manipulated. The first restriction is that a total ordering of the variables is enforced in the graph. That is, if we consider variables to be ordered as $x_1 < x_2 < ... < x_n$, then every path from the root to a sink encounters the variables in that order. The second restriction is that the graph is reduced. A graph can be reduced by the repeated application of the following two rules until they are no longer applicable [7]. These rules are:

- Merging Rule: Two isomorphic subgraphs should be merged.
- Deletion Rule: A vertex whose two branches point to the same vertex should be deleted.

The resulting BDD is called a Reduced Ordered BDD (ROBDD). The important symbolic manipulation procedures introduced by Bryant are *apply* and *compose*; these techniques operate on two identically ordered ROBDDs. *apply* allows two ROBDDs to be combined under some Boolean operations, and *compose* allows the substitution of an ROBDD variable with a function.

2.3.2 Variable Ordering on ROBDD

The size of an ROBDD representing a Boolean function can be exponential in the number of primary inputs in the worst case. This problem is commonly referred to as the "memory explosion" problem. One solution is to find a good variable order to reduce the size of an ROBDD because the size of an ROBDD is strongly dependent on this order [37].

For example, suppose we wish to build a BDD for the function $(x_1 \oplus y_1) \vee (x_2 \oplus y_2)$. Figure 2.2 shows two BDDs for this function using two different variable

orders. In general, the choice of the variable order can make the difference between a linear size BDD and an exponential one.

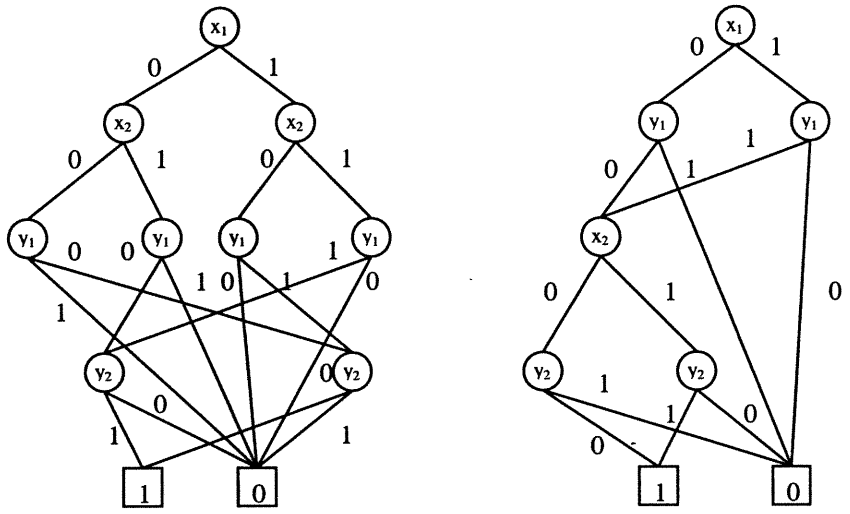


Figure 2.2 ROBDDs for the same function under two different variable orders

Much of the prior research in ROBDD has focused on finding good variable orders to reduce the size of an ROBDD representing a Boolean function. The methods for finding a good variable order can be classified into three categories:

- Static ordering: static methods based on information derived from the analysis of a multilevel logic implementation.
- Dynamic ordering: gradual improvement based on variable exchanges.
- Optimal ordering: exhaustive methods aimed at finding the best order.

The best published algorithms for the computation of an optimal variable order is a dynamic programming approach proposed in 1987 by Fridman and Supowit with a run time of $O(n^2 3^n)$ [26]. Bollig and Wegener have proved in 1994 that finding the best order is co-NP complete [3].

Since static and dynamic heuristic ordering methods can find a suitable order for most of realistic systems, it is not necessary to find the best order. Thus we only discuss static and dynamic methods here.

2.3.3 Static Variable Ordering

Because the choice of a good variable order is essential for the use of ROBDDs', many heuristics have been suggested. As any heuristic methods, they sometimes lead to a good variable order and sometimes they fail to compute a suitable order. The first and most successful static ordering algorithm was proposed by Fujita in 1988 [28]. His algorithm is based on two theorems:

Theorem 1 One of the best orders for a tree circuit (the number of fanouts of all inputs and gates is one) composed of only AND, OR and NOT gates is acquired by the following procedure: Traverse the gates from an output to the inputs in depth-first order (selection of input nets of a gate is arbitrary), and when an input is found, append that variable to the end of the current order.

Theorem 2 One of the best orders for a circuit which is composed of only AND, OR and NOT gates where only one input or gate has fanout of more than one (number of fanouts of all the other inputs and gates is one) is acquired by the following procedure which is slightly modified from that in Theorem 1: Traverse the gates from an output to the inputs in depth-first order, but a net which has fannout of more than one is selected first. When we find an input, append that variable to the end of the current order.

A heuristic ordering algorithm was developed from the two theorems. It gives a natural order by a depth-first traversal of a circuit (according to Theorem 1) and inputs which have fanout more than one are considered first (according to Theorem 2). The algorithm gives a good order for most of examples.

2.3.4 Dynamic Variable Ordering

Dynamic variable ordering was developed to allow ROBDD operation sequences that fail when using a fixed variable order to succeed when a new order is chosen mid-stream. This is done dynamically since ordering is performed by periodically applying a minimization algorithm which reorders the variables of the ROBDD to reduce its size. The sifting algorithm developed by Richard Rudel is one of the best reordering algorithms [57].

In the sifting algorithm, given an ROBDD G, a variable v is successively moved to each position in the order and the resulting graph size is examined. The variable is finally assigned the position which results in the smallest graph size. This process is known as sifting and is repeated for each variable in the graph.

The basic operation in reordering is the exchange of two adjacent variables (swap operation), which can be done in linear time [8]. Variable swapping involves moving all ROBDD nodes at level i to level $i + 1$ and nodes at level $i + 1$ to i. Figure 2.3 illustrates the procedure for swapping variables x_i and x_{i+1} in an ROBDD. Suppose in the original order, function f is indicated by a pointer to a node v in the ROBDD, where node v is labeled by variable x_i. After swapping, function f is indicated by a pointer to a node labeled by x_{i+1}; this node has branches to nodes labeled by x_i; these nodes in turn have branches to the subgraphs f_{00}, f_{10}, f_{01}, and f_{11}). Function f is still indicated by a pointer to node v, and other pointers to existing functions (shown as f_0 and f_1) remain undisturbed.

The swapping operation is completely local since only nodes of level i and $i + 1$ need to be traversed. Sifting n variables requires $O(n^2)$ swaps of adjacent variables, and each of these variable swaps has complexity proportional to the width of the ROBDD. Experimental results show that the sifting algorithm can produce ROBDDs 45% smaller than the original heuristic orders [57].

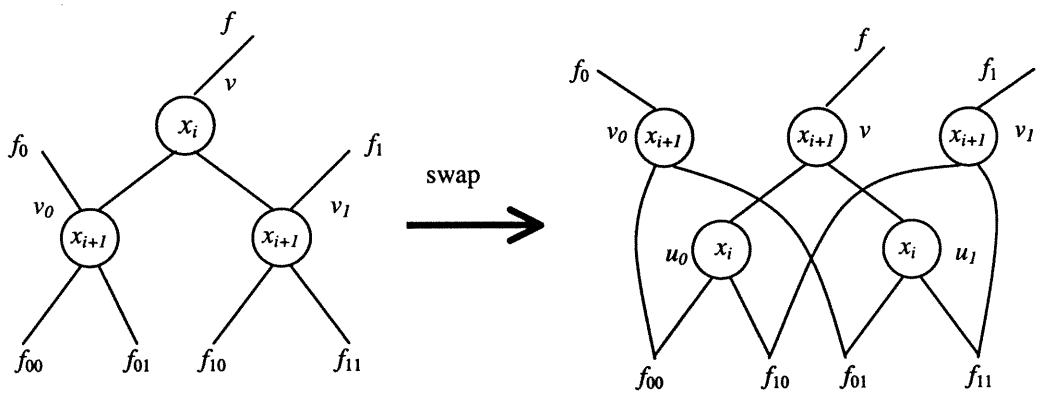


Figure 2.3 Variable swapping: node distribution before and after swapping

Since the time needed for sifting algorithm grows very fast with the number of variables, several improvements have been proposed. Panda presented in 1995 an extension of sifting that may sift groups of symmetry variables simultaneously to produce better results [55]. Meinel proposed in 1997 a *block-restricted sifting* to reduce the number of swap operations by restricting their application on variables within certain blocks [50].

Summary

In this chapter, we reviewed several modeling languages, including Verilog, VHDL, the SMV input language, Synchronous Verilog, and MDG-HDL. We then discussed model checking methods. The chapter concluded with the discussion of Reduced Ordered Binary Decision Diagram and two well-known variable ordering algorithms. In the next chapter, we introduce a new kind of decision graphs called MDG and MDG-based verification approaches.

Chapter 3 Multiway Decision Graphs

ROBDDs have proved to be a powerful tool for automated hardware verification. However, they require a Boolean representation of the circuit. Each individual bit of every data signal must be represented by a separate Boolean variable, while the size of an ROBDD grows, sometimes exponentially, with the number of Boolean variables. Therefore, ROBDD-based verification cannot be directly applied to circuits with complex datapaths.

The verification group at the University of Montreal has proposed a new class of decision graphs, called Multiway Decision Graphs (MDGs) that comprises, but is much broader than, the class of ROBDDs [11, 15]. With MDGs, a data signal can be represented by a single variable of abstract sort, rather than by a vector of Boolean variables, and a data operation can be viewed as a black box and represented by an uninterpreted function symbol. MDGs are thus more compact than ROBDDs for designs containing a datapath, and this greatly increases the range of circuits that can be verified.

This thesis explores two techniques to increase the scope of designs that can be verified using MDGs: variable ordering and automatic translation from VHDL to MDG-HDL. In this chapter, we review the MDG-related terminology that constitutes the theoretical background for the thesis. In Section 3.1, we describe the formal logic used by MDGs. In Section 3.2, we introduce the structure and several basic algorithms of MDGs. In Section 3.3, the abstract description of state machines is presented. This chapter is concluded with the introduction of MDG-based verification methods.

3.1 Formal Logic

3.1.1 Syntax

The formal logic underlying MDGs is a many-sorted first-order logic, augmented with the distinction between abstract sorts and concrete sorts [16]. This distinction is motivated by the natural division of datapath and control circuitry in RTL designs.

Concrete sorts have enumerations that are sets of individual constants, while abstract sorts do not. Variables of concrete sorts are used for representing control signals, and variables of abstract sorts are used for representing datapath signals. Data operations are represented by uninterpreted function symbols. An n-ary function symbol has a type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, where $\alpha_1 \ldots \alpha_{n+1}$ are sorts.

The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let f be a function symbol of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort then f is an abstract function symbol. If all the $\alpha_1 \ldots \alpha_{n+1}$ are concrete, f is a concrete function symbol. If α_{n+1} is concrete while at least one of $\alpha_1 \ldots \alpha_n$ is abstract, then we refer to f as a *cross-operator*. While abstract function symbols are used to denote data operations, cross-operators are useful for modeling feedback signals from the datapath to the control circuitry. Both abstract function symbols and cross-operators are *uninterpreted*, i.e., their intended interpretation is not specified.

The terms and their types (sorts) are defined inductively as follows: a constant or a variable of sort α is a term of type α, and if f is a function symbol of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, $n \geq 1$, and $A_1 \ldots A_n$ are terms of sorts $\alpha_1 \ldots \alpha_n$, then $f(A_1, \ldots, A_n)$ is a term of sort α_{n+1}. A term consisting of a single occurrence of an individual constant has multiple sorts (the sorts of the constant) but every other term has a

unique sort. The *top symbol* of a term is defined as follows: the top symbol of $f(A_1, \ldots, A_n)$ is f, and the top symbol of a term consisting of a single occurrence of a variable or a constant is that variable or constant.

We say that a term, variable or constant is concrete (resp. abstract) to indicate that it is of concrete (resp. abstract) sort. A term is *concretely reduced* iff it contains no concrete terms other than individual constants. Thus a concretely reduced term can contain abstract function symbols, abstract variables, abstract generic constants and individual constants, but it can contain no cross-operators, concrete function symbols, concrete generic constants, or concrete variables; and a concretely reduced term that is itself concrete must be an individual constant. A term of the form "$f(A_1, \ldots, A_n)$" where f is a cross-operator and $A_1, \ldots, A_n$ are concretely-reduced terms is called a *cross-term*. For example, if f is an abstract function symbol, c is an individual constant, x is a variable of concrete sort, and y is a variable of abstract sort, then $f(c, y)$ is a concretely-reduced term (assuming that it is well typed) while $f(x, y)$ is not.

A well-typed equation is an expression "$A_1 = A_2$" where the left-hand side (LHS) A_1 and the right-hand side (RHS) A_2 are terms of same type α. The *atomic formulas* are the equations, plus T (truth) and F (falsity). The *formulas* are defined inductively as follows: an atomic formula is a formula; if P and Q are formulas, then $\neg P$, $P \wedge Q$ and $P \vee Q$ are formulas; if P is a formula and x is a variable, then $(\exists x)P$ is a formula (with x bound in P). We use the abbreviation $P \Leftrightarrow Q$ for $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

3.1.2 Semantics

An *interpretation* is a mapping ψ that assigns a denotation to each sort, constant and function symbol, and satisfies the following conditions:

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

V being a set of variables, a *variable assignment* with domain V compatible with an interpretation ψ is a function ϕ that maps every variable $v \in V$ of sort α to an element $\phi(v)$ of $\psi(\alpha)$. We write Φ_V^{ψ} for the set of ψ-compatible assignments to the variables in V.

The denotation of a term under an interpretation ψ and a ψ-compatible variable assignment ϕ whose domain contains all the variables that occur in the term is defined by induction as follows: a constant c denotes $\psi(c)$; a variable x denotes $\phi(x)$; and if $A_1 \ldots A_n$ denote $v_1 \ldots v_n$, then $f(A_1, \ldots, A_n)$ denotes $(\phi(f))(v_1, \ldots, v_n)$. The truth of a formula P under an interpretation ψ and a ψ-compatible variable assignment ϕ whose domain contains the variable that occur free in P, written $\psi, \phi \models P$, is also defined by induction: $\psi, \phi \models A_1 = A_2$, iff A_1 and A_2 have same denotation; $\psi, \phi \models \neg P$ iff it is not the case that $\psi, \phi \models P$; $\psi, \phi \models P \wedge Q$ iff $\psi, \phi \models P$ and $\psi, \phi \models Q$; $\psi, \phi \models P \vee Q$ iff $\psi, \phi \models P$ or $\psi, \phi \models Q$; and $\psi, \phi \models (\exists x)P$ iff $\psi, \phi' \models P$ for some ϕ' that assigns an arbitrary value to x and otherwise coincides with ϕ.

We write $\psi \models P$ when $\psi, \phi \models P$ for every ψ-compatible assignment ϕ to the variables that occur free in P, and $\models$ P when $\psi \models P$ for all ψ. Two formulas P and Q are logically equivalent iff $\models P \Leftrightarrow Q$. A formula P *logically implies* a formula Q iff $\models P \Rightarrow Q$.

3.1.3 Directed Formulas

Given two disjoint sets of variables U and V, a directed formula of type $U \rightarrow V$ is a formula in disjunctive normal form (DNF) such that

1. Each disjunct is a conjunction of equations of the form

   $A = a$, where A is a term of concrete sort α of the form "$f(B_1, \ldots, B_n)$" that contains no variables other than elements of U, and a is an individual constant in the enumeration of α, or

   $u = a$, where $u \in U$ is a variable of concrete sort α and a is an individual constant in the enumeration of α, or

   $v = a$, where $v \in V$ is a variable of concrete sort α and a is an individual constant in the enumeration of α, or

   $v = A$, where $v \in V$ is a variable of abstract sort α and A is a term of type α containing no variables other than elements of U;
2. In each disjunct, the LHSs of the equations are pairwise distinct; and
3. Every abstract variable $v \in V$ appears as the LHS of an equation $v = A$ in each of the disjuncts. (Note that there need not be an equation $v = a$ for every concrete variable $v \in V$).

Intuitively, in a DF of type $U \rightarrow V$, the U variables play the role of independent variables, the V variables play the role of dependent variables, and the disjuncts enumerate possible cases. In each disjunct, the equations of the form $u = a$ and $A = a$ specify a case in terms of the U variables, while the other equations specify the values of (some of the) V variables.

A DF is said to be *concretely reduced* iff every A in an equation $A = a$ is a cross-term, and every A in an equation $v = A$ is a concretely reduced term. It is easy to see that every DF is logically equivalent to a concretely reduced DF, given complete specifications of the concrete function symbols and concrete generic constants; the reduction can be accomplished by case splitting.

In the next section, we introduce the graphical representation of a Directed Formula – the Multiway Decision Graph (MDG). We first review the structure of MDGs, then discuss several basic MDG algorithms

3.2 Multiway Decision Graphs

3.2.1 Structure

Definition 3.1 A multiway decision graph (MDG) is a finite directed acyclic graph G where the leaf nodes are labeled by formulas, the internal nodes are labeled by terms, and the edges issuing from an internal node N are labeled by terms of the same sort as the label of N. Such a graph represents a formula defined inductively as follows: (i) if G consists of a single leaf node labeled by a formula P, then G represents P; (ii) if G has a root node labeled A with edges labeled $B_1 \ldots B_n$ leading to subgraphs $G_1' \ldots G_n'$, and if each G_i' represents a formula P_i, then G represents the formula $\bigvee_{1 \le i \le n}((A = B_i) \wedge P_i)$.

Six well-formedness conditions are provided to turn MDG into a canonical representation that can be manipulated by efficient algorithms [16]:

1. *Kinds of nodes.* An internal node must be labeled by a variable of abstract sort, with edges issuing from the node labeled by concretely-reduced terms of that same sort; or by a variable of concrete sort, with edges labeled by individual constants in the enumeration of that sort; or by a cross-term, with edges labeled by individual constants in the enumeration of the sort of the cross-term. A leaf node must be labeled by T (for True), except in the case where the graph has only one node labeled F (for False).

Note that the conditions about concretely-reduced terms and cross-terms are only syntactical restrictions, since it is possible to meet these restrictions using case splitting.

We refer to an occurrence of a variable in a term that labels an edge or in a cross-term that labels a node as a *secondary occurrence*, while an occurrence of a variable as the label of a node is a *primary occurrence*. Neither the edge labels, which are concretely-reduced, nor the cross-terms, contain concrete variables. Hence only abstract variables can have secondary occurrences. The *primary variables* (resp. *secondary variables*) of a graph G are those that have primary (resp. secondary) occurrences in G.

2. *Ordering*. The labels of the edges issuing from a given node must appear in a *standard term order*, without repetitions. Along each path, the variables and the cross-operators of the cross-terms that label the nodes must appear in a *custom symbol order*, and cross-terms with the same cross-operator must appear in the standard term order; there must be no repeated labels.

The custom symbol order is a generalization (to include cross-operators) of the variable ordering used for ROBDDS, and plays the same role. It need only involve the cross-operators and those variables that may appear as node labels. It is chosen carefully for each particular application so as to keep the MDGs of manageable size if possible. The standard term ordering, on the other hand, is chosen arbitrarily once and for all; it need not be compatible with the custom symbol order. From these two orderings we define the *node-label ordering* among the variables and cross-terms as follows: A comes before B iff the top symbol of A comes before the top symbol of B in the custom order, or A and B are cross-terms with the same cross-operator and A comes before B in standard term order. Condition 2 states that node labels must appear in node-label order along each path.

3. *Minimality*. There must be no distinct isomorphic subgraphs, and no redundant nodes.

In an MDG, a redundant node is a node labeled by a concrete variable or a cross-term of sort α, with edges labeled by all the individual constants in the enumeration of α, all leading to the same subgraph.

4. No variables should have both primary and secondary occurrences in the same graph.

5. The set of abstract variables having primary occurrences along a path is the same for all paths in a given graph.

6. If a node N is labeled by an abstract variable x, and an abstract variable y participating in the custom symbol order occurs in a term A that labels one of the edges that issue from N, then y must come before x in the custom symbol order. Similarly, if N is labeled by a cross-term A with cross-operator f, and y is an abstract variable that occurs in A, then y must come before f in the custom symbol order.

An MDG is said to be *well-formed* if and only if it satisfies conditions 1 through 6 above. From now on, MDG will mean well-formed MDG unless stated otherwise.

Given an MDG G, if U is the set of variables having secondary occurrences in G, and V is the set of variables having primary occurrences, then G is of type $U \rightarrow V$. When we say that an MDG is of type $U \rightarrow V$, it will always be understood that U and V are disjoint sets of variables, and that all the abstract variables in V participate in the custom symbol order.

An MDG is a graphical representation of a Directed Formula as defined above. Given a concretely reduced DF P of type $U \rightarrow V$, a standard term order, and a custom symbol order comprising all the variables in V and all the cross-operators in P, it is easy to construct an MDG representing a DF that coincides with P.

3.2.2 Basic Algorithms

The following basic MDG algorithms were implemented in the past [11]. To simplify the description of the algorithms we identify an MDG with the directed formula (DF) that it represents.

Disjunction: The disjunction algorithm is n-ary. It takes as inputs a set of DFs P_i, $1 \leq i \leq n$, of types $U_i \rightarrow V$, and produces a DF $R = \textbf{\textit{Disj}}(\{P_i\}_{1 \leq i \leq n})$ of type $(\underset{1 \leq i \leq n}{\cup} U_i) \rightarrow V$ such that

$$\models R \Leftrightarrow (\underset{1 \leq i \leq n}{\vee} P_i).$$

Note that this algorithm requires that all the P_i, $1 \leq i \leq n$, have the same set of abstract primary variables. If two DFs P_1, P_2 do not have the same set of abstract primary variables, then there is no DF R such that $\models R \Leftrightarrow (P_1 \vee P_2)$.

Conjunction: The conjunction algorithm takes as inputs a set of DFs P_i, $1 \leq i \leq n$, of types $U_i \rightarrow V_i$ and produces a DF $R = \text{Conj}(\{P_i\}_{1 \leq i \leq n})$ of type

$$((\underset{1 \leq i \leq n}{\cup} U_i) \setminus (\underset{1 \leq i \leq n}{\cup} V_i)) \rightarrow (\underset{1 \leq i \leq n}{\cup} V_i)$$

such that $\models R \Leftrightarrow (\underset{1 \leq i \leq n}{\wedge} P_i)$. Note that for $1 \leq i < j \leq n$, V_i and V_j must not have any abstract primary variables in common, otherwise the conjunction cannot be computed.

Relational product: The algorithm takes as inputs a set of DFs P_i, $1 \le i \le n$, of types $U_i \rightarrow V_i$, a set of variables E to be existentially quantified, and a renaming substitution η, and produces a DF $R = \mathbf{RelP}(\{P_i\}_{1 \le i \le n}, E, \eta)$ such that

$$\models R \Leftrightarrow ((\,(\exists E)\,(\underset{1 \le i \le n}{\wedge} P_i)) \cdot \eta\,).$$

The algorithm computes the conjunction of the P_i, existentially quantifies the variables in E, and applies the renaming substitution η. For $1 \le i < j \le n$, V_i and V_j must not have any primary abstract variables in common. The type of the result R is

$$((\underset{1 \le i \le n}{\cup} U_i) \setminus (\underset{1 \le i \le n}{\cup} V_i)) \rightarrow (((\underset{1 \le i \le n}{\cup} V_i) \setminus E) \cdot \eta).$$

Pruning by subsumption: The algorithm takes as inputs two DFs P and Q of types $U \rightarrow V_1$ and $U \rightarrow V_2$ respectively, and produces a DF $R = \mathrm{PbyS}(P,Q)$ of type $U \rightarrow V_1$ derivable from P by *pruning* (i.e., by removing some of the disjuncts) such that

$$\models R \vee (\exists U)\, Q \Leftrightarrow P \vee (\exists U)\, Q \qquad (3.1)$$

The disjuncts that are removed from P are *subsumed* by Q, hence the name of the algorithm.

Since R is derivable from P by pruning, after the formulas represented by R and P have been converted to DF, the disjuncts in the DF of R are a subset of those in the DF of P. Hence $\models R \Rightarrow P$. And, from (3.1), it follows tautologically that $\models P \wedge \neg(\exists U)Q \Rightarrow R$. Thus we have

$$\models (\,P \wedge \neg(\exists U)Q \Rightarrow R\,) \wedge (\,R \Rightarrow P\,).$$

We can then view R as approximating the logical difference of P and $(\exists U)Q$. In general, there is no DF logically equivalent to $P \wedge \neg(\exists U)Q$. If R is $\mathbf{F}$, then it follows tautologically from (3.1) that $\models P \Rightarrow (\exists U)Q$.

Those basic algorithms are the building blocks of the procedures for MDG-based verification. In MDG-based verification, abstract descriptions of state machines (ASM) are used to model the systems. In the next section, we introduce abstract state machines and the related state explosion.

3.3 Abstract State Machines

An Abstract State Machine (ASM) is a model used for describing hardware designs at the Register Transfer Level [11]. It is a state machine given by an abstract description in terms of MDGs (or equivalent Directed Formulas).

3.3.1 Representing Sets using MDGs

Let P be an MDG of type $U \rightarrow V$. Then, for a given interpretation ψ, P can be used to represent the set of vectors

$$Set_V^{\psi}(P) = \{ \phi \in \Phi_V^{\psi} \mid \psi, \phi \models (\exists U)\, P \}$$

In the next section, MDGs will thus be used in this fashion to represent sets of states and sets of output vectors. We shall also see how MDGs can be used to represent relations.

3.3.2 Describing State Machines with MDGs

An abstract description of a state machine M is a tuple $D = (X, Y, Z, F_I, F_T, F_O)$, where

X, Y, Z are disjoint sets of variables, viz. the input, state, and output variables, respectively. Let η be a one-to-one function that maps each variable y to a distinct variable $\eta(y)$ obtained, for example, by adorning y with a prime. The

variables in $Y' = \eta(Y)$ are used as the next-state variables. X, Y and Z must be disjoint from Y'.

Given an interpretation ψ, an input vector of the state machine M represented by D is a ψ–compatible assignment to the set of input variables X; thus the set of input vectors, or inputs alphabet, is Φ_X^ψ. Similarly, Φ_Z^ψ is the output alphabet. A state is a ψ-compatible assignment to the set of state variables Y; hence the state space is Φ_Y^ψ. A state ϕ can also described by an assignment $\phi' = \phi \circ \eta^{-1} \in \Phi_{Y'}^\psi$, to the next state variables.

F_I is an MDG representing the set of initial states, of type $U \to Y$, where U is a set of abstract variables disjoint from $X \cup Y \cup Y' \cup Z$. Typically, F_I is a one-path MDG where each internal node N is labeled by a variable $y \in Y$, and the edge that issues from N is labeled by the symbolic initial value of y, which can be an individual constant, an abstract generic constant, or an abstract variable $u \in U$. It is possible to specify that two data registers have the same value, but that this common value is arbitrary, by using the same u as symbolic initial value of the abstract state variables representing the two registers.

Given an interpretation ψ, a state $\phi \in \Phi_Y^\psi$ is an initial state iff $\psi, \phi \models (\exists U) F_I$. Thus the set of initial states of the state machine M represented by D is

$$S_I = \{\ \phi \in \Phi_Y^\psi \mid \psi,\ \phi \models (\exists U)\ F_I\ \} = Set_Y^\psi\ (F_I)$$

F_T is an MDG of type $(X \cup Y) \to Y'$ representing the transition relation. Given an interpretation ψ, an input vector $\phi \in \Phi_X^\psi$ and a state $\phi' \in \Phi_Y^\psi$, a state $\phi'' \in \Phi_Y^\psi$ is a possible next state iff $\psi, \phi \cup \phi' \cup \phi'' \circ \eta^{-1} \models F_T$. Thus the transition relation of the state machine M represented by D is

$$R_T = \{\ (\phi, \phi', \phi'') \in \Phi_X^\psi \times \Phi_Y^\psi \times \Phi_Z^\psi \mid \psi, \phi \cup \phi' \cup (\phi'' \circ \eta^{-1}) \models F_T\ \}$$

F_O is an MDG of type $(X \cup Y) \to Z$ representing the output relation. Given an interpretation ψ, the output relation of the state machine M represented by D is

$$R_O = \{\ (\phi, \phi', \phi'') \in \Phi_X^\psi \times \Phi_Y^\psi \times \Phi_Z^\psi \mid \psi, \phi \cup \phi' \cup \phi''\ \models F_O\ \}$$

To recapitulate, for every interpretation ψ of the sorts, constants and function symbols of the logic, the abstract description $D = (X, Y, Z, F_I, F_T, F_O)$ represents the state machine $M = (\Phi_X^\psi, \Phi_Y^\psi, \Phi_Z^\psi, S_I, R_T, R_O)$ with the input alphabet Φ_X^ψ, the state space Φ_Y^ψ, the output alphabet Φ_Z^ψ, the set of initial state S_I, the transition relation R_T, and the output relation R_O.

3.3.3 State Exploration

Given an abstract state machine description D = (X, Y, Z, F_I, F_T, F_O), we can compute the set of the reachable states of a state machine $M = (\Phi_X^\psi, \Phi_Y^\psi, \Phi_Z^\psi, S_I, R_T, R_O)$ represented by D, for any interpretation ψ, using the MDG algorithms mentioned above. At the same time we can check that a given condition on the outputs of the machine, the *invariant*, holds in all the reachable states. The invariant is represented by an MDG C of type $W \rightarrow Z$, where W is a set of abstract variables disjoint from X, Y, Y', Z and U. For a given interpretation ψ, an output vector is deemed to satisfy the invariant iff ψ, $\phi \models (\exists W)C$; therefore, $Set_Z^\psi(C)$ is the set output vectors that satisfy the invariant.

The invariant checking algorithm based on Reachability Analysis, can be described by the following pseudo-code:

```
1.    ReAn( D, C )
2.          R := F_I ; Q := F_I ; K := 0 ;
3.          loop
4.                K := K +1;
5.                I := Fresh(X, K);
6.                O := RelP({ I, Q, F_O }, X ∪ Y, ∅);
7.                P := PbyS( O, C );
```

8. if P ≠ F then return failure;
9. N := RelP($\{I, Q, F_T\}, X \cup Y, \eta$);
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;

Variables I, N, P, Q represent sets of states, and O represents a set of output vectors. Before each iteration, R contains the states reached so far, while Q is the frontier set, i.e., a subset of $Set_Y^{\Psi}(R)$ containing at least all those states that entered $Set_Y^{\Psi}(R)$ for the first time in the previous iteration.

In line 5, Fresh(X, K) constructs a one–path MDG representing a conjunction of equations $x = u$, one for each abstract input variable $x \in X$, where u is a fresh variable from the set of auxiliary abstract variables U. The value of the loop counter K is used to generate the fresh variables. This one-path MDG is assigned to I, which represents the set of input vectors.

In line 6, the relational product operation is used to compute the MDG representing the set of output vectors produced by the states in the frontier set. The resulting MDG is assigned to O. Then, in line 7, the pruning-by-subsumption operation is used to remove from O paths representing output vectors that satisfy the invariant C. The resulting MDG is assigned to P. In line 8, if P is not F, then the procedure stops and reports failure. If P is F, then $Set_Z^{\Psi}(O) \subseteq Set_Z^{\Psi}(C)$, i.e., every output vector produced by a state in the frontier set satisfies the invariant, and the verification procedure continues.

In line 9, the relational product operation is used again, this time to compute the MDG representing the set of states that can be reached in one step from the frontier set.

Lines 10 and 11 check whether $Set_Y^{\Psi}(N) \subseteq Set_Y^{\Psi}(R)$, by the same method as in lines 7 and 8 to check whether $Set_Z^{\Psi}(O) \subseteq Set_Z^{\Psi}(C)$. If this is indeed the case, then every state reachable from the frontier set was already in $Set_Y^{\Psi}(R)$. The fixpoint has been reached and R represents all the reachable states. Therefore, the procedure terminates and reports success. Otherwise the MDG assigned to Q in line 10 represents the new frontier set.

Line 12 simplifies R by removing from it any paths that are subsumed by Q, using PbyS. There may be such paths because Q was not computed earlier as an exact difference. Then line 13 computes the new value of R by taking the disjunction of R and Q, which represents the set of states $Set_Y^{\Psi}(R) \cup Set_Y^{\Psi}(Q)$, and assigns it to R.

3.4 MDG-based Verification Applications

The MDG-based method makes it possible to verify circuits automatically at the RT level, using abstract types and uninterpreted function symbols. The MDG package contains the following verification applications [66, 69]:

Combinational equivalence checking: Given two combinational circuits, we compute for each of them an MDG representing its input-output relation by combining the MDGs of the components of the circuit using the relational product operation. Because of the canonicity of MDGs, comparing the functionality of two combinational circuits reduces to computing the MDGs representing their input/output relations. If the two circuits have the same functionality, the two

MDGs must represent logically equivalent formulas, and hence they must be isomorphic.

Safety property checking: The safety property checking is based on the reachability analysis procedure. Given a state machine M and an invariant condition C, we check if C holds in all the reachable states of M. An invariant condition is specified by a combinational circuit whose output signals are named by the variables that occur in the condition. Pruning-by-subsumption is used to check that the invariant is satisfied for the states in each frontier set.

Sequential equivalence checking: One application of the safety property checking is the behavioral equivalence checking of two sequential circuits. To verify that two machines produce the same sequence of outputs for every sequence of inputs, we feed the same inputs to the two circuits, i.e., we form the product state machine. Then, reachability analysis is performed on this parallel composition using an invariant that asserts the equality of the corresponding outputs in all the reachable states. For machines at different time scales, it is possible to synchronize them first if they have cyclic behavior. Then we can perform reachability analysis on the product machine as usual.

Model checking: A model checking algorithm for a subset of Abstract-CTL* called L_{MDG} was developed by Xu [68]. It can verify safety and liveness properties with or without fairness constraints. To check a property p in L_{MDG} on an ASM M, we first build additional ASMs M_j for basic sub-formulas of p in which only the temporal operator **X** is allowed (called *Next_let_formulas*), and then we compose these additional ASMs with M. Finally, we apply the appropriate algorithm developed by Xu to verify a simplified property on the composite machine.

In all the above applications, MDGs require an initial variable order before any MDG manipulation. The variable order decides not only the size of the MDGs but also the memory requirement and the execution time needed for a verification

application. For sequential verifications (safety property checking, sequential equivalence checking and model checking), a dynamic reordering method may be needed to minimize the size of MDGs during the verification process.

Summary

MDG is a new type of a decision graph which incorporates abstract sorts and uninterpreted function symbols. MDGs can represent RTL designs at an abstract level that is independent of the datapath width. A new verification approach based on MDG was summarized. In this approach, abstract state machines are used to model systems.

MDG-based verification alleviates the state explosion at the Boolean level and thus greatly increases the scope of designs that can be verified. However, the presence of abstract variables and cross-operators in the structure of MDGs makes the variable ordering problem more difficult than in the ROBDD case.

In the following chapters, we concentrate on developing two techniques that improve the efficiency of the MDG system: automatic variable ordering methods on MDG in Chapter 4, 5 and 6, and automatic translation from VHDL to MDG-HDL in Chapter 7.

Chapter 4 Static Variable Ordering on MDG

MDGs are a canonical representation of circuits at a higher abstraction level provided that the order of variables is fixed like in ROBDD. The MDG for a given circuit can have many different forms depending on the variable order, and the size of MDGs may greatly vary with the order. The size of an MDG determines not only the memory requirements but also the amount of execution time for its manipulation. The variable ordering algorithm is thus one of the most important issues in the application of MDGs.

This chapter presents static variable ordering algorithms that give an initial order to generate MDGs. This order is *static*, in the sense that it is chosen *a priori* (before any MDG manipulation) and uses information about the circuit topology. We first define 3 constraints on variable ordering in Section 4.1. We then discuss several heuristic rules for variable ordering in Section 4.2. In Section 4.3, we present static variable ordering algorithms for combinational and sequential circuits. The proof of convergence of the algorithms is given in Section 4.4. Section 4.5 reports experimental results on IFIP benchmark circuits, an Island Tunnel Controller and the Fairisle 4×4 ATM switch fabric.

4.1 Constraints on Variable Ordering on MDG

For an MDG, a total order "<" over the set of variables is imposed: on any path from a root to a terminal node the variables occur in the given order. In this thesis, we use $a < b$ to denote that a comes before b in the order and we also say a has a higher position than b in an MDG. An ordering list of variables is denoted as $x_0 < x_1 \ \ldots < x_n$. x_0 is at the beginning of the list and is the variable labeling the root

node in most of cases. For example, in Figure 4.1, the order is $x_1 < x_2 < y$. The head of the order is x_1. It is the concrete variable labeling the root node and has a higher position than x_2 and y.

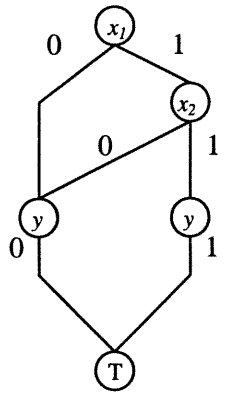


Figure 4.1 An example of MDG

Compared to ROBDD, variable ordering on MDG is complicated by the presence of first order terms. To keep MDG well formed, the following constraints on the order of abstract variables and cross-operators must be respected [70]:

1. If an abstract variable a appears as a secondary variable in an edge label of node b, then $a < b$.
2. If a variable a appears as a secondary variable in a cross-term having cross-operator f, then $a < f$.
3. The present and next state variables must be in a corresponding order. If the present state variables are in the order $a < b < c$, then the corresponding next state variables should be in the order $a' < b' < c'$.

Figure 4.2 shows an MDG representing a circuit which is assigned the smaller value of the two inputs x_1, x_2 (an abstract sort). The inputs are compared using an uninterpreted cross-operator *leq* such that for any two values a and b of sort s, $leq(a, b) = 1$ if and only if a is less than or equal to b.

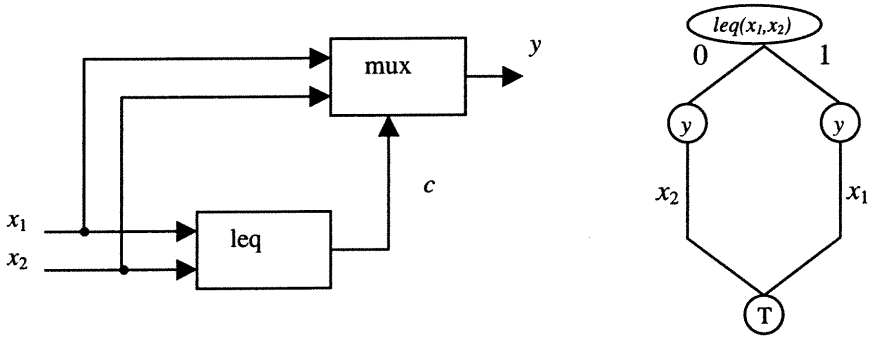


Figure 4.2 An MDG representation of the circuit that selects a smaller value

x_1 and x_2 are secondary variables and they label the edge of y. According to Constraint 1, x_1 and x_2 must come before y. Since they also appear in the cross-term $leq(x_1, x_2)$, they must come before the cross-operator *leq* according to Constraint 2. Thus there are 4 constraints in this MDG: $x_1 < y$, $x_2 < y$, $x_1 < leq$ and $x_2 < leq$.

In the process of ordering, all these constraints have to be observed. In Section 4.3, we present a constraint checking/adjusting procedure in the static ordering algorithm. Before discussing the algorithm, we introduce next several heuristic rules for variable ordering on MDG.

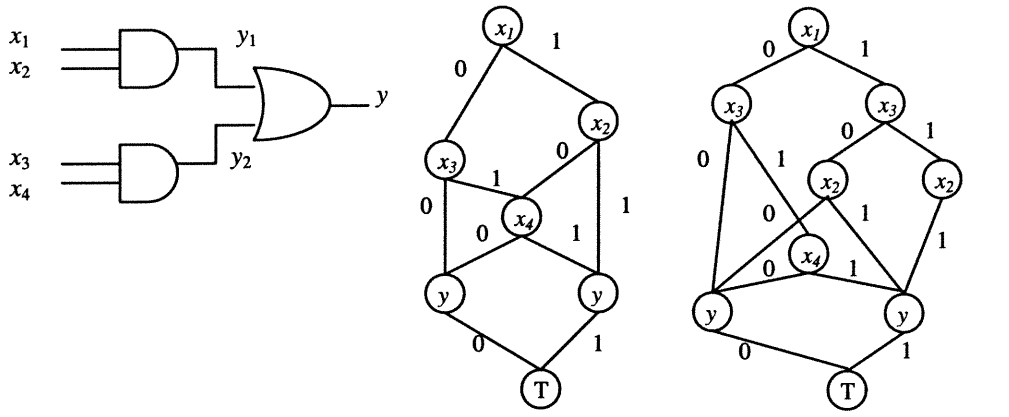
4.2 Heuristic Rules for Variable Ordering

Since MDGs subsume ROBDDs, there are two variable ordering rules of ROBDDs that can be imported for MDGs that have concrete variables [29, 30].

1. The variables whose connections are topologically close to each other in the circuit should be near each other in the variable order.

Variables appearing in the same sub-circuit are close. Consider, for example, the circuit shown in Figure 4.3(a). x_1 and x_2 are close because they appear in an AND operation and directly decide the value of y_1. x_1 and x_4 are less close than x_1 and x_2 since they only decide the value of y indirectly. The circuit's MDG has 6 nodes under the ordering $x_1 < x_2 < x_3 < x_4 < y$ (Figure 4.3(b)), but has 8 nodes under the ordering $x_1 < x_4 < x_2 < x_3 < y$ (Figure 4.3(c)).

For a circuit of the form $(x_1 \wedge x_2) \vee \ \ldots \vee (x_{2n-1} \wedge x_{2n})$, an MDG requires $2n + 2$ nodes under the order $x_1 < x_2 < \ldots < x_{2n-1} < x_{2n}$, while it requires 2^{n+1} nodes under the order $x_1 < x_{n+1} < \ldots < x_n < x_{2n}$.



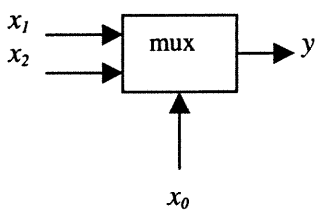
(a) circuit (b) $x_1 < x_2 < x_3 < x_4 < y$ (c) $x_1 < x_4 < x_2 < x_3 < y$

Figure 4.3 MDGs for a 2-level AND-OR circuit

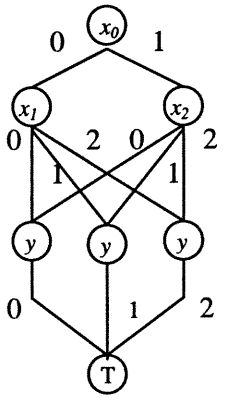
2. The control variables of a circuit should be located at higher positions in the order than other related data variables.

Consider a two-way multiplexer as shown in Figure 4.4(a). x_1, x_2 and y are of a concrete sort with enumeration $\{0, 1, 2\}$. y is the control variable. The MDGs

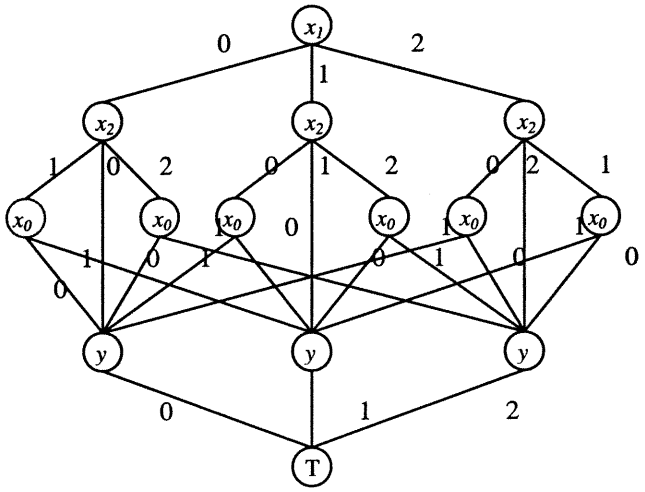
under the order $x_0 < x_1 < x_2 < y$ and $x_1 < x_2 < x_0 < y$ are shown in Figure 4.4(b) and (c). If x_1, x_2 and y are of a concrete sort with enumeration $\{c_i\}_{1 \le i \le m}$, it would take m+3 nodes under the first order, but take $m^2 + m + 1$ nodes under the second order. However, in this example, if x_1, x_2 and y were of an abstract sort and x_0 of a concrete sort, the MDGs would be the same under both orders since x_1 and x_2 only label edges and must precede y in the order (Figure 4.4(d)).



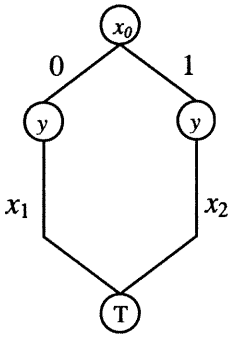
(a) Circuit



(b) $x_0 < x_1 < x_2 < y$



(c) $x_1 < x_2 < x_0 < y$



(d) x_1, x_2 and y are abstract variables

Figure 4.4 MDGs for 3-sort multiplexer

An extreme example of this rule is to represent an n-bit data selector with Boolean variables. When the control inputs are high in the order, the MDG size is linear, whereas it becomes exponential using the reverse order.

Based on our experience with MDGs, we derived several additional rules for variable ordering in MDGs.

3. For any component, the output should be located after all the inputs.

Consider a simple 3–input AND gate ($y = x_1 \wedge x_2 \wedge x_3$) and its MDG representation for two different orders of the variables in Figure 4.5(a). We can get the best order by ordering the variables in the order $x_1 < x_2 < x_3 < y$ (Figure 4.5(b)). We can have a better understanding of how this arises when we compare the two graphs in Figure 4.5 (b) and (c). In (b), whenever $x_i = 0$ and y is 0, we do not need to consider the inputs after x_i. However, in the gray part of (c), we have to have more nodes to hold the values when x_i is 1 and x_{i+1} is 0. For an n-bit AND gate or any other Boolean gate, putting the output after all inputs yields a better order. The resulting graph has n-1 nodes less than the other possible graphs.

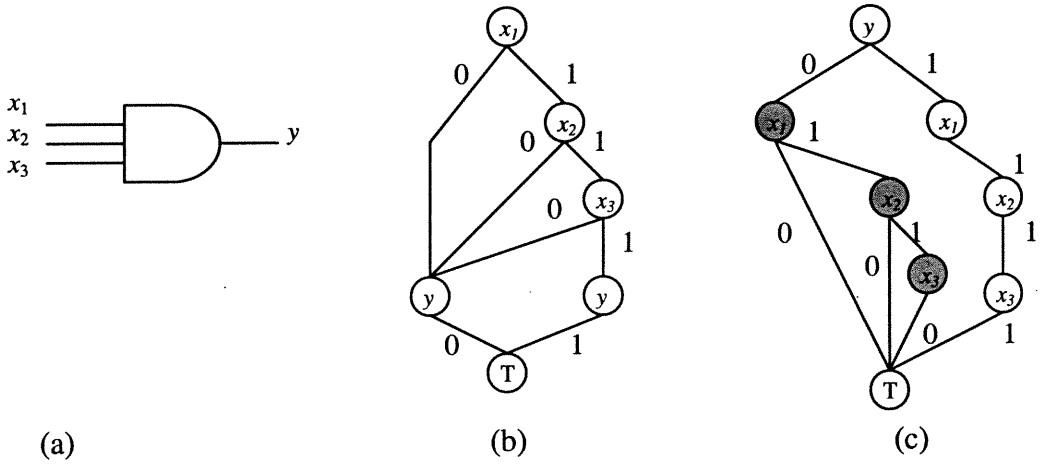


Figure 4.5 MDGs for 3-input AND gate

4. It is usually more efficient to put a concrete variable whose sort has a large enumeration before other variables.

This is because the graphs can be restricted dramatically as early as possible when doing a conjunction. In the abstract implicit state enumeration, conjunction is an important part for the relational product operation. Conjunction is recursively computed from two MDGs. If the MDGs contain some common concrete variables whose sorts have a large enumeration, conjoining them early in the recursive process saves both memory and time.

5. An abstract variable may have a large number of distinct terms that can be assigned to itself. For example, an output of a data operation may have many different results according to different conditions. If such variables could be identified, it is usually better to put them at the beginning of the order list.

This rule can be demonstrated by a simple example shown in Figure 4.6. The circuit and its output tables are shown in (a) and (b), respectively. The input variables x_1, x_2, x_3, x_4 and the outputs y_1, y_2 are of an abstract sort, while the control variable x_0 is of a concrete sort with the enumeration {0, 1, 2, 3}. Depending on the value of x_0, the ALU can add, subtract, increment, or produce *zero*. The operations are represented by symbols *add*, *sub* and *inc*. The symbol *zero* is a generic constant. The output of the multiplexer is x_3 or x_4 depending on whether x_0 is 0. The abstract variable y_1 has 4 distinct terms that can be assigned to itself while y_2 only has 2. When we put y_1 before y_2, the corresponding MDG shown in (c) is more compact than the MDG using the reverse order as shown in (d).

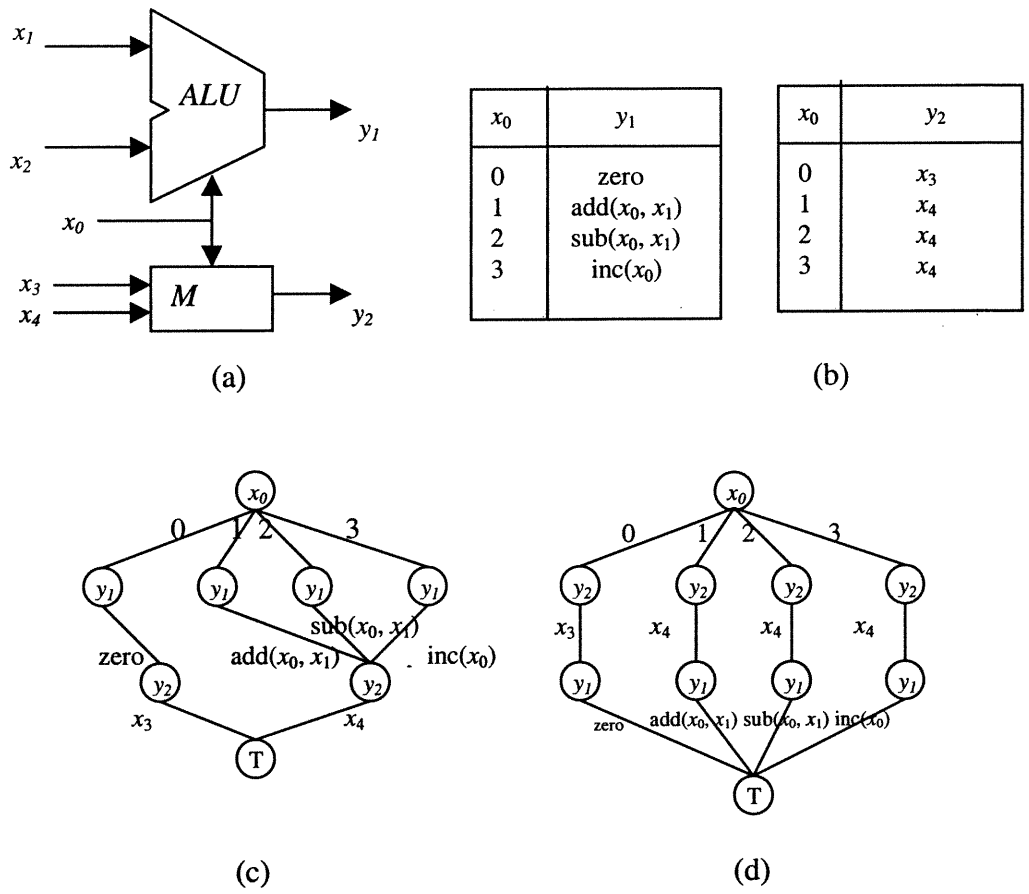


Figure 4.6 MDGs for an ALU and a multiplexer

If we could find a variable order that satisfies all those rules, the MDGs would be rather compact. However, a practical circuit consists of many functions, which may require different and conflicting orders. It is difficult to find a point of compromise to satisfy all the rules. From our experience with MDGs, we have found that rules 4 and 5 are more important for sequential circuits than for combinational circuits. Rules 1, 2 and 3 are important for both kinds of circuits and their priorities can be ordered as "rule 1 < rule 2 < rule 3". Our static algorithm was developed under this metarule.

4.3 A Static Variable Ordering Algorithm for MDG

It is hard to find the best order for practical circuits due to complex signal relations inside the circuit. This section presents a static variable ordering algorithm based on circuit topology and the heuristic rules discussed in Section 4.2. Before presenting the algorithms, we give two definitions needed to introduce the algorithm.

Let $Z = (z_1, \ldots z_n)$ be the output variables and $X = (x_1, \ldots x_n)$ be the primary input variables. Let fi be the output function, then $z_i = fi\ (X)$.

Definition 4.1. Let $ddv(z_i)$ be the set of direct determining variables of z_i. It includes all variables $v \in Z \cup X$ such that $fi \mid v=a \neq fi \mid v=b$ for some $a \neq b$, where $fi \mid v=x$ is the cofactor of fi for $v=x$. Let $dv(z_i)$ denote the set of determining variables of z_i, which is defined recursively as follows: $dv(z_i) = ddv(z_i) \cup \{\ dv(z_j) \mid z_j \in ddv(z_i) \backslash z_j\}$.

Definition 4.2. The depth of an output is the longest path (measured in the number of components) from this output to the primary inputs.

For example, in Figure 4.7, the depth of $o1$ is 5 and the depth of $o2$ is 3. We extend Definition 4.2 to any kind of variables. The depth of an intermediate variable is the longest path from this variable to the primary inputs. In sequential circuits, a present state variable is taken as a primary input. Thus, the depth of a present variable is 0 and the depth of a next state variable is the longest path from this variable to the primary inputs and present state variables.

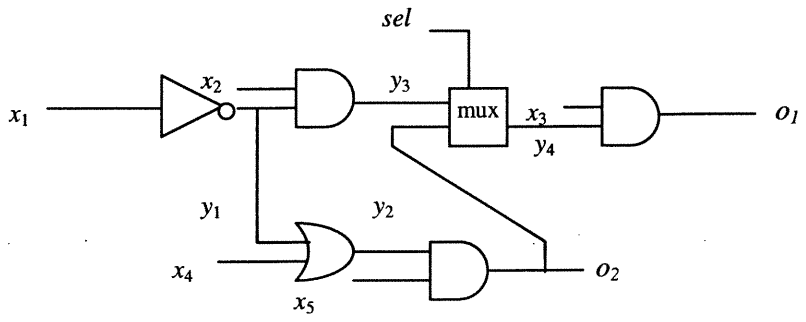


Figure 4.7 A combinational circuit

In the following subsections, we introduce an algorithm for combinational circuits first, then we adapt it for sequential circuits.

4.3.1 Static Variable Ordering Algorithms for Combinational Circuits

The basic idea of our static variable ordering algorithm for combinational circuits (*svoc*) is to calculate a *weight* for each variable. The weight of a variable reflects the contribution of this variable to the circuit, thus the variables with large weights should be put in the front of the list.

Motivated by rule 3 and Fujita's theorems [28], the algorithm is based on a depth-first search. During the search, a variable will experience three states: initial, visited and ordered. To visit as many variables as possible in one search, the algorithm starts from the output with the largest depth. During the search, the weights of inputs are always assigned by a value larger than outputs. Thus the weights of a variable's determining variables are always larger than a variable itself.

In a sub-circuit with control variables, since a control signal should come before its closely related data input variables and their determining variables, the weight of the control signal is larger than the data inputs and their determining variables' weights.

The calculation of the weight for each variable starts from the outputs. First, each output is assigned 0. Then from the output with the largest depth, the weight calculation is propagated toward the primary inputs. At each component $y = f(x_1, \ldots, x_n)$ except a multiplexer, the weight of x_i, $1 \le i \le n$, is given as

$$weight(x_i) = \begin{cases} weight(y) + 1 & x_i \ \ not \ \ visited \\ \max(weight(x_i), weight(y) + 1) & otherwise \end{cases} \quad (4.1)$$

If x_i has been visited and *weight(y)* + 1 is larger than weight(x_i), the components reached starting from x_i will have to be visited again and the weights will have to be re-calculated for the determining variables of x_i. However, if *weight(y)* + 1 is smaller than or equal to weight(x_i), the algorithm stops searching the x_i branch.

If the component is a multiplexer $y = f(sel, x_1, \ldots, x_n)$, the algorithm calculates the weights for the variables $x_1, \ldots, x_n$ using the same way until it reaches the primary inputs, but it memorizes the largest weight assigned to the determining variables of y. The weight of *sel* is then this largest weight plus 1.

The weight of a cross-operator is the weight of its output plus 0.5. According to Constraint 2, the abstract inputs of a cross-operator should come before the cross-operator. The value of 0.5 makes the weight of the cross-operator smaller than its inputs and does not violate the constraint in the order. If a cross-operator is used more than once in the circuit and has different weights depending on its outputs, the final weight of the cross-operator is the smallest one. By assigned the smallest

weight, the cross-operator will come after all its abstract inputs in the final order and satisfy Constraint 2.

After the algorithm visited all variables contributing to this output, it sorts the variables in a decreasing order of the weights in two parts: first it sorts all primary inputs, and second it sorts the other variables that include internal variables, cross-operators and outputs.

The algorithm marks all the visited variables as ordered and removes them from the circuit file. It then repeats the same procedure for the still unordered outputs. If all determining variables for an output have already been ordered, the algorithm would mark this output ordered and append it to the ordered variable list directly.

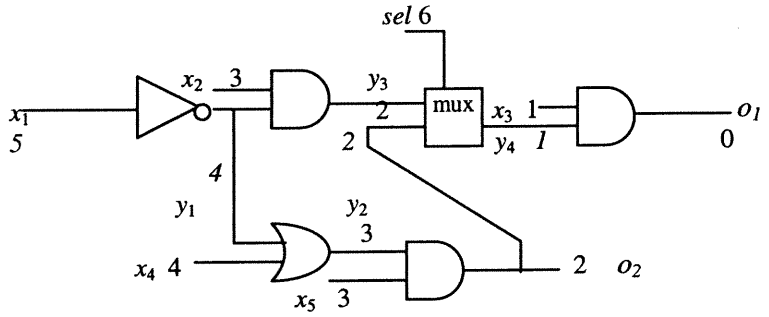
The algorithm *svoc*(C, L) is shown in Figure 4.8, where C is a given circuit file, L is the order generated by the algorithm. After initialization, the algorithm uses the sub-procedure *weight-calculation* to compute weights for the determining variables of an output. X and O are lists that contain all inputs and other variables visited in this sub-procedure. They are sorted in the main procedure and are appended to L respectively. Procedure *weight-assigning* is assigning a weight to a variable using Equation 4.1. Procedure *constraint-adjusting* is used at the end of the algorithm to check if all constraints have been observed. If there are two variables $a < b$ violating a constraint, just simply move a right after b. This move may result in another constraint violation, so all constraints have to be checked again until they are all satisfied.

An example of this algorithm applied to the circuit in Figure 4.7 is shown in Figure 4.9. x_1, x_2, x_3, x_4 and x_5 are primary inputs. The algorithm searches the circuit starting from o_1 that has larger depth than o_2. The variable y_1 has two fan-out branches and its new weight is larger than the old one, therefore the *NOT* component has to be visited twice in this propagation. The select signal *sel* is the

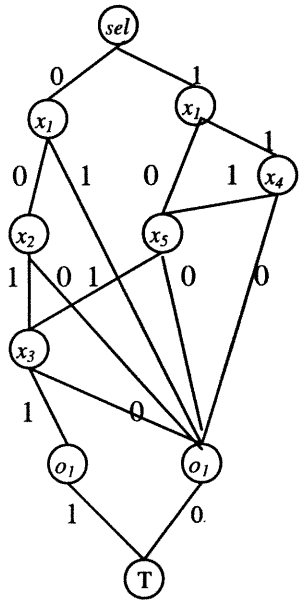
```
Procedure  SVOC (C, L)
  begin
      initialization
      for  all outputs i  in decreasing order of depth
          if the output i is not ordered
              weight-calculation ( i, X, O)
              L = append-to-tail (L, X, O)
              mark X and O as ordered and clear them
     constraint-adjusting (L)
  end

Procedure initialization
  begin
      L = ∅;  X = ∅;  O = ∅;
     mark all the variables as unordered
      for all variables i
            weight[i] = 0
  end

Procedure weight-calculation (i, X, O)
  begin
     from  i propagate  to the primary inputs
        for each component f   y = f (x1, …, xn) in this propagation process
           if  xi is not a select signal in a multiplexer
               weight-assigning (x1,.. xn,  y)
           else
               weight(select) = max (the weights of all signals starting from f) + 1
            if f is a cross-function
                  weight(f) = weight(y) + 0.5
      X = sort-in-decreasing-order (all primary inputs visited in this sub-procedure)
      O = sort-in-decreasing-order (all other variables visited in this sub-procedure)
   End

Procedure  weight-assigning (x1,.. xn,  y)
  begin
        for each xi ,  1 ≤ i ≤ n,
              if xi = 0
                  weight(xi) = weight(y)  + 1
              else if xi > 0 and not ordered
                  weight(xi) = max (weight(xi), weight(y) + 1)
  end

Procedure constraint-adjusting (L)
  begin
     repeat  check constraints for all secondary variables and cross-operators
              if two secondary variables or  cross-operators  a and  b are in the order b < a,
             and this violates one of  the constraints, reverse the order so that  a < b.
     until all constraints are satisfied
  end
```

Figure 4.8 A static variable ordering algorithm for combinational circuits

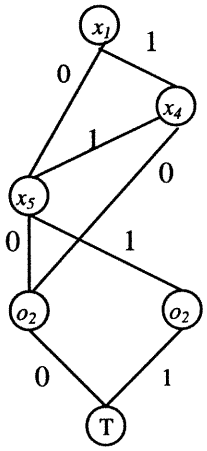
largest weight from all variables starting from the multiplexer plus 1. The final order for this example is $sel < x_1 < x_4 < x_2 < x_5 < x_3 < y_1 < y_2 < y_3 < o_2 < y_4 < o_1$. It is one of the best orders for this example. The resulting MDGs for o_1 and o_2 are shown in Figure 4.9(b) and (c) respectively. They are 55% smaller than using the reverse order.



(a) Circuit



(b) The result MDG for o_1



(c) The result MDG for o_2

Figure 4.9 An example of application of the *svoc* algorithm

If a circuit component is described by a table, a slightly different strategy is used in the weight calculation. First, we define $disv(x_i)$ as the number of distinct values x_i can take. For example, a circuit component described by a table is shown in Table 4.1, where m' is the output, r, c, x, m are the inputs, and *leq* is a cross-operator.

r	c	$leq(x, m)$	m'
0	*	*	*max*
1	1	*	x
2	0	0	m
3	0	1	x

Table 4.1 An example of a circuit component in tabular form

The number of distinct values that c, x and m can take in the table is 2. The number of distinct values that r can take is 4.

When a table is encountered in the propagation of the weight calculation, we add the number of distinct values (*disv*) a variable can take instead of adding 1 in Equation (4.1). The new equation is as follows:

$$weight(x_i) = \begin{cases} weight(y) + disv(x_i) & x_i \ \ not \ \ visited \\ \max(weight(x_i), weight(y) + disv(x_i)) & otherwise \end{cases} \quad (4.2)$$

This modification is motivated by heuristics 4 and 5. Since a cross-operator should be located after its secondary variables, we subtract 0.5 from its *disv*. If all (or part of) outputs are described by tables, the ordering algorithm starts from the output having the largest number of determining variables instead of the largest depth.

4.3.2 Static Variable Ordering Algorithms for Sequential Circuits

We extend this algorithm for sequential circuits, where the variable order influences not only the size of the MDG representation of the transition function but also the size of the MDG representation of the set of reachable states.

In the transition system of a model M, let $Y = (y_1, \ldots y_n)$ be the state variables, Y'= (y_1', ... y_n') the corresponding next state variables. We first determine a good order of the next state variables (y_1', ... y_n') based on two factors: their depth to the primary inputs, and the size of the sorts of concrete variables or the number of distinct terms of abstract variables. If all (or part of) the next state variables are given using tables, we use the number of determining variables instead of the depth to the primary inputs. For some abstract variables, we may not know the number of terms from a circuit. In this case, we assume that they have 3 terms. This is based on our experience that most of abstract state variables have at least 3 terms. After ordering the next state variables using the above factors, we use the procedure *constraint-adjusting* to modify the order list if there is a next state variable violating the constraints.

We order the determining variables *dv* of a next state variable y_i using the same algorithm as for combinational circuits. If there are some state variables dependent on other state variables, we ignore those dependent state variables in the process of propagation. Finally, we order the variables as follows: y_1, $dv(y_1')$, y_1', y_2, $dv(y_2')\backslash dv(y_1')$, y_2', ..., y_n, $dv(y_n')\backslash(dv(y_1')\cup \ldots.\cup dv(y_{n-1}'))$, y_n'.

The static variable ordering algorithm for sequential circuits (*svos*) is shown in Figure 4.10.

```
Procedure  SVOS(C, L)
  begin
      initialization
      S is the set of the next state variables
      S'= order-next-state-variables(S)
      for  each state variable i and i' in S'
             weight-calculation( i', X, O)
              L1 = append-to-tail(L, [i], X, O, [i'] )
      for  all outputs i  in decreasing order of depth
          if the output i is not ordered
              weight-calculation ( i, X, O )
          L = append-to-tail (L1, X, O)
          mark X and O as ordered and clear them
     constraint-adjusting (L)
  end

Function  order-next-state-variables(S)
  begin
      for each next state variables i
        if i is a concrete variable
            weight[i] = depth(i) + size(sorts)
        else weight[i] = depth(i) + number(terms)
      S' = sort-in-decreasing-order (S)
      S' = constraint-adjusting (S')
      return S'
  end
```

Figure 4.10 A static variable ordering algorithm for sequential circuits

In the next section, we prove the convergence of our variable ordering algorithm.

4.4 Proof of Convergence of the Algorithms

In the procedure *constraint-adjusting*, if two variables or cross-operators a and b are in the order $b < a$ and this violates one of the constraints, we reverse the order so that $a < b$. This may cause a new violation, so constraints are checked repeatedly until all of them have been satisfied. However, if such reversals always causes new violations to other constraints, the procedure *constraint-adjusting* would not terminate. For instance, suppose that there are three constraints $x_1 < x_2$, $x_2 < x_3$, $x_3 < x_1$, forming a circular chain. The procedure cannot terminate since it is

not able to find an order satisfying all the constraints. In other words, non-termination of the procedure *constraint-adjusting* occurs if there exist two or more constraints forming a circular chain: $x_1 < x_2$, $x_2 < x_3$, ..., $x_{n-1} < x_n$, $x_n < x_1$ $n \geq 2$.

In the following, we prove that non-termination does not happen in the case of MDG graphs modeling any RTL circuit without combinational loops (we do not allow false loops either, i.e., loops that exist topologically, but not logically due to dependencies between logical variables). We do that by inspecting the sources of the constraints (Constraints 1, 2 and 3 mentioned in Section 4.1). Note that each x_i in the chain must appear on the left-hand side of a constraint and the right-hand side of another constraint. x_i cannot be a cross-operator since a cross-operator does not appear on the left-hand side of a constraint, thus Constraint 2 need not be considered. All constraints in the circular chain can only be caused by Constraint 1 and 3.

Before proving this statement, however, we discuss how Constraint 1 can be introduced by a circuit component. As we explained in Section 4.1, Constraint 1 is caused by an abstract variable appearing as a secondary variable in an edge label of a node. This situation can be produced by a combinational module such as a multiplexer or a function. For example, in Figure 4.2, constraints $x_1 < y$ and $x_2 < y$ are introduced by a multiplexer. Note that in MDG a register such as shown in Figure 4.11 does not cause constraint $y' < y$ since the next state variable is determined by the MDG as function of the present state variables and inputs. Therefore, y' is a primary variable and depends on y, and y as a secondary variable must come before y' in the order.

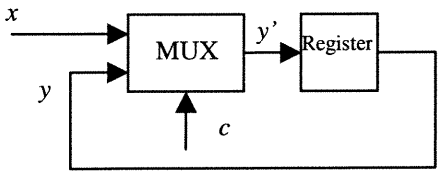


Figure 4.11 An example of a register

Theorem 1: Procedure *constraint-adjusting* terminates for any circuit.

First, assume that $x_1 < x_2$, $x_2 < x_3$, ..., $x_n < x_1$ are all caused by Constraint 1, which means x_{i-1} labels an edge of a node labeled by x_i. Suppose $x_1 < x_2$, $x_2 < x_3$, ..., $x_n < x_1$ are caused by some combinational modules m_i , i = 1, ..., n. The circuit that introduces all the constraints in the chain is shown in Figure 4.12.

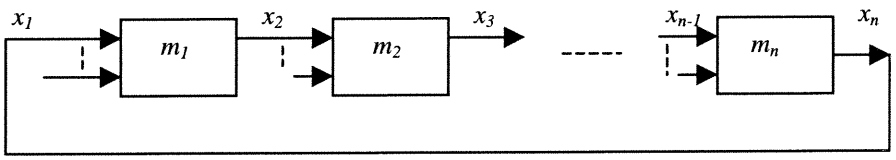


Figure 4.12 A circuit causing non-termination of procedure *constraint-adjusting*

However, as mentioned earlier, we do not allow combinational loops in RTL circuits. Thus, the constraints $x_1 < x_2$, $x_2 < x_3$, ..., $x_n < x_1$ can not be all caused by Constraint 1.

Second, suppose $x_1 < x_2$, $x_2 < x_3$, ..., $x_n < x_1$ are all caused by Constraint 3. Since we first fix the order of the next state variables, say $x_1' < \ldots < x_n'$, the present state variables must be in a corresponding order such as $x_1 < \ldots < x_n$. The constraints $x_1 < \ldots < x_n$ are thus imposed. Since no circular chain of constraints over the next-

state variables $x_1' < \ldots < x_n' < x_1'$ would be fixed by our algorithm, $x_1 < x_2$, $x_2 < x_3$, ..., and $x_n < x_1$ thus cannot be all caused by Constraint 3.

Third, without loss of generality, suppose $x_1 < x_2$ is caused by Constraint 1, and $x_n < x_1$ is caused by Constraint 3. Our algorithm modifies the order of the next state variables if some of them violate Constraint 1. The present state variables are arranged according to the order of the next state variables. Thus, x_n and x_1 must be present state variables. However, a present state variable can be a primary variable (labeling a node) only in an MDG that assigns the initial state value. It is used in the first step of reachability analysis and depends on some fresh variables or constants that can be freely moved in the order as their constraint is only relative to the present-state variable. In all other instances, present-state variables are secondary variables. In other words, a present state variable cannot be the output of a combinational module that may cause the present state variable to appear on the right-hand side of a constraint. Thus the constraint $x_{n-1} < x_n$ cannot be caused by Constraint 1 and can only be caused by Constraint 3 too. Similarly, we can show that $x_{n-2} < x_{n-1}$, $x_{n-3} < x_{n-2}$, ..., $x_1 < x_2$ must also be caused by Constraint 3. This contradicts the assumption that $x_1 < x_2$ is caused by Constraint 1.

Therefore, the procedure *constraint-adjusting* always terminates.

□

4.5 Experimental Results

The algorithms have been implemented in Prolog and integrated in the MDG package. We conducted experiments to evaluate the efficiency of the ordering methods. In this section, we present experimental results on a number of IFIP benchmark circuits, an Island Tunnel Controller and an ATM Switch Fabric

model. In this thesis, all the experimental results were carried out on a Sun Ultra 10 workstation with 333MHz and 1GB of memory.

4.5.1 Experiments on the IFIP Benchmark Circuits

Table 4.2 shows the experimental results on some IFIP benchmark circuits [40] for "*seq_traverse*" operation. "*seq_traverse*" is an application provided by MDG package for reachability analysis for an ASM with an invariant which is always true [70]. Designers who have experience with MDGs provided the original orders. '-' means that the verification did not terminate in 20 hours.

The algorithm gave a better order than the manually generated ones in most cases, especially in circuits with a large number of variables when manual ordering becomes difficult. In particular, in the case of the benchmark "queue", the improvement was quite dramatic, and in the case of "buffer" our order allowed the reachability analysis to complete.

circuit	No. of state var.& functions	our algorithm		original order	
		nodes	time(sec)	nodes	time(sec)
minmax	3 & 14	166	0.12	175	0.16
tlc	2 & 18	245	1.22	240	1.17
gcd	3 & 34	375	0.26	386	0.26
tama	9 & 49	2211	1.44	2260	1.46
filter	12 & 21	2957	4.08	3402	4.19
queue	8 & 26	6551	3.58	18695	22.16
buffer	256 & 257	2307	2.21	-	-

Table 4.2 Experimental results for IFIP benchmark circuits

4.5.2 Invariant Checking on the Island Tunnel Controller

In this subsection we apply our static ordering algorithm to the Island Tunnel Controller (ITC) example. ITC was originally introduced by Fisler and Johnson [25]. The ITC controls the traffic lights at both ends of a tunnel based on information collected by sensors installed at both ends of the tunnel, a single lane tunnel connecting the mainland to an island, as shown in Figure 4.13. At each end of the tunnel, there is a traffic light. There are four sensors for detecting the presence of vehicles: one at the tunnel entrance (*ie*) and one at the tunnel exit on the island side (*ix*), and one at the tunnel entrance (*me*) and one at the tunnel exit on the mainland side (*mx*).

In [25], the following constraint is imposed: "at most sixteen cars may be on the island at any time". The number "sixteen" can be taken as a parameter and it can be any natural number. The constraint can thus be read as follows: "at most n ($n \geq 0$) cars may be on the island at any time".

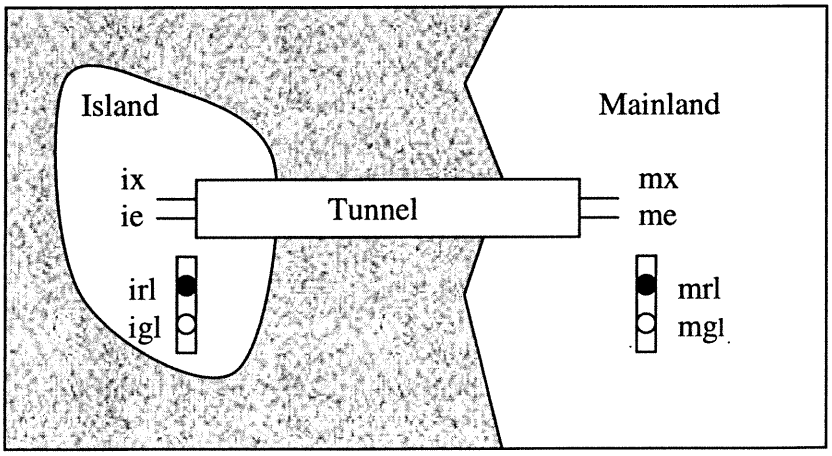


Figure 4. 13 The Island Tunnel Controller

We verified three safety properties P1, P2, P3 on ITC. The detailed description of the ITC example and the properties verified are presented in Appendix A. Table 4.3 shows the results of checking the conjunction of P1, P2 and P3 with various counter widths. Here we suppose the number of cars allowed on the island and in the tunnel equals 2^n where n is the counter width. We verified the properties with the counter width ranging from 4 to 8 bits. Because MDGs can model an abstract data path, we also use an abstract variable to describe the counter.

counter width	our algorithm		original order	
	nodes	time(sec)	nodes	time(sec)
abstract	3244	2.36	3826	2.89
4 bits	7687	8.33	65815	186.27
5 bits	8945	10.44	69827	216.11
6 bits	12275	17.76	82468	333.38
7 bits	18931	36.62	107748	667.83
8 bits	32243	89.51	158308	1684.62

Table 4.3 Experimental results for invariant checking on ITC

The experimental results in Table 4.4 show that the MDGs based on the orders from our algorithm are 69.8% on the average smaller than the MDGs built from manual orders. This example further shows that our algorithm can generate a good order for the MDG applications.

4.5.3 Property Checking on the Fairisle ATM Switch Fabric

Another circuit we conducted the experiments on is the Fairisle 4×4 ATM (Asynchronous Transfer Mode) switch fabric. The device was in use in the

Cambridge Fairisle network [17], designed at the Computer Laboratory of the University of Cambridge.

The 4×4 Fairisle switch consists of three types of components: the input port controllers, the output port controllers and the switch fabric, as shown in Figure 4.14. An (Fairisle) ATM cell consists of a header (one-byte tag containing routing information as shown in Figure 4.15) and a fixed number of octets. Cells are switched from input ports to output ports according to the header.

The behavior of the switch is cyclical. In each cycle or frame, the input port controllers synchronize incoming data cells, prepend control information in the front of the cells, and send them to the fabric. The fabric waits for cells to arrive, strips off the header, arbitrates between cells destined to the same port, sends successful cells to the appropriate output port controllers, and passes acknowledgments from the output port controllers to the input port controllers.

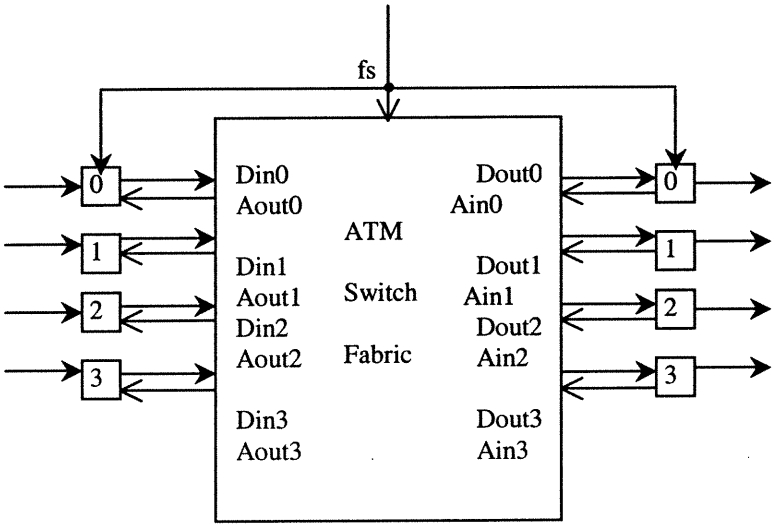


Figure 4.14 The Fairisle ATM Switch

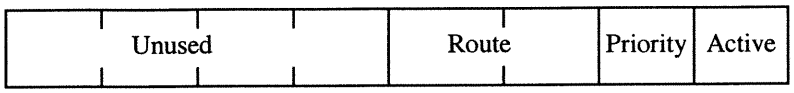


Figure 4.15 The header (routing tag) of a Fairisle ATM cell

We verified properties P1 to P4 on this switch. The hardware description of the switch and properties are presented in Appendix B. The model contains 34 state variables and 111 functions. The experimental results in Table 4.4 show that the MDGs based on the orders from our algorithm are 30.6% smaller than the MDGs built from manual orders. This example demonstrates that our algorithm can also generate a good order for the MDG-based property checker.

Property	our algorithm		original order	
	nodes	time(sec)	nodes	time(sec)
P1	19572	42.39	28077	361.01
P2	19633	40.69	28147	376.85
P3	22249	52.05	32139	331.50
P4	19601	38.08	28394	366.43

Table 4.4 Experimental results of property checking of the ATM model

Summary

This chapter presented a method for automatically generating a static variable order based on information about the circuit. Variable ordering on MDGs is more difficult than ROBDDs because of the first order terms that appear in MDGs. Our method can generate a good order for most benchmarks we experimented with.

The MDG package became more efficient with our static variable ordering algorithm. In the next chapter, we present a dynamic variable ordering algorithm to change an order during the verification process.

Chapter 5 Dynamic Variable Ordering on MDG

MDGs become more efficient with the development of heuristic methods for automatically generating a variable order based on information about the design under verification. A static method yields an order for a verification application from start to finish. However, in practice, the ideal order may change as the verification algorithm moves through different phases. The application may need a new order in each such phase.

Dynamic variable ordering reorders the variables of an MDG to reduce its size. It differs from the static order where the variables are ordered before the MDG is created. Variable reordering is still being studied intensively, and one note-worthy method is the *sifting* algorithm developed by Rudell in 1993 [57]. Some significant improvements have been made on sifting by Panda and Meinel separately in 1995 and 1997 [55, 50].

In this chapter, we present a dynamic variable ordering algorithm on MDG. In Section 5.1, we introduce variable swapping that is a basic operation for variable reordering. The implementation of the basic sifting algorithm of Rudell on MDG is discussed in Section 5.2. In Section 5.3, we present a dynamic variable ordering algorithm which integrates symmetry and group sifting. Experimental results are discussed in Section 5.4.

5.1 Variable Swapping in Multiway Decision Graphs

Dynamic reordering improves the variable order by a series of swaps of adjacent variables. The basic operation in reordering is thus that of variable swapping. We

begin with an introduction of the implementation of the swap operation in MDG, then continue with a discussion of the effects of the swap operation on the variable order and the constraints imposed on variable swapping.

5.1.1 Implementation of a Variable Swapping Operation

Variable swapping involves moving all MDG nodes at level i to level i+1 and nodes at level i+1 to level i. Level 0 is the root node of an MDG. All nodes at a level are labeled by the same variable. The level itself is also labeled by the same variable. The variables that label the levels must appear in a custom order. The level that a variable labels does not reflect its exact position in the order since secondary variables do not label the levels in the particular graph, but may in other MDGs that describe the design. All of them follow the same global order.

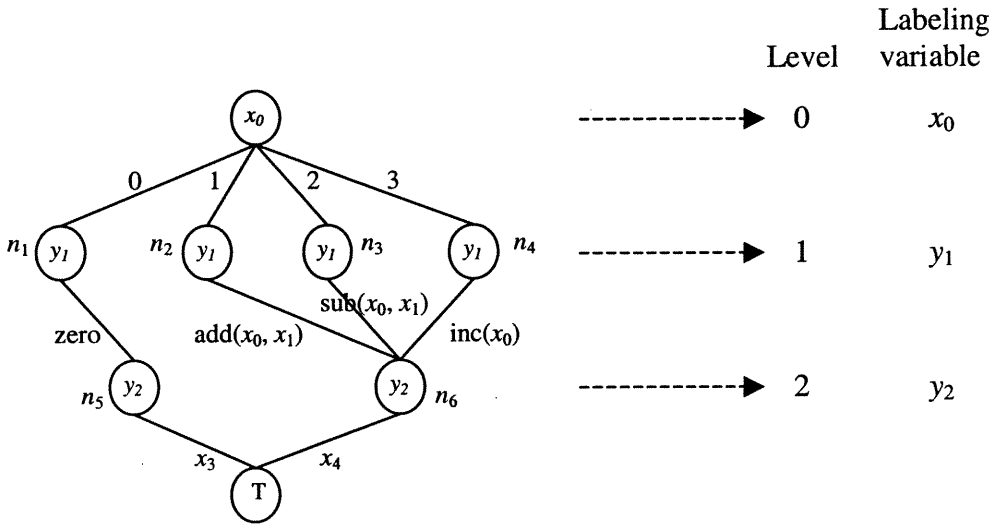


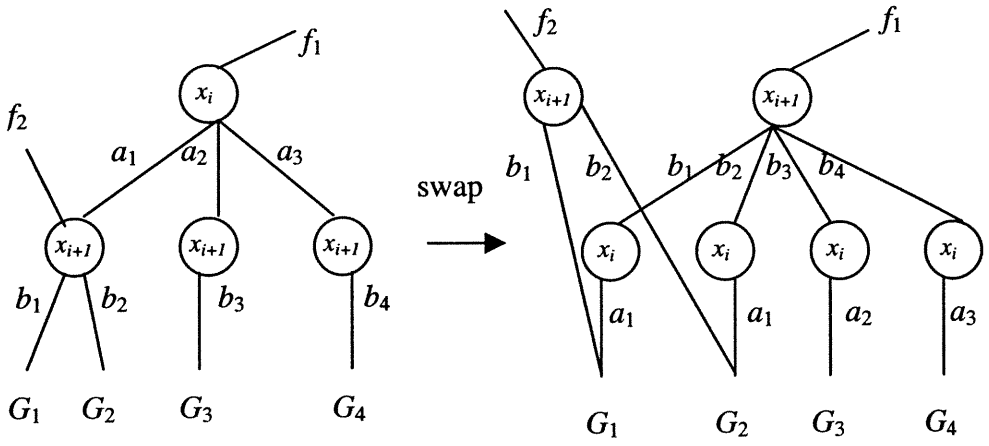
Figure 5.1 A 3-level MDG

Figure 5.1 shows an example of the MDG with three levels. A swap of variables y_1 and y_2 means moving node n_5 and n_6 to level 1 and moving n_1, n_2, n_3 and n_4 to level 2. The MDG after this swap can be found in Figure 4.6(d) on page 52.

Because an MDG node can be labeled by an abstract variable, a concrete variable or a cross-term, a variable swapping operation is of 3 kinds: a swap between two abstract variables, a swap between two concrete variables (or cross-terms) and a swap between a concrete variable (or cross-term) and an abstract variable.

We first present swap between two abstract variables. Suppose we wish to swap abstract variables x_i and x_{i+1} in an MDG shown in Figure 5.2(a). The depicted MDG represents the following Direct Formulas (DFs) f_1 and f_2:

$$\begin{aligned} f_1 = &((x_i = a_1) \wedge (((x_{i+1} = b_1) \wedge G_1) \vee ((x_{i+1} = b_2) \wedge G_2))) \vee \\ &((x_i = a_2) \wedge ((x_{i+1} = b_3) \wedge G_3)) \vee \\ &((x_i = a_3) \wedge ((x_{i+1} = b_4) \wedge G_4)) \qquad (5.1) \\ f_2 = &((x_{i+1} = b_1) \wedge G_1) \vee ((x_{i+1} = b_2) \wedge G_2)) \end{aligned}$$



(a) Before swapping (b) After swapping

Figure 5.2 Variable swap between two abstract variables

In first order logic, for any two formulas P and Q, there are the following axioms:

$$P \wedge Q \Leftrightarrow Q \wedge P,$$

$$\mathrm{P} \wedge (\mathrm{Q} \wedge \mathrm{R}) \Leftrightarrow (P \wedge Q) \wedge R,$$

and $$\mathrm{P} \wedge (\mathrm{Q} \vee \mathrm{R}) \Leftrightarrow (P \wedge Q) \vee (P \wedge Q).$$

Thus, we can deduce the following formulas from (5.1) :

$$
\begin{aligned}
f_1 \Leftrightarrow & (((x_i = a_1) \wedge ((x_{i+1} = b_1) \wedge G_1)) \vee ((x_i = a_1) \wedge ((x_{i+1} = b_2) \wedge G_2))) \vee \\
& ((x_{i+1} = b_3) \wedge ((x_i = a_2) \wedge G_3)) \vee \\
& ((x_{i+1} = b_4) \wedge ((x_i = a_3) \wedge G_4)) \\
\Leftrightarrow & ((x_{i+1} = b_1) \wedge ((x_i = a_1) \wedge G_1)) \vee \\
& ((x_{i+1} = b_2) \wedge ((x_i = a_1) \wedge G_2)) \vee \\
& ((x_{i+1} = b_3) \wedge ((x_i = a_2) \wedge G_3)) \vee \\
& ((x_{i+1} = b_4) \wedge ((x_i = a_3) \wedge G_4)) \qquad (5.2)
\end{aligned}
$$

Figure 5.2(b) shows the MDG after swapping variables x_i and x_{i+1}. This MDG represents formula (5.2), which is equivalent to (5.1). DF f_2 remains undisturbed.

Note that in Figure 5.2, each edge issuing from nodes in level i must reach a node labeled by x_{i+1} before swapping and must reach a node labeled by x_i after swapping. According to MDG well-formed condition 5: the set of abstract variables having primary occurrences along a path is the same for all paths in a given graph. Thus, no edge issuing from nodes in level i reaches subgraphs in level i+2 directly. However, when swapping an abstract variable and a concrete variable (or cross-term) or two concrete variables (or cross-terms), there may exist an edge or edges issuing from the nodes in level i to the subgraphs in level i+2 directly.

Figure 5.3 shows variable swapping between an abstract variable and a concrete variable. x_i is of an abstract sort. x_{i+1} is of a concrete sort of enumeration $\{c_1, c_2, c_3, c_4\}$. Edge a_3 is issued from u (a node in level i) to subgraph G_4 directly without passing through level i+1. The MDG in Figure 5.3(a) represents the formula:

$$
\begin{aligned}
f_1 \Leftrightarrow & ((x_i = a_1) \wedge (((x_{i+1} = c_1) \wedge G_1) \vee ((x_{i+1} = c_2) \wedge G_2))) \vee \\
& ((x_i = a_2) \wedge ((x_{i+1} = c_3) \wedge G_3)) \vee \\
& ((x_i = a_3) \wedge G_4) \qquad (5.3)
\end{aligned}
$$

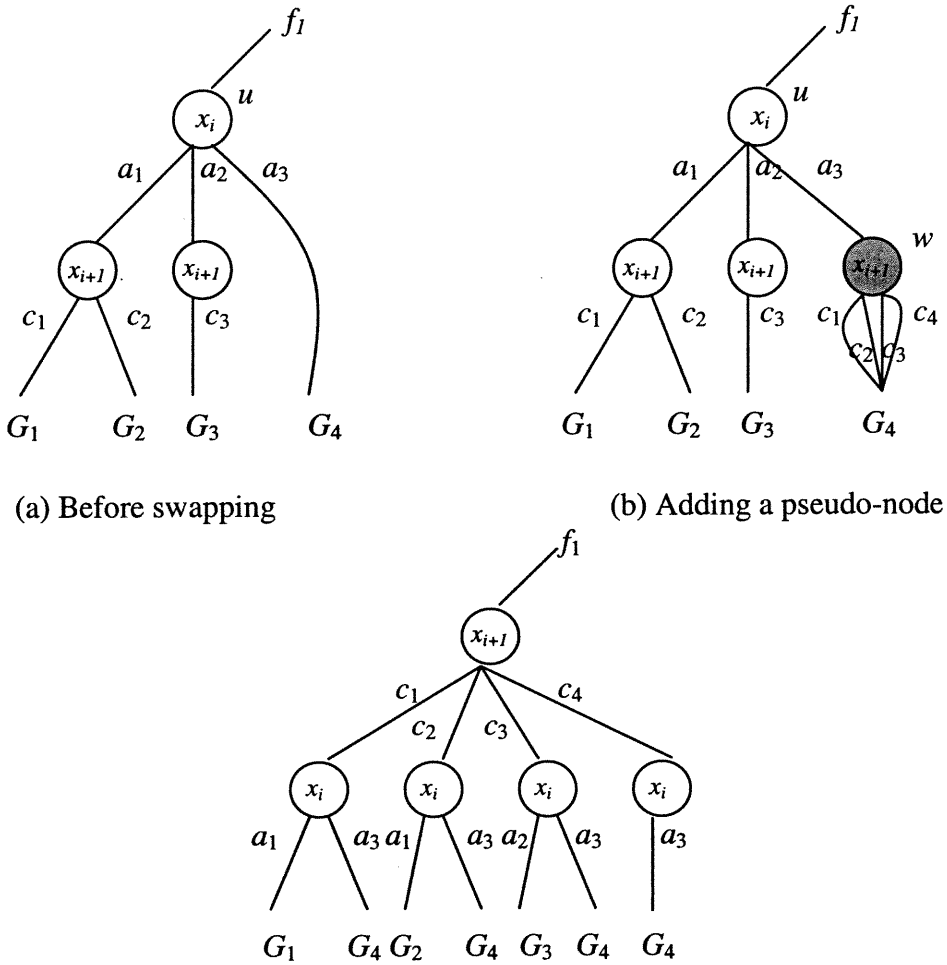


Figure 5.3 Variable swapping between an abstract variable and a concrete variable

Because $((x_{i+1} = c_1) \vee (x_{i+1} = c_2) \vee (x_{i+1} = c_3) \vee (x_{i+1} = c_4)) \equiv \mathbf{T}$, we can deduce the following equivalent formulas from (5.3) :

$$
\begin{aligned}
f_1 \Leftrightarrow\ & ((x_i = a_1) \wedge (((x_{i+1} = c_1) \wedge G_1) \vee ((x_{i+1} = c_2) \wedge G_2))) \vee \\
& ((x_i = a_2) \wedge ((x_{i+1} = c_3) \wedge G_3)) \vee \\
& ((x_i = a_3) \wedge (((x_{i+1} = c_1) \wedge G_4) \vee ((x_{i+1} = c_2) \wedge G_4) \vee \\
& \qquad ((x_{i+1} = c_3) \wedge G_4) \vee ((x_{i+1} = c_4) \wedge G_4))) \qquad (5.4) \\
\Leftrightarrow\ & ((x_{i+1} = c_1) \wedge ((x_i = a_1) \wedge G_1)) \vee \\
& ((x_{i+1} = c_2) \wedge ((x_i = a_1) \wedge G_2)) \vee \\
& ((x_{i+1} = c_3) \wedge ((x_i = a_2) \wedge G_3)) \vee
\end{aligned}
$$

$$
\begin{aligned}
&((x_{i+1} = c_1) \wedge ((x_i = a_3) \wedge G_4)) \vee \\
&((x_{i+1} = c_1) \wedge ((x_i = a_3) \wedge G_4)) \vee \\
&((x_{i+1} = c_1) \wedge ((x_i = a_3) \wedge G_4)) \vee \\
&((x_{i+1} = c_1) \wedge ((x_i = a_3) \wedge G_4)) \\
\Leftrightarrow\ &((x_{i+1} = c_1) \wedge (((x_i = a_1) \wedge G_1) \vee ((x_i = a_3) \wedge G_4))) \vee \\
&((x_{i+1} = c_2) \wedge (((x_i = a_1) \wedge G_2) \vee ((x_i = a_3) \wedge G_4))) \vee \\
&((x_{i+1} = c_3) \wedge (((x_i = a_2) \wedge G_1) \vee ((x_i = a_3) \wedge G_4))) \vee \\
&((x_{i+1} = c_4) \wedge (((x_i = a_3) \wedge G_1) \vee ((x_i = a_3) \wedge G_4))) \qquad (5.5)
\end{aligned}
$$

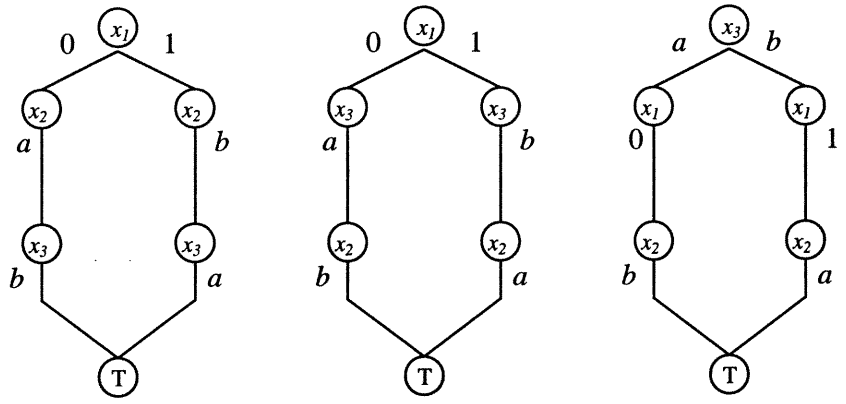
A pseudo node w is added in Figure 5.3 (b), which represents (5.4). Formula (5.4) can be further reduced to (5.5) where variables x_i and x_{i+1} are swapped. The MDG for (5.5) is shown in Figure 5.2(c).

Variable swaps between concrete variables (cross-terms) or a concrete variable (cross-term) and an abstract variable can be implemented in a similar way.

The variable swapping operation provides the basis for the sifting algorithm. It is easily seen that the swapping operation is completely local since only the nodes at level i and level i+1 need to be traversed.

5.1.2 The Effects of the Swapping Operation on Variable Order

In an MDG, swapping two adjacent variables does not imply that they just exchange the positions in the ordering list. For example, suppose we apply two swap operations to the abstract variable x_3 in the MDG shown in Figure 5.4. The MDG is under the order $x_1 < a < b < x_2 < x_3$. x_1 is a Boolean variable and x_2, x_3 are abstract variables. a and b are secondary variables. There are four constraints in the MDG: $a < x_2$, $a < x_3$, $b < x_2$, and $b < x_3$.



(a) Before sifting (b) Swap x_3 and x_2 (c) Swap x_3 and x_1

Figure 5.4 Effects of swap operation on the order

First, swapping x_2 and x_3 results in a new order $x_1 < a < b < x_3 < x_2$. Second, after swapping x_3 and x_1, we can not just swap their positions because of the constraints imposed between a, b and x_3. A new order $a < b < x_3 < x_1 < x_2$ is obtained by moving a, b with x_3 before x_1.

5.1.3 Constraints on a Variable Swap

Constraints sometimes not only affect the variable order after a swap, but also decide whether two variables can be swapped.

In Section 4.1, we discussed that the present and next state variables must be in a corresponding order, i.e., if the present state variables are in the order $y_1 < y_2 < y_3$, then the next state variables should be in the order $y_1' < y_2' < y_3'$. So if two present/next state variables have been exchanged, the corresponding next/present state variables should be exchanged as well. For instance, suppose that an initial order generated by our static variable ordering algorithm for sequential circuits (*svos*) is as follows:

$$y_1 < x_1 < x_2 < y_1' < y_2 < x_3 < x_4 < y_2' < y_3 < x_5 < y_3'.$$

x_1, x_2, x_3, x_4, x_5 are the determining variables of the next state variables y_1', y_2' and y_3'.

If we apply swap operations repeatedly to y_1 until it reaches y_2, the resulting order is

$$x_1 < x_2 < y_1' < y_1 < y_2 < x_3 < x_4 < y_2' < y_3 < x_5 < y_3'.$$

After swapping y_1 and y_2, their next state variables are not in the corresponding order. In this case, we have to take a series of swaps to move y_1' after y_2'. The resulting order is then

$$x_1 < x_2 < y_2 < y_1 < x_3 < x_4 < y_2' < y_1' < y_3 < x_5 < y_3'.$$

However, such additional swaps take a lot of computation time, and may not decrease the size of MDGs very much. We will introduce a dynamic reordering algorithm to avoid these useless swaps in Section 5.3.

In the next section, we present a basic sifting algorithm using the swapping operation.

5.2 A Basic Sifting Algorithm on MDG

We have implemented a basic sifting algorithm following Rudell' sifting algorithm [57]. This algorithm is triggered by the growth of the number of nodes in the MDG. Every time the MDGs grow to a specific size, a sifting process is invoked automatically, just as garbage collection.

During a sifting process, each primary variable in an MDG is examined in turn and is moved up and down in the order so as to take all positions successively. The variable is then returned to the position where the minimum size of the MDG was obtained. The process then continues with another variable.

For instance, suppose we want to sift variable x_2 in Figure 4.7(b). *sel*, x_1, x_2, x_3, x_4, x_5, and o_1 are primary variables. Figure 5.5 shows the variable permutations that are explored when applying the sifting algorithm to x_2. It is exchanged with its successor variable until it reaches the end of the order (bottom of the MDG), then it is exchanged with predecessor variable until it reaches the head of order (top of the MDG). This sifting up↔down process finds the optimum position for variable x_2 assuming all other variables remain fixed. In this example, the initial order and the fourth order are the best ones.

variables	swap	size of MDG (nodes)
sel, x_1, x_2, x_3, x_4, x_5, o_1	initial	9
sel, x_1, x_3, x_2, x_4, x_5, o_1	swap(x_2, x_3)	11
sel, x_1, x_3, x_4, x_2, x_5, o_1	swap(x_2, x_4)	12
sel, x_1, x_3, x_4, x_5, x_2, o_1	swap(x_2, x_5)	9
sel, x_1, x_3, x_4, x_5, o_1, x_2	swap(x_2, o_1)	12
sel, x_1, x_2, x_3, x_4, x_5, o_1	initial	9
sel, x_2, x_1, x_3, x_4, x_5, o_1	swap(x_1, x_2)	11
x_2, *sel*, x_1, x_3, x_4, x_5, o_1	swap(*sel*, x_2)	18

Figure 5.5 Sifting algorithm example

It is possible that some variables are already in their best positions in the order, such as variable *sel* in the example above, or find their best positions in the first several swaps. Continued moves of these variables only increase the size of the MDG. To avoid this situation, we set a limit for the growth of the MDG. The sifting of a variable stops in this direction when the MDG grows twice the optimum size found so far.

The sifting process searches for the best position for each variable. This method is very effective in reducing the MDG size, although it sometimes takes a long

computation time. Sifting n variables usually requires $O(n^2)$ swaps. In the next section, we present an algorithm that improves and speeds up sifting.

5.3 Sifting-based Variable Reordering Algorithm on MDG

Sifting is based on the idea of searching for the best position for each variable. Individual variables are moved through the whole order, one at a time. However, the total time for sifting grows very fast with the number of variables. In applications with thousands of variables, the time to sift one variable over the whole order takes too long.

Several attempts have been made to improve the performance of the original sifting on ROBDD. One improvement is to keep information about symmetry variables that can be swapped without any change in the size of the ROBDD. This is called *symmetry sifting*. As observed in [55] by Panda, knowing some variables are symmetrical in a function can help produce better variable orders for ROBDDs. He proposed an extension of sifting that may sift groups of symmetry variables simultaneously to produce better results. However, his definition and detection of symmetry variables cannot be used directly in MDGs which contain abstract variables and uninterpreted function symbols.

The other improvement is a method of *block-restricted sifting* proposed by Meinel [50]. This method reduces the number of swap operations by restricting their application on variables within certain blocks. Determination of the variables blocks follows from computing an ROBDD characteristic called *subfunction profile*.

In this section, we first investigate symmetry sifting on MDG. We then propose a group sifting to speed up sifting. Like *block-restricted sifting*, our group sifting

method restricts sifting within blocks. However, our way of dividing variables into groups is an extension of our static variable ordering method.

5.3.1 Symmetry Sifting

Definition 5.1 An MDG representing the direct formula f is symmetrical in x_i and x_j if the interchange of x_i and x_j leaves the MDG function unchanged. x_i and x_j are called symmetry variables in f (or MDG).

In other words, after the interchange of x_i and x_j, all the nodes in the MDG labeled by x_i are now labeled by x_j and all the nodes labeled by x_j are now labeled by x_i.

For example, consider an OR gate ($y = x_1 \vee x_2$). Exchanging variables x_1 and x_2 leaves the MDG function unchanged. The MDGs before and after exchanging are shown in Figure 5.6.

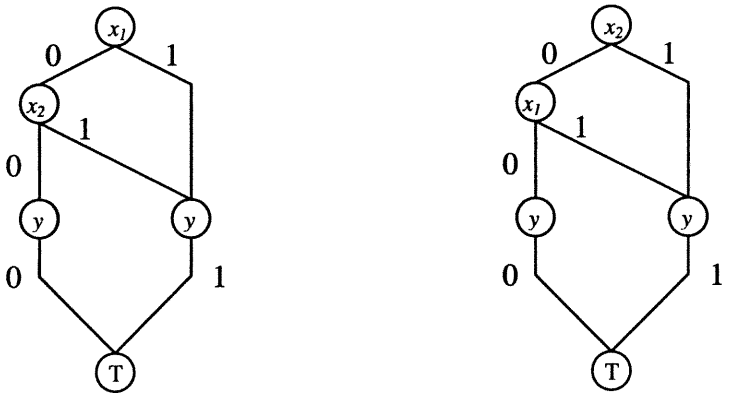
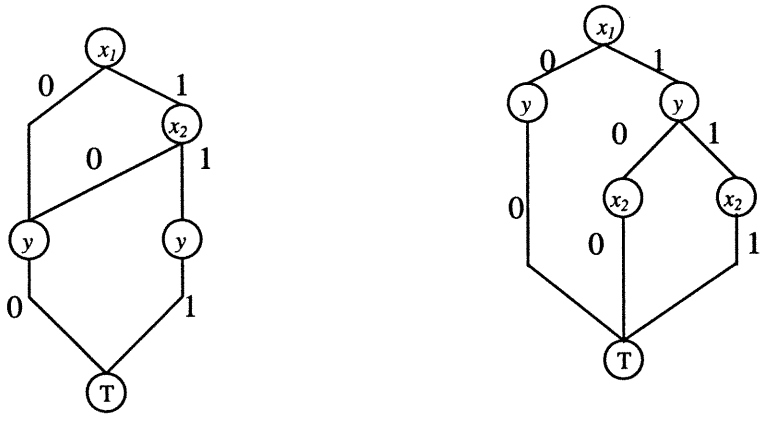


Figure 5.6 An example of symmetry variables

It is easy to see that interchanging symmetric variables in the order does not change the size of an MDG. Furthermore, it has been empirically observed that

symmetry variables should stay together in the order. A simple example is an AND gate ($y = x_1 \wedge x_2$). x_1, x_2 are symmetry variables in the MDG shown in Figure 5.7 (a). Once we separate x_1 and x_2, the MDG representation shown in (b) has one more node than (a).



(a) under order $x_1 < x_2 < y$ (b) under order $x_1 < y < x_2$

Figure 5.7 MDGs for an AND gate

An extreme example for the fact that symmetry variables should stay together is a circuit of the form $(x_1 \wedge x_2) \vee \ \ldots \vee (x_{2n-1} \wedge x_{2n})$, where x_1 and x_2, ..., x_{2n-1} and x_{2n} are symmetry variables. The MDG size is linear when we position the symmetry variables together. It becomes exponential when we separate them.

In the sifting algorithm, every variable that is moved up and down is at some point adjacent to other variables. Hence, if we combine the checking for symmetry of variables with sifting, we can find all the symmetry variables pairs. After swapping two adjacent variables, we test if the new MDG is identical to the old one. If yes, then x_i and x_{i+1} are symmetry variables. They will move together in the rest of the sifting process.

Suppose x_1, x_2, x_3, x_4, x_5, and x_6 are the primary variables in an MDG. We wish to sift variable x_2 as shown in Figure 5.8. After the first 3 swaps, we identify x_2 and

x_5 as symmetry variables and group them together. Sifting one variable then turns into sifting a group of variables (with two variable instances). One swap between the group and the next adjacent variable includes two swaps — each variable in the group has to swap with the next variable. The group of variables continues to check symmetry with other variables and may collect more variables along the way. In one sifting process, every two variables only need to check symmetry once. Thus, when the group is moved back, there is no need to check symmetry for variables which have been swapped before.

variables	swap	identify symmetry variables	
$x_1, x_2, x_3, x_4, x_5, x_6$	initial		
$x_1, x_3, x_2, x_4, x_5, x_6$	swap(x_2, x_3)	no	
$x_1, x_3, x_4, x_2, x_5, x_6$	swap(x_2, x_4)	no	
$x_1, x_3, x_4, x_5, x_2, x_6$	swap(x_2, x_5)	yes	bind x_5 and x_2
$x_1, x_3, x_4, x_5, x_6, x_2$	swap(x_2, x_6)	no	
$x_1, x_3, x_4, x_6, x_5, x_2$	swap(x_5, x_2)	-	
$x_1, x_3, x_4, x_2, x_5, x_6$	initial	-	
$x_1, x_3, x_2, x_4, x_5, x_6$	swap(x_4, x_2)	-	
$x_1, x_3, x_2, x_5, x_4, x_6$	swap(x_4, x_5)	-	
$x_1, x_2, x_3, x_5, x_4, x_6$	swap(x_3, x_2)	-	
$x_1, x_2, x_5, x_3, x_4, x_6$	swap(x_3, x_2)	-	
$x_2, x_1, x_5, x_3, x_4, x_6$	swap(x_1, x_2)	no	
$x_2, x_5, x_1, x_3, x_4, x_6$	swap(x_1, x_5)	no	

Figure 5.8 An example of symmetry sifting

We have explored several properties from the observation of symmetry variables to help identifying symmetry variables. If x_i and x_j are symmetry variables, the following necessary conditions must hold:

1. x_i and x_j must be of the same sort.
2. All the nodes labeled by variable x_i have the same number of distinct edges as the nodes labeled by variable x_j.
3. All edges into nodes labeled by x_j are from nodes labeled by x_i.

Only when a pair of variables satisfies the above properties, we check if it is a symmetry pair. This checking is completely local since we only need to compare the nodes at the two adjacent levels.

Symmetry sifting for a group of m variables (every two variables in this group are symmetry variables) from one end of the order to the other end requires $m(n-m)$ swaps, where n is the total number of variables. This is roughly equivalent to sifting the m variables one at a time.

In the next subsection, we introduce *group sifting* that can speed up the sifting operation.

5.3.2 Group Sifting

Basic sifting and symmetry sifting tend to be time consuming. To speed up sifting, we propose a method called group sifting. The main idea is to divide the variables into groups in which the variables are close to each other and each variable only sifts within its group. This is motivated by the observation that topologically close variables should stay together in the order (heuristic rule 1 in Section 4.2).

Suppose a group of variables are close to each other (such as being the determining variables of an output) and another variable is not related to this group (such that this variable and any variable of this group do not contribute to any common output). Sifting this variable among the group only increases the size of the MDG because this variable is not related to any of its adjacent variables and it

has to pass all information flow received from the nodes above itself to the nodes below. No information reduction can be achieved with this variable. Thus, if we can divide the variables into several groups so that the variables in a group are close to each other, we then only apply sifting in each group, i.e., a variable sifts up and down only in the scope of its own group.

The difficulty in group sifting is how to identify the groups. An intuitive idea is to divide the variables into groups according to the next state variables. Our reordering algorithm starts with the order generated from the static variable ordering algorithm for sequential circuits discussed in the last chapter. The static algorithm arranges the present state variables and the next state variables and their determining variables as follows: y_1, $dv(y_1')$, y_1', y_2, $dv(y_2')\backslash dv(y_1')$, y_2', ..., y_n, $dv(y_n')\backslash(dv(y_1') \vee .... \vee dv(y_{n-1}'))$, y_n'.

Thus, the present state variables, the next state variables and their determining variables can be divided into groups like: (y_1, $dv(y_1')$, y_1'), (y_2, $dv(y_2')\backslash dv(y_1')$, y_2'), ..., (y_n, $dv(y_n')\backslash(dv(y_1') \vee ....\vee dv(y_{n-1}'))$, y_n'). However, since some of the determining variables of the next state variables may be included in other groups, we add a *close-detection* procedure between two adjacent next state variables. For example, if $dv(y_1') = dv(y_2')$, then y_1' and y_2' are very close. So the groups (y_1, $dv(y_1')$, y_1') (y_2, $\varnothing$, y_2') are merged into one group. In our algorithm, if $dv(y_i')$ and $dv(y_{i+1}')$ have more than half of the determining variable in common, we combine them into one group. For combined groups, we do not swap between next/present state variables. Because once two next/present variables are swapped, their present/next state variables are not in the corresponding order and this violates Constraint 3.

An actual circuit may be very complex, hence our division of the state groups may put two close variables into different groups. This results in an increase in the size

of the MDG compared with the basic sifting, although group sifting can speed up reordering greatly.

We integrated group sifting and symmetry sifting into one dynamic reordering algorithm. The algorithm includes two kinds of groups: one is formed by symmetry variables called the *symmetry group*; the other one is formed between present and next state variables called the *state group*. State groups are decided before the sifting process starts and a variable only sifts inside its group. Symmetry groups are created by the reordering algorithm, when the symmetry variables are identified. The variables in the symmetry group sift up and down together within a state group.

We have implemented our sifting-based dynamic reordering on MDG. In the next section, we present experimental results.

5.4 Experimental Results

We applied the dynamic reordering algorithm to the benchmarks we used in the last chapter. We started our dynamic reordering algorithm from the order generated by the static ordering algorithm presented in the last chapter. We compare the results from our algorithm with those from the basic sifting algorithm and also with those only from the static order. Table 5.1 shows the experimental results for the IFIP benchmarks. Experimental results on the ATM switch fabric are shown in Table 5.2. Table 5.3 shows the results for the Island Tunnel Controller.

circuit	basic sifting		our algorithm		static order	
	nodes	time(sec)	nodes	time(sec)	nodes	time(sec)
minmax	164	1.22	164	0.78	166	0.16
tlc	245	10.95	228	10.41	245	1.22
gcd	369	4.26	369	4.24	375	0.26
tama	2136	17.28	2150	9.97	2211	1.44
filter	2483	30.44	2485	16.98	2957	4.08
queue	6432	24.71	6438	22.65	6551	3.58
buffer	2307	33.15	2307	4.43	2307	2.21

Table 5.1 Experimental results for IFIP benchmark circuits

property	basic sifting		our algorithm		static order	
	nodes	time(sec)	nodes	time(sec)	nodes	time(sec)
P1	12135	640.09	12713	311.72	19572	42.39
P2	11582	646.97	12084	329.95	19633	40.69
P3	9649	833.39	10035	458.36	22249	52.05
P4	11828	583.77	13166	279.53	19601	38.08

Table 5.2 Experimental results of property checking on the ATM model

Counter width	basic sifting		our algorithm		static order	
	nodes	time(sec)	nodes	time(sec)	nodes	time(sec)
Abstract	2150	39.60	2316	25.96	3244	2.36
4 bits	2489	112.32	2765	60.44	7687	8.33
5 bits	4326	143.74	4452	92.47	8945	10.44
6 bits	6132	189.67	6326	114.52	12275	17.76
7 bits	6885	446.33	7224	272.25	18931	36.62
8 bits	10265	1422.87	11342	672.32	32243	89.51
9 bits	18287	2458.43	19103	1321.55	-	-
10 bits	29926	5327.44	30634	3237.29	-	-

Table 5.3 Experimental results for invariant checking on ITC

The experimental results in Table 5.1 show that the basic sifting and our reordering algorithm can reduce the size of MDGs for most of benchmarks we experimented with. Table 5.2 and 5.3 show that the basic sifting can reduce the size of MDGs (obtained from the static order) from 33% to 68% with the increase of computing time up to 16.8 times. Our reordering algorithm can reduce the size of MDGs (obtained from the static order) from 28% to 65% with the increase of computing time up to 11.0 times. Compared to the basic sifting algorithm, our reordering algorithm results in about 44.1% time improvement and costs less than 5.4% size increase on the average. In the ITC example, both the basic sifting and our algorithm can verify the counter width until 10 bits whereas the static method can only verify it up to 8 bits. Although reordering increases the computing time of verification compared with static ordering methods, it can greatly reduce the size of MDGs. It is particularly useful for verification of circuits where insufficient memory size is the bottleneck.

Summary

In this chapter a dynamic variable ordering algorithm on MDG was presented. Some MDG applications may fail with a fixed order. However, the size of MDGs can be minimized with a new order chosen in the middle of verification. The experiments show that dynamic reordering reduces the size of MDGs effectively and increase the range of circuits that can be verified by MDGs. In the next chapter, we will present a problem caused by standard term ordering and a solution based on function renaming and rewriting.

Chapter 6 Standard Term Ordering Problem

Although selecting a good static variable order and changing it dynamically as the application proceeds can minimize the size of the MDG, there are still cases where under any order the size of the MDG is exponential with the number of function instances. These cases may be caused by the standard order adopted in MDG to order cross-terms with the same cross-operator. We call this kind of state explosion the *standard term ordering problem.*

In Section 6.1, we introduce a kind of circuit structure that leads to the standard term ordering problem in MDG-based verification. We propose a solution based on function renaming and unconditional cross-term rewriting in Section 6.2. In Section 6.3, we present a case study on an ATM congestion controller using our solution. This chapter is concluded with discussions of the limitation of our solution and future research.

6.1 Introduction to the Standard Term Ordering Problem

According to one of MDG's well-formedness conditions, the variables and the cross-operators of the cross-terms that label the nodes must appear in a custom symbol order along each path of an MDG. For different cross-terms with the same cross-operator, they must appear in the standard term order. In the MDG system, the standard term order is the lexicographical order. This does not mean that the standard term order has no influence on the size of the MDG. In fact, it may result in the state explosion for certain kinds of circuits. Before introducing the standard term ordering problem, we first describe the internal representation of terms and cross-terms, which form the decision factors for the standard term order.

6.1.1 Internal Representation of Terms and TermID Assignment

A term in an MDG is defined as follows [70]:

- A (individual or generic) constant is a term.
- A (concrete or abstract) variable is a term.
- If $a_1, \ldots, a_n$ are terms and f is a function symbol, then $f(a_1, \ldots, a_n)$ is a term, which is also referred to as a compound term.

In an MDG, there may be many very large compound terms. Thus, it is important to have compound terms share some common components. In order to achieve term sharing, MDG system assigns a unique identifier (TermID) to each term. The TermID for a constant or a variable is the constant or the variable itself. The TermID for a compound term is a number obtained by hashing the function symbol using a Quintus Prolog built-in hash function. It is assigned to a compound term when the term is first generated in the verification process.

For example, in Figure 6.1, root node n_1 and its labeling term $eq(x_1, x_2)$ is generated first. The TermID of $eq(x_1, x_2)$ is computed by hashing the function symbol eq. Suppose hash(eq) = k, the TermID of $eq(x_1, x_2)$ would be k. Since the MDG is built in a depth-first order, the left-hand branch is constructed first. Thus, nodes n_2 and its labeling term $eq(y_1, y_2)$ are generated next. To compute the TermID for $eq(y_1, y_2)$, we hash again the function symbol eq. In Quintus Prolog, hashing collision is resolved by linear probing, i.e., search the hash table sequentially, starting from the original hash location [77]. Suppose $k + 1$, $k + 2$ ... are all available, hashing eq again would get $k + 1$ and thus the TermID of $eq(y_1, y_2)$ would be $k + 1$. Nodes n_3 and n_4 are generated next in the depth-first order. Similarly, the TermIDs of their labeling compound terms $eq(y_3, y_4)$ and $eq(z_1, z_2)$ are $k + 2$ and $k + 3$ if there are no other new cross-terms in the branches of nodes n_2 and n_3. From this example, it is easy to see that the TermIDs of cross-terms with the same operator are assigned sequentially according to the order of the

terms constructed in the MDG. The first generated one has the smallest TermID (original hash location). The rest has TermIDs sequentially starting from the original location.

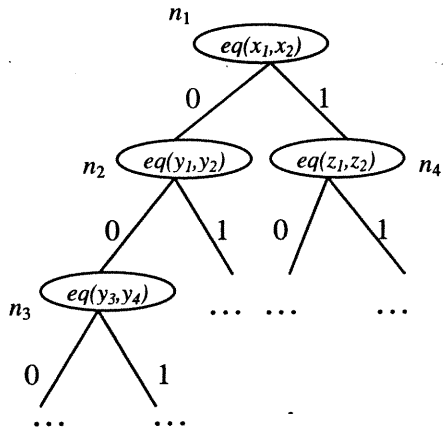


Figure 6.1 TermID assignment for cross-terms with the same function symbol

The internal representation of a compound term is *term(Function_symbol, TermID, Subterms)*. For instance, $f(a_i, f(a_j))$ is represented as *term(f, Id*1, [*term(f, Id*2, [])]). Standard order of the cross-terms with the same cross-operators is the lexicographical order of their internal representation. Since those cross-terms have the same function symbol (no matter if they have Subterms), their order is thus decided by the value of their TermIDs. In other words, standard order of the cross-terms with the same function symbol is the same order as when the terms are first generated during the construction of the MDG.

6.1.2 Identification of the Standard Term Ordering Problem

Consider the circuit shown in Figure 6.2. Function f is a cross-function with two arguments c_i and a_i, where c_i is a Boolean variable and a_i is an abstract variable.

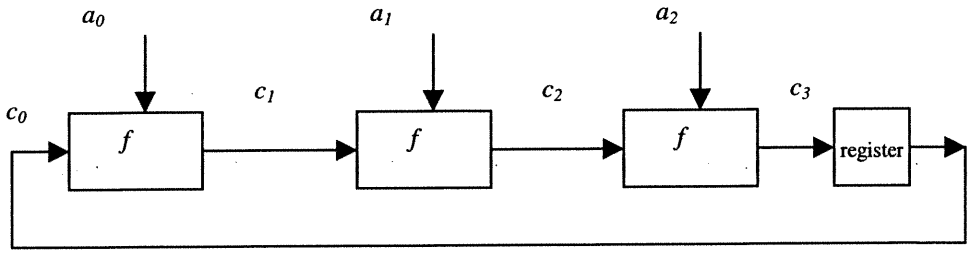


Figure 6.2 A circuit with standard term ordering problem

The MDG representation of the new states in the first transition step of reachability analysis is shown in Figure 6.3. $f(b, a_i)$ $0 \leq i \leq 2$ is a cross-term where b is either 0 or 1. $f(b, a_i)$ is a compound term. The leftmost branch is built first and the order of cross-terms is thus as follows: $f(0, a_0) < f(0, a_1) < f(0, a_2) < f(1, a_2)$. When the system starts to build the next left-hand branch (in gray), $f(0, a_2)$ and $f(1, a_2)$ have been built before and thus have smaller TermID than $f(1, a_1)$, which is newly generated. The left-hand branch is thus constructed under the order $f(0, a_0) < f(0, a_2) < f(1, a_2) < f(1, a_1)$. Note that no node sharing is possible for this branch under this order. The number of paths that the MDG has is thus exponential with the number of the function instances as it decodes all possible combinations of the values of the first argument of f.

The exponential number of paths in the MDG increases the execution time of reachability analysis. In the reachability analysis procedure, the Pruning-by-Subsumption (PbyS) operation is used to simplify the set of reachable states found so far by removing from it any paths that are subsumed by the frontier set [69]. The large number of paths in the MDG representing the set of reachable states raises the time and space needed by the PbyS operation. In this example, with the increasing number of function instances, PbyS operations could not terminate because of insufficient memory when it reached 4 instances.

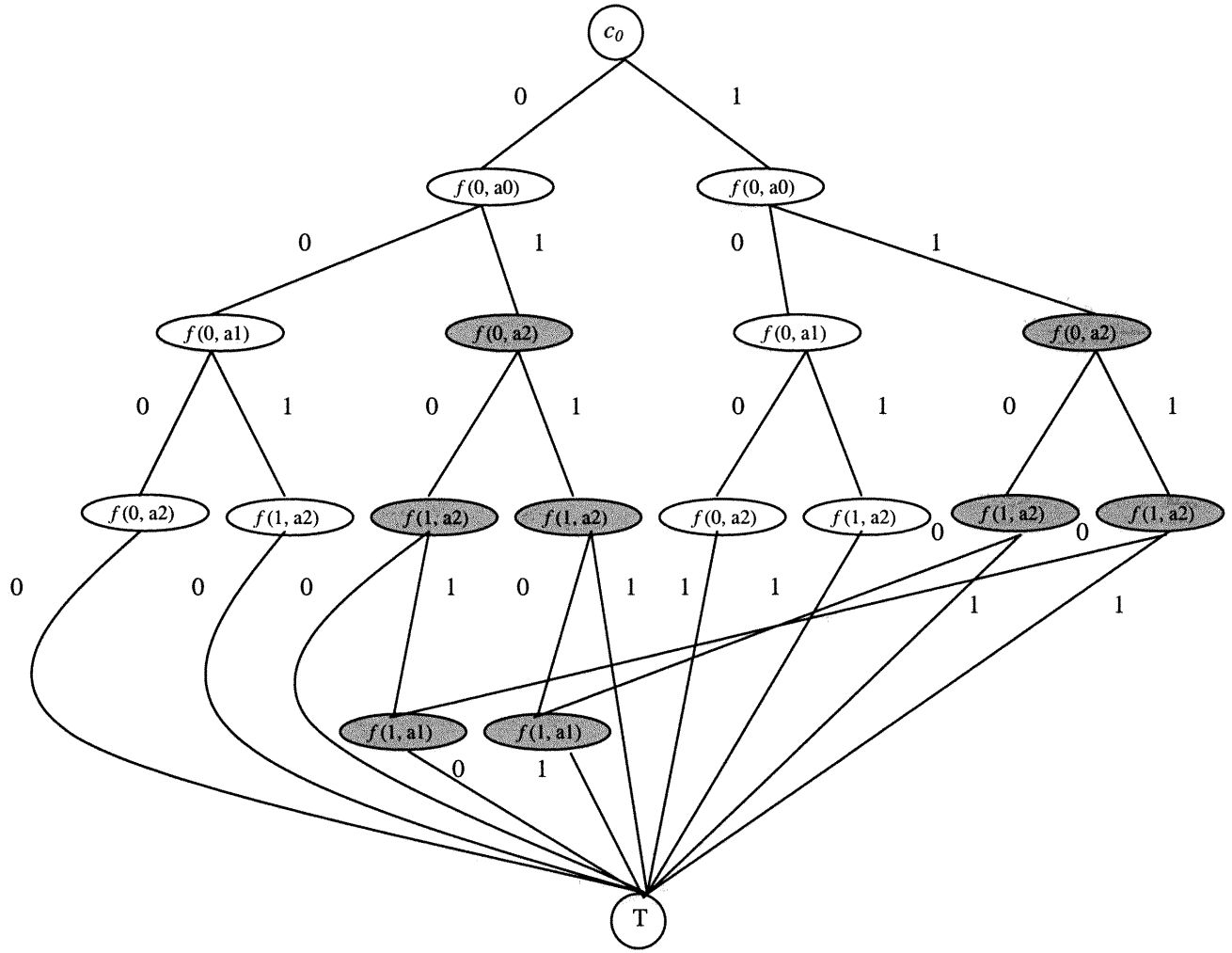


Figure 6.3 The MDG of new states in the first transition step of reachability analysis of the circuit

6.1.3 A Chain Circuit Structure with the Standard Term Ordering Problem

In Figure 6.1, the output of each function instance is the input to the next function instance, but the standard term ordering decides that the cross-terms cannot be in the order leading to a linear size MDG. However, there is no standard term ordering problem in the circuits containing the cross-function instances in the parallel structure shown in Figure 6.4. Cross-functions 2, 3 and 4 do not depend on each other, thus the cross-terms of those instances are quite independent and their positions in the order do not have strong effect on each other. The MDG of this circuit can be of linear size regardless the order the cross-function instances 2, 3, and 4 take.

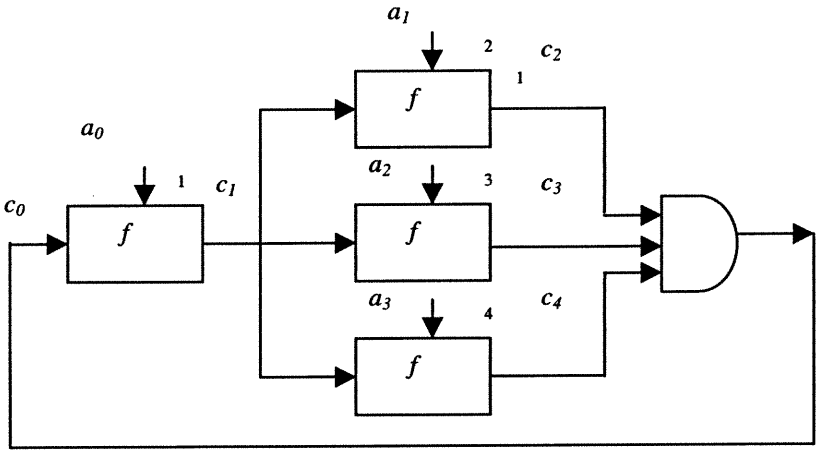


Figure 6.4 Parallel structure of cross-functions

We can thus say that the standard term ordering problem only happens to a circuit containing a chain of the same cross-functions as shown in Figure 6.5. The output of a function instance is the direct/indirect input of the next function instance. The circuit can have pipeline registers (or multiplexers) between function instances. Experiments show that the verification of the circuits with this kind of a chain structure would require rapidly increasing amounts of memory with an increasing number of function instances.

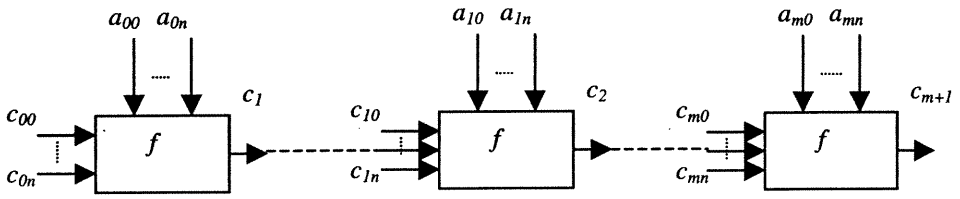


Figure 6.5 A common circuit structure resulting standard term ordering problem

Since the standard term ordering problem is caused by one of MDG well-formedness conditions, we are not able to solve this problem using the usual ordering strategies. In the custom order, we only consider the order of cross-functions. The cross-terms are created dynamically in the verification process and no static ordering or reordering procedure is possible at this stage. In the next section, we present a solution to this problem by integrating function renaming and cross-term rewriting.

6.2 A Solution to the Standard Term Ordering Problem

A solution to the standard term ordering problem is to order the cross-terms with the same cross-operators to make the corresponding MDG linear in size. The order of those cross-terms is decided by the lexicographical order of their MDG internal representations, hence we can only change the order by renaming the function instances, thereby allowing them to be freely placed within the custom order. Rewriting rules have to be used to map the renamed functions back to their original functions during the PbyS operation. We will describe next our solution in detail.

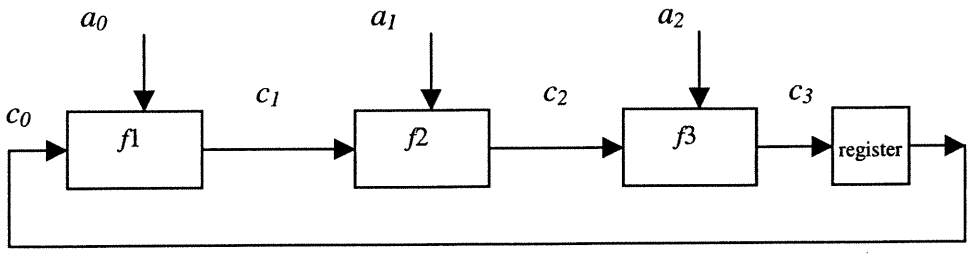
6.2.1 Function Renaming

After identifying the chain structure (Figure 6.5) in a circuit, we can rename all the cross-functions instances in the chain to different names. For instance, for the circuit shown in Figure 6.2, we rename the three instances of f to $f1$, $f2$, $f3$ and impose the order $f1 < f2 < f3$. The circuit and the MDG of new states in the first transition step are shown in Figure 6.6. The MDG becomes linear in size because of the sharing of subgraphs from the second level.

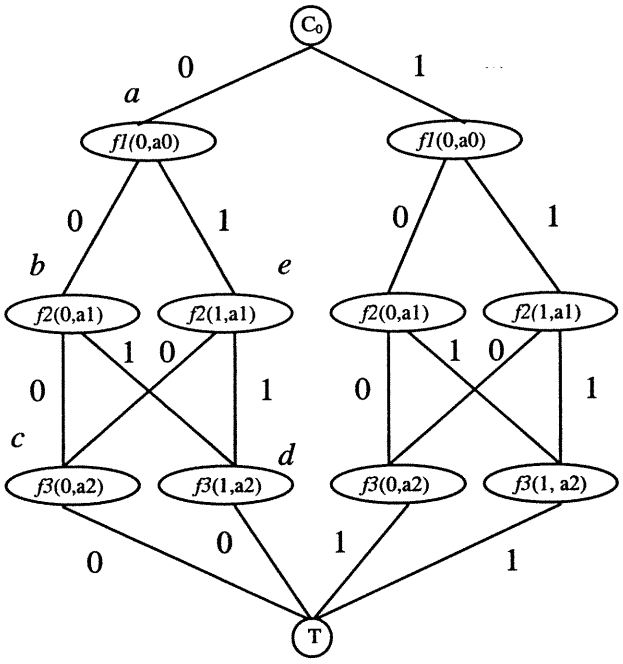
As explained in Chapter 4, the order of an output should come after all its inputs. Thus, after renaming all the functions in the chain structure, a static order for the renamed functions is chosen according to the positions of the functions appearing in the chain structure, i.e., from inputs to outputs. In this example (Figure 6.6), the order $f1 < f2 < f3$ is imposed. The resulting MDG is 26% smaller than using the reverse order.

For the circuit which contains more than one chain, we first choose the longest chain and rename all the functions in the chain. Since renaming may break some other chains, the remaining chains have to be re-inspected and the chains that still exist have to be renamed.

Function renaming solves the standard term ordering problem. However, it brings out a new problem: the actual function of the circuit has changed and the circuit is thus more general than needed. To solve this new problem, we introduce cross-term rewriting. With cross-term rewriting, the MDG system can rewrite the renamed functions back to the original function when an actual comparison is being done and the verification result is not changed because of function renaming.



(a) Circuit



(b) The MDG of new states in the first transition step

Figure 6.6 The MDG after function renaming

6.2.2 An Unconditional Cross-term Rewriting System

A rewriting system was first used in MDG to partially interpret an uninterpreted function symbol [69]. A data operation is represented by an uninterpreted function symbol. However, function symbols need to be partially interpreted in many designs where optimization is used. For example, a multiplier can be bypassed

when one of the operands is 1. To verify the designs including such bypassing logic, we need to use the fact that 1 is the unit element of multiplication. One effective way to reason about the partially interpreted function symbols is term rewriting. For this multiplication example, the algebraic equations $1 * x = x$ and $x * 1 = x$ could be used as rewrite rules. Rewrite rules for cross-terms can be used to shrink the MDG size on the fly. For example, if there is a path in an MDG which has a cross-term $eq(x, x) = 0$ where eq stands for equality, we could use the rewrite rule $eq(x, x) \rightarrow 1$ to eliminate this path. The scope of MDG-based verification can thus be extended.

In the original rewriting system, a cross-term cannot be rewritten into another cross-term having a different cross-operator [69]. This is to avoid the necessity of node reordering after rewriting. We thus introduce an unconditional cross-term rewriting system, as follows:

Definition 6.1: An unconditional cross-term rewriting system (CTRS) H is a finite set of formulas, called rewrite rules, having the following form:

$$\text{LHS} \rightarrow \text{RHS}$$

where LHS and RHS are cross-terms. (LHS/RHS stands for the left/right-hand side.)

When a cross-term is matched with the LHS of a rule, LHS will be substituted by RHS unconditionally. For instance, in Figure 6.6, a set of rewrite rules can be defined as follows:

$$f1(x, y) \rightarrow f(x, y)$$
$$f2(x, y) \rightarrow f(x, y)$$
$$f3(x, y) \rightarrow f(x, y)$$

So the cross-terms in Figure 6.6(b) are substituted under the following rules:

$$f1(0, a_0) \rightarrow f(0, a_0) \quad f1(1, a_0) \rightarrow f(1, a_0)$$
$$f2(0, a_1) \rightarrow f(0, a_1) \quad f2(1, a_1) \rightarrow f(1, a_1)$$

$$f3(0, a_2) \rightarrow f(0, a_2) \qquad f3(1, a_2) \rightarrow f(1, a_2)$$

The rewriting system rewrites the cross-terms in an MDG when a comparison is needed in the PbyS operation. We show how the rewriting system works in PbyS by computing the frontier set in the second transition step for the circuit in Figure 6.6(a).

In Figure 6.7, N_2 represents the set of states that can be reached in the second step from the last frontier set and R_1 represents all the states reached in the first transition step. The new frontier set Q_2 is computed using PbyS by removing from N_2 the paths representing states that are already in R_1.

When the PbyS operation starts pruning the leftmost path π of N_2 (shown in gray), it compares each node and edge along this path with those of the path π' of R_1 (also shown in gray). When matching each node, the rewriting system changes the renamed operators $f1, f2, f3$ to the original operator f. It is easy to see that the path π of N_2 can be subsumed by the path π' of R_1 after substituting $a_i\#1$ for $a_i\#2$, $1 \leq i \leq 3$.

Since every path in N_2 can be subsumed by R_1, the new frontier set Q_2 is empty. It means that the fixpoint has been reached, and the reachability analysis procedure terminates and reports success.

We use function renaming to build an MDG with a good order generated by our variable ordering algorithm, then use rewriting to restore the original function of circuits without changing the size of the MDG. We have implemented our solution in the MDG package. Our method can automatically detect a chain structure in the circuit. When the chain contains more than 3 functions, those functions are renamed and rewriting rules are used later during PbyS comparisons to rewrite them back to the original function symbol.

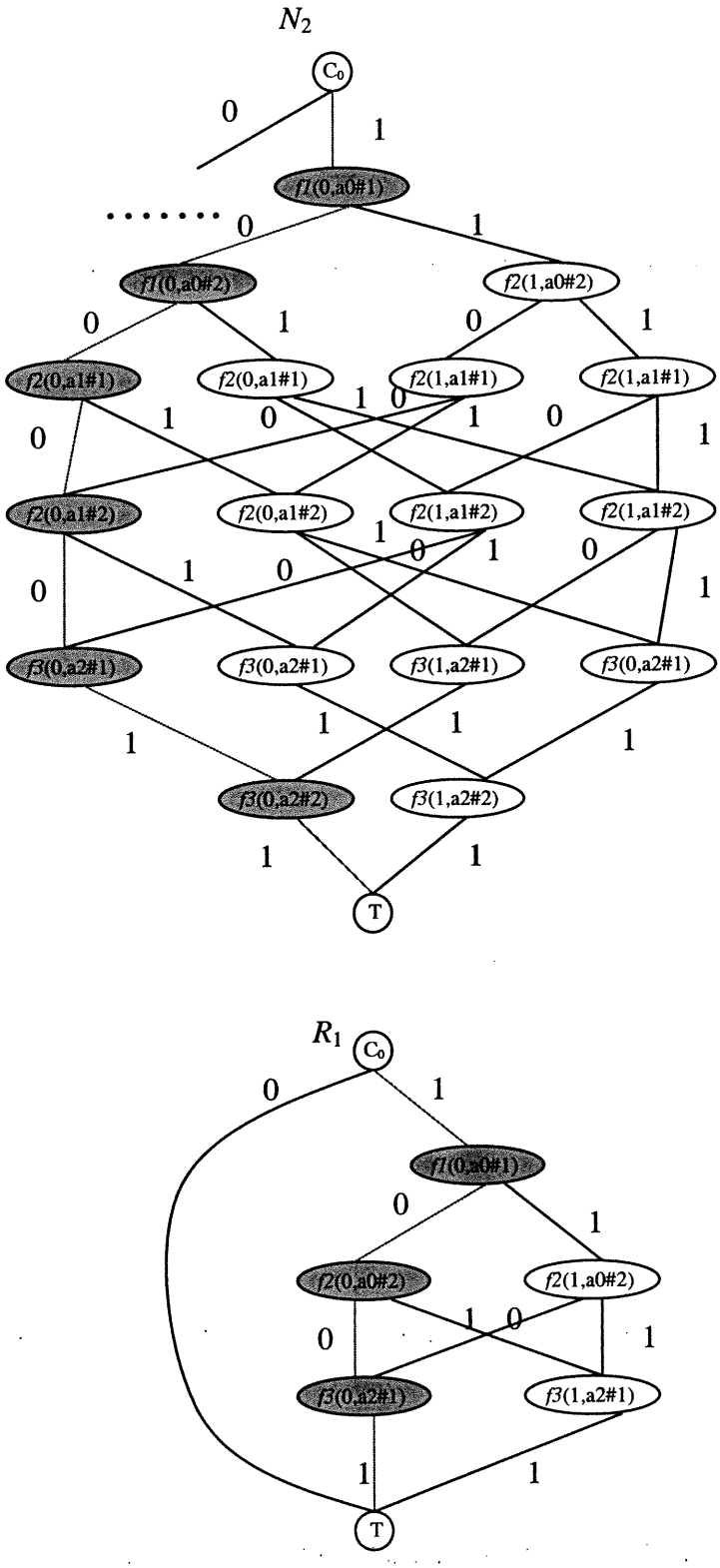


Figure 6.7 An example of the PbyS operation to compute the new frontier set

We extend our renaming and rewriting method to the parallel structure as shown in Figure 6.4 and further to all the circuits containing cross-operators which have more than two instances placed irregularly. Although the standard term ordering problem may not happen in these two kinds of circuits, the instances of the same cross-operator have different positions in the circuit and thus renaming of the instances allows them to have different positions in the custom order. Our method provides more flexibility in variable ordering. As different function symbol, each instance can now be freely placed in the custom order. The MDG can thus achieve better sub-graphs sharing and reduce its size and execution time.

6.3 A Case Study

We introduce in this section an ATM congestion controller [44] which leads to the standard term ordering problem in MDG-based verification. In an ATM switch, during the periods of heavy traffic within the network, an outgoing link may become temporarily overloaded and data packets (cells) begin to build up in the outgoing queue. This is known as congestion.

A congestion controller is used to solve congestion by comparing the priorities of two packets. The packets with higher priority will be passed, the other ones will be discarded. This does not mean that the discarded messages would be lost. The packets can be resent by the senders when they do not receive acknowledgement from the receivers, using higher-level protocols.

The controller is shown in Figure 6.8. a_0, b_0 are abstract variables representing information packets to be sent. The cross-function *compare* is used to compare the priorities of two packets. Multiplexers are used to transfer the packets with high priority. When a circuit contains more than four cross-functions, the standard ordering problem occurs.

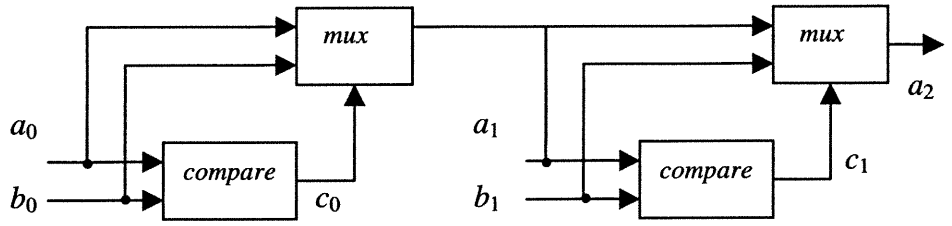


Figure 6.8 An ATM congestion controller

We implemented our algorithm in the reachability analysis procedure and the result of applying it to the circuit in Figure 6.7 is shown in Table 6.1. The experiments were carried out on a 333MHz Sun Ultra 10 workstation with 1GB of memory. The table shows that without our standard term ordering problem, the controller is only limited to four blocks. With our solution, the reachability analysis can be done up to more than 320 blocks.

No. of Functions	Without the solution		With the solution	
	Nodes	Time(sec)	Nodes	Time(sec)
2	129	0.40	163	0.68
3	273	1.56	329	0.79
4	541	378.02	507	0.98
5	-	-	697	1.20
6	-	-	910	1.36
7	-	-	1113	1.72
10	-	-	1402	2.12
100	-	-	25880	93.21
200	-	-	81330	883.68
300	-	-	166780	4236.64
320	-	-	187470	6226.46

Table 6.1 Experimental results for an ATM congestion controller

Summary

In this chapter, we introduced a special problem caused by the standard term ordering that MDGs use to order cross-terms with the same function symbols. Since this problem is the result of the MDG well-formedness conditions, it cannot be solved by ordering/reordering variables. We presented a solution integrating function renaming and cross-term rewriting. Experimental results on a congestion controller show that our solution can considerably improve the performance and increase the type and size of circuits that can be verified.

In the preceding chapters, we discussed the static variable ordering, the dynamic variable ordering, and the standard term ordering problems in MDG, and we proposed solutions. We integrated the variable ordering algorithms with the MDG design verification system for circuits described in MDG-HDL. However, in industry, most of designs are described in VHDL or Verilog. The MDG-based verification system needs a standard HDL entry to be easier to use. In the next chapter, we concentrate on the other part of our work - automatic translation between VHDL and MDG-HDL.

Chapter 7 Translation from VHDL to MDG-HDL

Variable ordering methods increased the range of circuits that can be verified using MDGs. However, the MDG-based verification system only accepts a Prolog-style HDL, MDG-HDL. In this chapter, we present another technique that improves the ability of MDG to deal with the realistic systems: an automatic translator from VHDL to MDG-HDL. The translator accepts a VHDL model given at the synthesizable Register Transfer Level and translates it to a Directed Acyclic Graph (DAG) which describes the control and data flow in one clock period. MDG-HDL is then generated from the DAG.

In Section 7.1, we introduce the construction of the Directed Acyclic Graph from a VHDL model given at the synthesizable RTL level using only one clock and clock edge. The construction was developed by Boubezari [4]. Translation from DAG to MDG-HDL is presented in Section 7.2. We conclude this chapter with two examples.

7.1 Transformation from VHDL to a DAG

VHDL is a hardware description language used to describe both the structure of a circuit (what parts constitute it and how they are connected) and its behavior (how it reacts to given inputs). It is an international interoperability standard for design automation – in other words, a formal mechanism to "converse" with design automation tools.

A Directed Acyclic Graph (DAG) to be translated into is a representation of the flow of control and data dependencies in one clock cycle [4]. Each internal node of

the DAG corresponds to an operation of the VHDL model such as arithmetic, relational, data transfer and logical operations. The source (sink) nodes of the DAG represent the present (next) state and primary inputs (outputs). The present and the next state variables are given by the registers that are identified in the VHDL model. Edges represent interconnection signals/variables in the VHDL model and intermediate signals/variables as defined in Definition 7.1. Note that no hardware sharing is performed during the VHDL translation in the DAG. That means, each VHDL operation corresponds to a new node in the DAG.

Definition 7.1: An intermediate signal/variable is an unnamed signal/variable formed by an expression which is not a simple signal/variable name. For example, in the VHDL model shown in Figure 7.1(a), two intermediate signals (s_int1 and s_int2) and two intermediate variables (v_int1 and v_int2) are implied as shown in Figure 7.1(b).

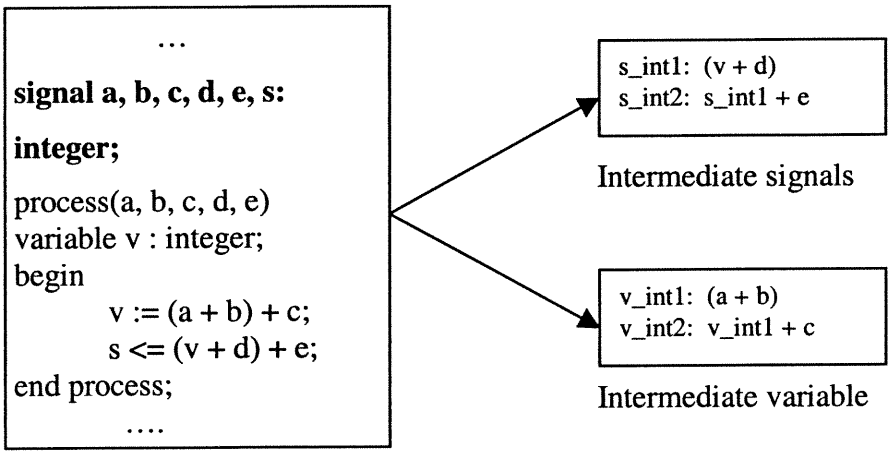


Figure 7.1 An example of definition of intermediate signals/variables

In the rest of this thesis, whenever a signal/variable is mentioned, it refers to an interconnection signal/variable declared by the designer in the VHDL model or to

an intermediate signal/variable as stated in Definition 7.1. Below, we summarize the main steps in constructing a DAG of a VHDL model:

1. Generating Control and Data Flow Graph (CDFG) for each process of the VHDL model.
2. Unrolling all for...loops and expanding procedures/functions by adding new nodes to the CDFG.
3. Identifying multiplexers
4. Identifying registers
5. Translating the resulting CDFG into a DAG.
6. Connecting DAG graphs of individual processes to produce the overall DAG.

Each step is described in detail next.

7.1.1 Generation of CDFG

Each VHDL process can be transformed into a Control Flow Graph (CFG) and a Data Flow Graph by the LEDA Graph Generator tool to represent the control flow of operations [74].

Control flow graphs are used to represent the different possible sequences of execution for each set of statements in a VHDL process. A CFG is a directed graph defined as follows:

$$CFG = (V, E), \text{ where}$$

V is a set of nodes corresponding to the different VHDL sequential statements:

$$V = V_W \cup V_b \cup V_l \cup V_e, \text{ where}$$

V_w is the set of synchronization nodes (wait statements),

V_b is the set of conditional statement nodes (if, case),

V_l is the set of for...loop statement nodes,

V_e is the set of other nodes (signal/variable assignments, procedure calls,...).

and E is the set of edges representing the flow of control.

Each node of a CFG is associated with a Data Flow Graph (DFG). Each node of DFG represents one operation in the VHDL model, and a DFG edge represents signals and variables connecting the nodes. There is an edge from operation o_i to operation o_j if the result of operation o_i is input to operation o_j. A CFG in which each node is a DFG is called a Control and Data Flow Graph (CDFG) [74].

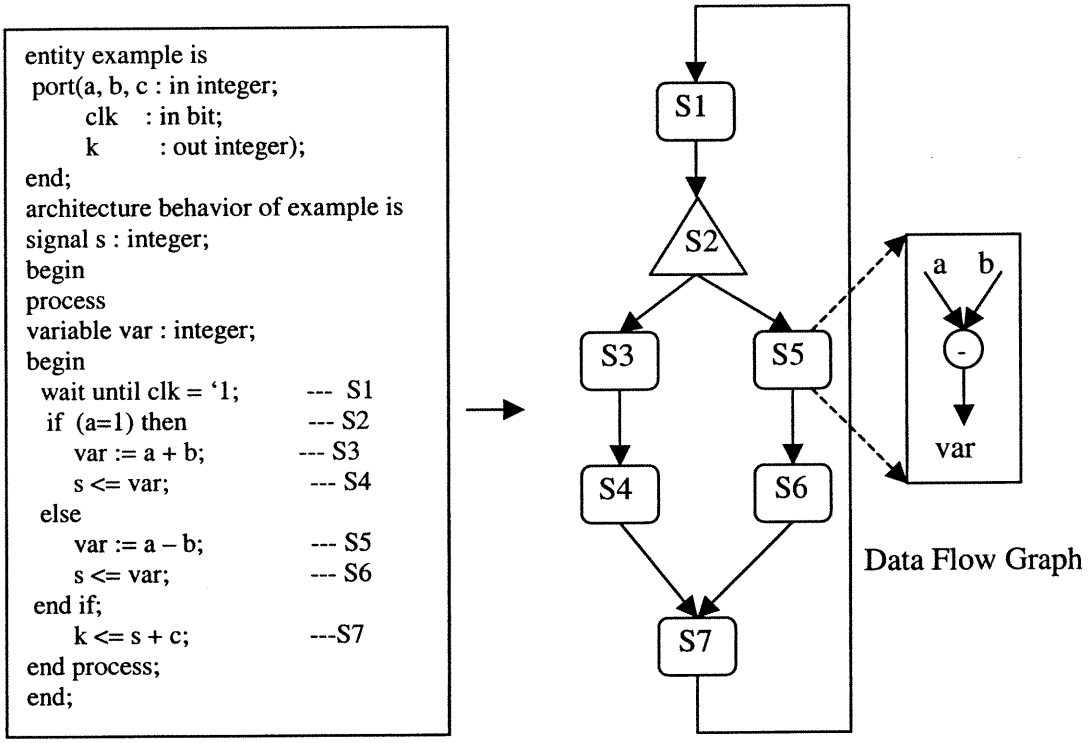


Figure 7.2 An example of a CDFG

Figure 7.2 shows a VHDL model and its corresponding CDFG generated by LEDA [73]. Node S1 represents the "wait" statement and node S2 corresponds to the conditional "if" statement. Signal/variable assignments are represented by nodes S4, S6 and S7 (S3 and S5). Each node of CDFG is associated with a Data Flow Graph describing the data dependencies in the statement. An example of a DFG is given for node S5.

7.1.2 Loop Unrolling and Expansion of Procedures/Functions

In VHDL, a for...loop statement is used to repeat a sequence of operations for a constant number of times. Since there is no hardware sharing in a DAG, a for...loop statement would be unrolled into the statements inside the loop, one for each iteration. Each procedure call is expanded in-line. The contents of the procedure are first copied into the process in place of the call. Then, the actual parameters are substituted for the formal parameters of the procedure. Functions are expanded in a similar way except that a function is expanded immediately before the expression that calls it. As a result of loop unrolling and expansion of procedures/functions, the CDFG is augmented by new nodes.

7.1.3 Identification of Multiplexers

We give a translation for the VHDL conditional statements ("if" and "case"). A new node type is added to the DAG representing the multiplexer operation. Each conditional statement ("if" or "case") is translated into a set of mulitplexers with one output for each signal/variable assigned within the conditional statement. The condition expression is translated into a set of nodes which feed the control inputs of each multiplexer. The data inputs to each multiplexer are fed from the nodes of the corresponding expression being assigned. Thus, to translate each conditional statement to a multiplexer operations, we have to find its scope, i.e., its corresponding end-statement.

7.1.4 Identification of Synthesized Registers

Registers in a synthesized RTL VHDL model are inferred on some signals/variables used in clocked processes. We thus have to first identify clocked

processes in the VHDL model. In the following, we give some rules that determine when registers are inferred on signals and variables.

1. All signals that are assigned new values within a clocked process are synthesized as outputs of registers.

2. Between two synchronization statements (wait on clock), if a variable is always the target of an assignment before being read, this variable does not infer any register.

3. Between two synchronization statements, if a variable is read at least once before appearing as the target of an assignment, this variable implies storage of data and thus a register is synthesized.

Figure 7.3 shows an example of rule 1 and 2. *q* is assigned a new value within a clocked process and thus is synthesized as a register. The variable *var* is always the target of an assignment before being read. Therefore, no register is required to memorized its value from one clock cycle to the next. *var* is thus synthesized as a wire.

```
...
process
variable var : bit;
begin
  wait until clk= '1';
  var := a and b;
  q <= var and c;
end process.
...
```

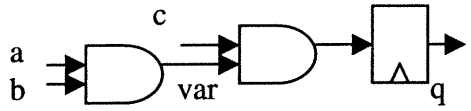


Figure 7.3 Illustration of rule 1 and 2

An example in which a variable in a clocked process is synthesized as a register, is shown in Figure 7.4. Between two synchronization statements, the variable *var* in the VHDL model is read at least once before appearing as the target of an assignment. Therefore, a register is needed to store the value of *var*. Note that in

this example, reading a variable does not necessarily mean having it in the right hand side of an assignment. A "case" expression, an "if" condition or an "in" parameter of a procedure or a function are other ways of reading.

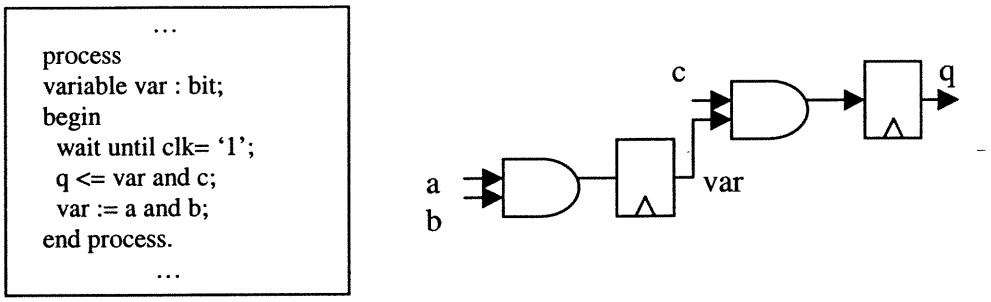


Figure 7.4 Illustration of rule 3

7.1.5 Translation of a CDFG into a DAG

The next step is to translate the resulting CDFG into a DAG. Synchronous registers are inferred on signals and some variables assigned in a clocked process. Once we separate the present and the next states of registers into separate nodes, we obtain a DAG. The source nodes are the primary inputs and present state values of registers and the sink nodes are the primary outputs and the next-state values. Pseudo-primary inputs (outputs) correspond to synthesized outputs (inputs). The algorithm to identify the synthesized registers in the VHDL model is the same as used by most synthesis tools.

7.1.6 Connecting Processes and Component Instances

A VHDL model consists of a set of interacting processes (clocked and unclocked). Entity ports and internal signals (declared in the VHDL architecture) are used to

communicate between the processes. The connection between processes is represented as a Directed Graph. However, we have seen that each VHDL process can be transformed into a CDFG which is represented as a DAG. Hence, the global VHDL model is also represented as a DAG.

In VHDL, component statements are virtual design entities and are instantiated in DAG construction like functions. First, the signals listed in the component instantiations substitute the ports listed in the component declarations. Then, the DAGs for the components are inserted right before the statements that call it.

We illustrate the overall construction of a DAG by presenting two examples in Section 7.3. The generation of MDG-HDL from DAG is discussed next.

7.2 Translation from a DAG to a MDG-HDL Model

Hardware modeling can be broadly divided into two styles: structural and behavioral. Structural modeling implicitly defines the input/output function by describing components and their interconnections; the input/output function can be derived from the structural implementation. Conversely, behavior modeling explicitly defines the input/output function by describing signal transformations.

MDG-HDL supports both structural and behavioral descriptions. In MDG-HDL, a structural description is usually a netlist of components (predefined in MDG-HDL) connected by signals. A behavioral description is given by a tabular representation of the transition/output relation or by a truth table [70].

When we translate a DAG of a VHDL structural model, each internal node of the DAG is translated into a predefined component module in MDG-HDL. All Boolean operations are predefined in MDG-HDL. For uninterpreted functions,

MDG-HDL provides a predefined component "transform" to define them. A multiplexer node of the DAG corresponds to the predefined component "multiplexer" in MDG-HDL.

When we translate the DAG of a VHDL behavioral RTL model, a predefined component *table* is used. The first row of a table is a list containing variables and cross-terms. Starting from the second row, each row is a list of values that the corresponding variables or cross-terms can take. To translate a DAG to the tabular representation, the select signals of multiplexer nodes are taken as the inputs of the table (the first row) and the inputs of multiplexer nodes are taken as the list of values used by the table (starting from the second row). We will explain it in more detail by an example in the next section.

One of the important issues in the translation is to decide which variables/signals should be represented by a single variable of abstract type, rather than by concrete variables. We abstract variables/signals using the following two rules:

1. The signals/variables of integer or floating point number type are taken as abstract variables and all arithmetic operations are taken as uninterpreted function symbols.

For instance, suppose an expression $c <= a + b$, where a, b, c are integers and the addition operation is an uninterpreted function symbol. The MDG-HDL representation is

component($a1$, transform(inputs([a, b]), function(*add*), output(c))),

where $a1$ is the name of the instance of predefined component "transform".

2. A constrained array can be abstracted only when no single element of the array is used separately in the VHDL model or the properties to be verified, i.e., the array is only used as a whole. Otherwise, each element of the array is represented by a separate (concrete/abstract) variable.

Figure 7.5 shows an example of array abstraction. Arrays *a*, *b* and *o1* will be abstracted in MDG-HDL because no individual element of these arrays is manipulated by the VHDL model. Arrays *c* and *o2* can not be abstracted because their elements have been used separately. Each element of *c* and *o2* will be represented as a Boolean variable. Thus, the loop statement turns into

$$o2(0) <= c(0) \text{ and } d(0)$$
$$o2(1) <= c(1) \text{ and } d(1)$$

```
entity example is
  port (a, b, c, d  : in BIT_VECTOR(1 downto 0);
        e,  clk     : in  bit;
        o3          : out bit;
        o1, o2      : out BIT_VECTOR(1 downto 0));
end example;
architecture  behavior of example is
begin
process
begin
      wait until CLK = '1';
      o1 = a and b;
      for i in 0 to 1 loop
         o2(i) <= c(i) and d(i)
      o3 <= c(0) or e;
end process;
end behavior;
```

Figure 7.5 An example of array abstraction

In the next section, we give two examples of translation from VHDL to DAG and then to MDG-HDL.

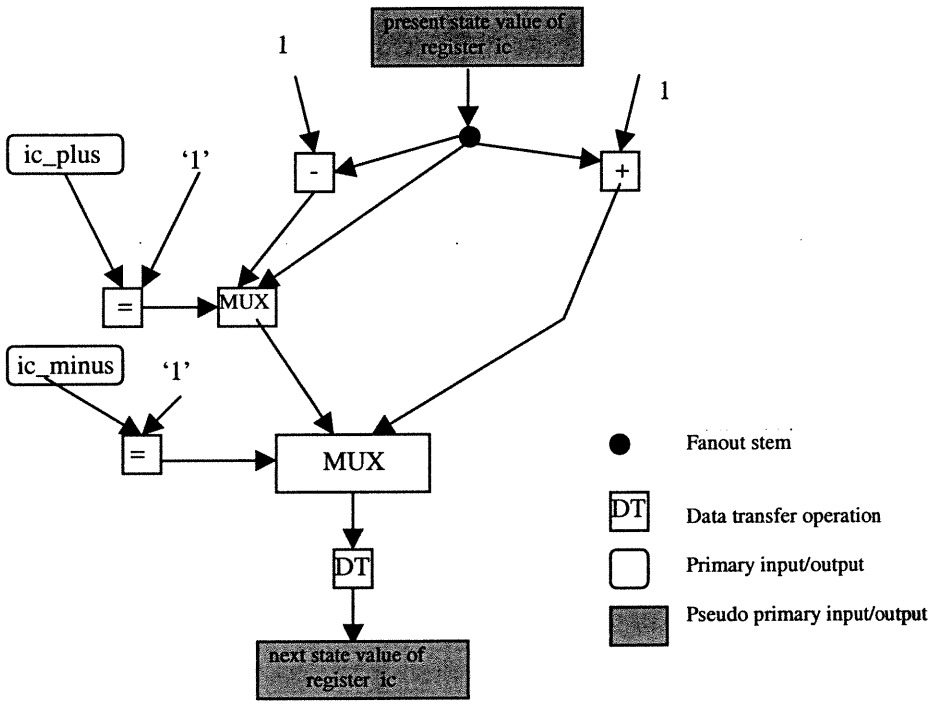
7.3 Examples of Translation from VHDL to MDG-HDL

7.3.1 Island Tunnel Control Counter

Consider the ITC counter example we discussed in Chapter 6. The VHDL structural model is as shown in Figure 7.6. The model has only one clocked process. Signal *ic* counts the number of cars and increments/decrements depending on the signal *ic_plus* or *ic_minus*. Its DAG is shown in Figure 7.7(a), where the pseudo-primary inputs/outputs represent the present/next state values of registers. The signal *ic* is synthesized as a register. Its MDG-HDL representation is shown in Figure 7.7(b). Note that signal *ic* is abstracted as sort *wordn* and the operations increment and decrement of 1 are represented by predefined unintepreted function symbols *inc* and *dec*. *wordn* is a predefined abstract type. The multiplexer nodes are represented by the predefined component multiplexer.

```
entity itc_counter is
  port ( ic_plus,ic_minus, clk : in std_logic;
         ic  : out integer);
end itc_counter;
architecture itc_counter_asm of itc_counter is
begin
  process
    variable x : integer;
  begin
    wait until ( clk'event and clk = '1');
    if ( ic_plus = '1') then
        ic <= x + 1 ;
    else
      if (ic_minus = '1') then
        ic <= x - 1 ;
      else ic <= x;
      end if;
    end if;
  end process;
end itc_counter_fsm;
```

Figure 7.6 The VHDL model for ITC counter



(a) Directed Acyclic Graph

```
% ---------- Signals ----------
signal(x,wordn).
signal(ic1,wordn).
signal(ic2,wordn).
signal(ic_minus,bool).
signal(icm1,wordn).
signal(ic_plus,bool).
signal(icm2,wordn).
signal(ic,wordn).
% ---------- Components ----------
component(c1,reg(input(icm2), output(ic))).
component(c2,mux(sel(ic_plus),inputs([(1,ic1),(0,icm1)]),output(icm2))).
component(c3,mux(sel(ic_minus),inputs([(1,ic2),(0, x)]),output(icm1))).
component(c4,transform(inputs([x]), function(dec), output(ic2))).
component(c5,transform(inputs([x]), function(inc), output(ic1))).
% ---------- State variable, next state variable mapping ----------
st_nxst(ic,icm2).
% ---------- Initial state ----------
init_val(ic,0).
% ---------- Outputs ----------
outputs([ic]).
```

(b) MDG-HDL

Figure 7.7 The DAG and the MDG-HDL model for the ITC counter

7.3.2 A Moore Finite State Machine

Consider the VHDL model shown in Figure 7.8. It represents a Moore finite state machine with 4 states (*s*0, *s*1, *s*2, *s*3) and one output *z*. The model consists of two processes (a clocked one and a combinational one). Its DAG is shown in Figure 7.9. The signal *current_state* is synthesized as a register (a state variable). Each conditional statement corresponds to a multiplexer operation.

```
entity moore is
   port(x, clock: in BIT;
        z: out BIT);
end;
architecture behav of moore is
   type state_type is (s0, s1, s2, s3);
  signal current_state, next_state: state_type;
begin
combin: process(current_state, x)
 begin
   case CURRENT_STATE is
    when s0 =>
     z <= '0';
     if x = '0' then
       next_state <= s0;
     else
      next_state <= s2;
     end if;
    when s1 =>
     z <= '1';
     if x = '0' then
      next_state <= s0;
     else
      next_state <= s2;
     end if;
    when s2 =>
      z <= '1';
      if x = '0' then
       next_state <= s2;
      else
       next_state <= s3;
      end if;
     when s3 =>
      z <= '0';
      if x = '0' then
       next_state <= s3;
      else
       next_state <= s1;
      end if;
   end case;
end process combin;
-- Process to hold synchronous elements
(flip-flops)
 synch: process
 begin
   wait until clock = '1';
   current_state <= next_state;
 end process synch;
end behav;
```

Figure 7.8 The VHDL model of a Moore machine

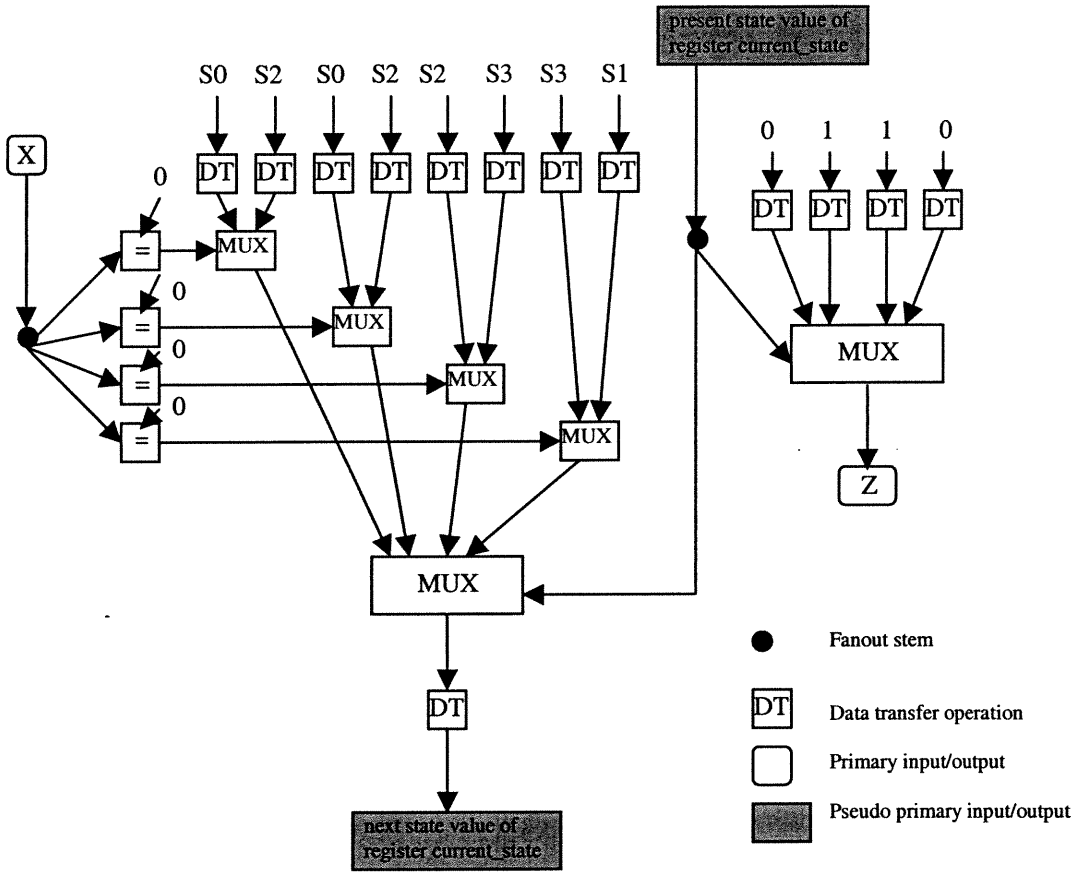


Figure 7.9 Directed Acyclic Graph for the Moore machine

The MDG-HDL representation of the Moore machine is shown in Figure 7.10. *current_state* and *next_state* are defined as signals of concrete sort *state_type*. *state-type* is not a predefined concrete type, it thus needs to be defined as follows:

conc_sort(*state_type*, [*s*0, *s*1, *s*2, *s*3]).

Since the Moore machine is given as a behavior model, its MDG-HDL representation is presented in a tabular form. *current_state* and *x* are taken as the inputs of table *c*2, where *next_state* is the output. The combination of values of *current_state* and *x* decides the output of *next_state*.

```
% % ---------- Signals ------------
signal(current_state, state_type).
signal(z, bool).
signal(x, bool).
signal(next_state, state_type).
% ---------- Components ------------
component(c1, reg(input(next_state), output(current_state))).
component(c2, table([[current_state, x, next_state],
                     [s0, 0, s0],
                     [s0, 1, s2],
                     [s1, 0, s0],
                     [s1, 1, s2],
                     [s2, 0, s2],
                     [s2, 1, s3],
                     [s3, 0, s3],
                     [s3, 1, s1]])).
component(c3, table([[current_state, z],
                     [s0, 0],
                     [s1, 1],
                     [s2, 1],
                     [s3, 0]])).
% ---------- State variable, next state variable mapping ----
st_nxst(current_state, next_state).
% ---------- Initial state ------------
init_val(current_state, s0).
% ---------- Outputs ------------
outputs([z]).
```

Figure 7.10 MDG-HDL model of the Moore machine

Summary

In this chapter, we presented a translator from VHDL to MDG-HDL. We first transform each VHDL process into the Control/Data Flow Graphs (CDFGs) using the LEDA Graph Generator tool. A Directed Acyclic Graph (DAG) is then constructed from CDFGs. We generate MDG-HDL from the resulting DAG by translating each node into a predefined component of MDG-HDL. We illustrated the translation in two examples: the ITC counter and a Moore machine. The automatic translation from VHDL to MDG-HDL enables the verification of

designs described in the synchronous RTL subset of the VHDL hardware description language.

Chapter 8 Conclusions and Future Work

8.1 Conclusions

ROBDDs have proved to be a powerful tool for automated hardware verification. However, they are not adequate in general for verifying circuits with large and complex datapaths, because of the Boolean representation of circuits. Our group has proposed a new class of decision graphs, called Multiway Decision Graphs. With MDGs, we can integrate two verification techniques that have been very successful: *implicit state enumeration* and *the use of abstract sorts and uninterpreted function symbols*. MDGs can represent relations as well as sets of states, and incorporate variables of abstract sorts to denote data values and uninterpreted function symbols to denote data operations.

In this thesis, we explored two techniques to increase the scope of designs that can be verified using MDG: variable ordering to decrease the size of MDGs and automatic translation from VHDL to MDG-HDL.

Like ROBDDs, MDGs require a total order over all the decision variables in the graph and the size of MDGs greatly varies with the order. Since memory requirements and processing time increase with MDG size, it is important to keep MDGs as small as possible. Compared to ROBDDs, variable ordering on MDG is more complicated due to the presence of constraints caused by first order terms in the structure.

We first presented static and dynamic variable ordering algorithms on MDG. Our static variable ordering algorithm generates a variable order before an MDG is built. It is based on several heuristic rules that we explored for MDG. Our dynamic

reordering algorithm minimizes the size of the MDGs during the verification process. It combines the merits of symmetry and group sifting.

We then identified the standard term ordering problem, caused by the lexicographical order adopted by MDG system to order cross-terms having the same cross-operator. Our solution is based on function renaming and cross-term rewriting. Function renaming is used to build an MDG with a good order generated by the static variable ordering algorithm. Cross-term rewriting is used to restore the original behavior of the circuits without changing the size of the MDG.

Finally, we presented an automatic translation from VHDL to MDG-HDL. This translator can accept a VHDL model given at the synthesizable RTL level and generate a MDG-HDL description for the MDG system.

We integrated the translation and the (re)ordering algorithms into the MDG design verification system. Experimental results proved that this updated system can handle larger designs than before by alleviating the effects of state explosion, thus increasing the range of the circuits that can be verified.

My original contributions in this thesis can be summarized as follows:

1. The Development and implementation of automatic variable ordering algorithms on MDG.
   a) Static variable ordering:
      - Exploration of 5 heuristic rules for ordering.
      - Development and implementation of a static variable ordering algorithm for combinational circuits.
      - Development and implementation of a static variable ordering algorithm for sequential circuits.
      - Experiments on benchmark circuits.

b) Dynamic variable ordering:
   - Implementation of a variable swap operation in Multiway Decision Graphs.
   - Implementation of the basic sifting algorithm.
   - Development and implementation of a dynamic reordering algorithm integrating symmetry and group sifting.
   - Experiments on benchmark circuits.

c) Standard term ordering:
   - Identification of the standard term ordering problem.
   - Development of a solution based on function renaming and rewriting.
   - A case study on an ATM congestion controller.

2. Automatic translation from DAG – MDG-HDL:
   - Identification of abstract variables and uninterpreted functions.
   - Translation from DAG- MDG-HDL.
   - Two examples: the Island Tunnel Control Counter and a Moore machine.

8.2 Future Work

Although much work has been accomplished on the MDG-based verification system, there are several areas in which the system could be extended.

- Developing an ω–automaton based model checker on MDG:

  It is possible to represent ω-automaton with MDG and develop an ω-automaton based model checker for the MDG system. ω–automaton based model checking considers containment between the languages representing the implementation and the specification. It is a powerful approach in which the property ("task") and the model are represented by an ω-automaton. It was

defined and used by R. P. Kurshan in COSPAN and included as the verification engine in FormalCheck [42]. While this verification can be referred to as linear-time based, the class of ω-automaton is strictly more expressive than linear-temporal logic, and is well adapted to the specification of fairness constraints and liveness properties, as well as sequential properties such as counting. Besides that advantage, various refinement and abstraction techniques as well as heuristics for decomposition and algorithms for localization have been developed specifically to control the complexity of state-space analysis and to provide a basis for hierarchical verification.

- Link to theorem provers:

  It is possible to explore the links between theorem proving systems and the MDG tool. There are two possible approaches: (1) The model checker can be embedded as a specialized decision procedure in a theorem proving system, and could be used to handle appropriate sub-problems automatically. Namely, some sub-goals could be treated by the model checker. This makes the theorem proving system more efficient. (2) The MDG-based model checking can succeed when output checking and state set inclusion can be decided by rewriting and syntactic matching. When this is not possible (in general, the equivalence of two first order formulas is semi-decidable), we could prove the specific sub-goal using a theorem proving system. For systems with complex structures, such as loops, we have to combine model checking and inductive proofs to accomplish the verification task effectively. Both approaches need a translator to realize the transformation between a sub-goal (a theorem) in theorem proving and a logic formula in the MDG-based model checking.

- Experimental verification of the method using industrial and academic benchmarks:

The MDG-based verification system used variables of abstract type to represent data and uninterpreted functions to describe data operation. The data width is no longer the bottleneck to cause the state explosion. This is ideal for verification of designs with large data path (most telecommunication circuits happen to fall into this category) and academic benchmarks using the MDG system, in order to evaluate and to improve its performance.

- Solving the non-termination problem:

  In some cases, MDGs may suffer from the non-termination of the state enumeration procedure. Some early research has shown that two approaches could solve the non-termination problem in some situations. The first one is based on the use of ρ-terms which can finitely represent infinite sets of state [54]. An extension of the syntax of MDGs and MDG-based algorithms could incorporate ρ-terms to solve the non-termination problem when the generated set of states exhibit certain repetitive patterns. The second approach is to modify the original ASM structural description according to certain rules to avoid the non-termination problem [53]. It would be valuable to explore a more general method that could automatically analyze the ASM description, modify the design description and infer ρ-terms. Implementing these ideas in the MDG package would extend the applicability of the MDG-based verification tool.

- Abstraction of subarrays:

  In our automatic translation from VHDL to MDG-HDL, an array can be abstracted only when no single element of the array or a subarray is used separately in the VHDL model or the properties to be verified. This even precludes the situation in which only one element in the array is used separately. To increase the scope of array abstraction, we can further abstract

subarrays and corresponding functions. An arrary can be spliced into several parts or even single elements, i.e., the subarray(s) can be abstracted and the single element(s) of the array can be taken as concrete variables. Abstract and/or cross functions have to be used to splice and if needed merge any spliced parts. User guidance may be needed to control excessive abstraction that could increase the pessimism of the verification.

Bibliography

[1] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, Vol. C-27, pp. 509-516, June 1978.

[2] K. D. Anon, N. Boulerice, E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, Y. Xu and Z. Zhou. MDG Tools for the Verification of RTL designs. In *Proceedings of Conference on Computer-Aided Verification (CAV'96)*, pp. 433-436, New Jersey, USA, July 1996.

[3] B. Bollig and I. Wegener. Improving the variable ordering of OBDD is NP Complete. *IEEE Transactions on Computers*, Vol. 45, No. 9, pp. 993-1001, September 1996.

[4] S. Boubezari, E. Cerny, B. Kaminska, B. Nadeau-Dostie, 1999, "Testability Analysis and Test-Point Insertion in RTL VHDL Specifications for Scan-Based BIST," IEEE Transaction on Computer-aided Design of Integrated Circuits and Systems, Vol. 18, No. 9, pp. 1327-1340, September 1999.

[5] M. C. Browne, E. M. Clarke, D.L. Dill and B. Mishra. Automatic verification of sequential circuits using Temporal Logic. *IEEE Transactions on Computers*, Vol. 35, No. 12, pp. 1035-1044, December 1986.

[6] R. K. Brayton et. al. VIS: A system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification,* pp. 428-432, Springer Lecture Notes in Computer Science, #1102, Edited by R. Alur and T. Henzinger, New Brunswick, USA, July 1996.

[7] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers,* Vol. 35 No. 8, pp. 677-691, August 1986.

[8] R. E. Bryant. Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. *International Conference on Computer-Aided Design (ICCAD'95)*, pp. 236-243, San Jose, USA, November 1995.

[9] K. M. Bulter, D.E. Ross, R. Kapur and M.R. Mercer. Heuristics to compute Variable Orderings for Efficient Manipulation of Ordered Binary Decision Diagrams. In *Proceedings of 28th ACM/IEEE Design Automation Conference*, pp. 417-420, San Francisco, USA, June 1991.

[10] 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*, pp. 46-51, Orlando, USA, June 1990.

[11] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar and Z. Zhou. Automated Verification with Abstract State Machines Using Multiway Decision Graphs. *Formal Hardware Verification: Methods and Systems in Comparison*, pp. 79-113, edited by T. Kropf, Springer-Verlag Publishers, 1997.

[12] K. C. Chang. Digital Systems Design with VHDL and Synthesis: an Integrated Approach. IEEE Computer Society, Los Alamitos, USA, 1999.

[13] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM transactions on Programming Languages and Systems,* Vol. 8, No. 2, pp. 244-263, April 1986.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. Model checking. The MIT press, Cambridge, Massachusetts, 1997.

[15] F. Corella, M. Langevin, E. Cerny, Z. Zhou and X. Song. State enumeration with abstract descriptions of state machines. In *Proceedings IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (*Charme'95*), pp. 146-160, Frankfurt, Germany, October 1995.

[16] F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. *Formal Methods in System Design*. Vol. 10. No. 1, pp. 7-46, February 1997.

[17] P. Curzon. The Formal Verification of the Fairisle ATM Switching Element. *Technical Report No. 328 & No. 329*, University of Cambridge, Computer Laboratory, March 1994.

[18] D. R. Dams. Abstract Interpretation and Partition Refinement for Model Checking. Ph.D thesis, Eindhoven University of Technology, 1996.

[19] D. R. Dams, O. Grunberg and R. Gerth. Generation of Reduced Models for Checking Fragement of CTL. In *Computer Aided Verification* (*CAV*), pp. 479-490, Elounda, Greece, June 1993.

[20] M. Dewey. Analysis and Design of digital systems with VHDL. PWS Publishing Company, 1997.

[21] R. Drechsler and B. Becker. Binary Decision Diagrams: Theory and Implementation. Kluwer Academic Publishers, 1998.

[22] E. A. Emerson. Temporal and Modal logic. *16th chapter of Handbook of Theoretical Computer Science*, edited by Van Leeuwen, Elsevier Science Publishers B.V., 1990.

[23] E. A. Emerson, A. P. Sistla. Symmetry and Model Checking. In *Computer Aided Verification (CAV)* , pp. 463-478, Elounda, Greece, June 1993.

[24] E. A. Emerson, C. L. Lei. Modalities for Model Checking: Branching Time Logic Strikes Back. *Science of Computer Programming 8*, pp. 275-306, Elsevier Science Publishers, 1987.

[25] K. Fisler and S. Johnson. Integrating Design and Verification Environments through Logic Supporting Hardware Diagrams. In *Proceeding of IFIP Conference on Hardware Description Languages and their Applications*, pp. 669-674, Chiba, Japan, August 1995.

[26] S. J. Friedman and K.J. Supowit. Finding the optimal variable ordering for Binary Decision Diagrams. In *Proc. 24th ACM/IEEE Design Automation Conference*, pp. 348-355, Miami Beach, USA, June 1987.

[27] H. Fujii, G. Ootomo and C. Hori. Interleaving based variable ordering methods for Ordered Binary Decision Diagrams. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design*, pp. 38-41, Dallas, USA, June 1993.

[28] M. Fujita, H. Fujisawa and N. Kawato, Evaluation and Improvements of Boolean Comparison Method based on Binary Decision Diagrams. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pp. 2-5, Santa Clara, USA, November 1988.

[29] M. Fujita, Y. Matsunaga and T. Kakuda, On variable ordering of Binary Decision Diagrams for the Applications of Multi-level Logic Synthesis. In *Proceedings of the European Conference on Design Automation*, pp. 50-54, Amsterdam, February 1991.

[30] M. Fujita, H. Fujisawa and Y. Matsunaga. Variable ordering algorithms for Ordered Binary Decision Diagrams and their Evaluation. *IEEE Trans. on Computer-Aided Design of Circuits and Systems,* Vol. 12, No. 1, January 1993.

[31] M. J. C. Gordon, T. F. Melham. Introduction to HOL: A theorem Proving Environment for Higher Order Logic. Cambridge University Press, Cambridge, UK, 1993.

[32] M. J. C. Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic, *Cambridge University, Computer Laboratory Technical Report No. 68*, Cambridge, England, 1985.

[33] A. Gupta. Formal Hardware Verification Methods: A Survey. *Formal Methods in System Design*, vol. 1, pp. 151-238,1992.

[34] J. Hou and E. Cerny. Model Reductions and a Case Study. In *Formal methods in Computer-Aided Design* (*FMCAD*), pp. 299-315, Austin, USA, November, 2000.

[35] J. Hou and E. Cerny. Model Reductions in MDG-based Model Checking. In *Conference of IEEE International ASIC/SOC*, Washington, USA, September 2000

[36] G. D. Hachtel, F. Somenzi. Logic Synthesis and Verification Algorithms. Kluwer Academic Publishers, 1996.

[37] A. J. Hu. Formal hardware verification with BDDs: an introduction. *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*, pp. 677-682, Victoria, Canada, August 1997.

[38] N. Ishiura, H. Sawada and S. Yajima, Minimization of binary decision diagrams based on exchanges of variables. In *Proceedings of the International Conference on Computer Aided Design*, pp. 473-475, Santa Clara, USA, November 1991.

[39] S. W. Jeong, B. Plessier, G. D. Hachtel and F. Somenzi. Variable Ordering for Binary Decision Diagrams. In *Proceedings of the European Conference on Design Automation,* pp. 447-451, Burssles, March 1992.

[40] T. Kropf. Benchmark-Circuits for Hardware Verification. In *Proceeding 2nd International Conference on Theorem Provers in Circuit Design (TPCD 94)*, LNCS 901, pp. 1-12, Springer-Verlag, September 1994.

[41] R. P. Kurshan. Automata-Theoretic Verification of Coordinating Processes. Princeton University Press, 1994.

[42] R. P. Kurshan. Formal Verification in a Commercial Setting. In *Proceedings of the 34th Design Automation Conference (DAC'97)*, pp. 258-262, Anaheim, USA, July 1997.

[43] R. P. Kurshan. Analysis of Discrete Event Coordination. In *Research and Education in Concurrent Systems (REX) Workshop*, pp. 414-453, Mook, The Netherlands, 1989.

[44] H. Liu. Congestion Control for ATM in Broadband ISDN. Master Thesis, University of Saskatchewan, August, 1992.

[45] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of International Conference on Computer-Aided Design*, pp. 6-9, Santa Clara, USA, November 1988.

[46] M. C. McFarland. Formal Verification of Sequential Hardware: A Tutorial. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, pp. 633-654. Vol.12, No.5, May 1993.

[47] K. L. McMillan. Symbolic Model checking. Kluwer Acardmeic Publishers. 1998

[48] K. L. McMillan. The SMV system draft. http://www-cad.eecs.berkely.edu/~kenmcmil

[49] K. L. McMillan. Get started with SMV. http://www-cad.eecs.berkely.edu/~kenmcmil

[50] C. Meinel and A. Slobodova. Speeding up Variable Reordering of OBDDs. *International conference on Computer Design* (ICCD), pp. 338-343, Austin, USA, October 1997.

[51] S. Minato, N. Ishiura, and S. Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation. *In Proceedings 27th Design Automation Conference,* pp. 52-57, Orlando, USA, June 1990.

[52] S. Minato. Binary Decision Diagrams and Applications for VLSI CAD. Kluwer Academic Publishers, 1997.

[53] A. Otmane, E. Cerny, X. Song. MDG-based Verification by Retiming and Combinational Transformations. In *Proceedings of the IEEE 8th Great Lakes Symposium on VLSI,* pp. 356-361, Louisiana, USA, February 1998.

[54] A. Otmane, X. Song, E. Cerny. On the Non-termination of MDG-based Abstract State Enumeration. In *Proceedings of IFIP International Conference on Correct Hardware Design and Verification Methods, (CHARME'97)*, pp. 218-235, Montreal, Canada, 1997.

[55] S. Panda, F. Somenzi. Who are the variables in your neighborhood? In *Proceedings of International Conference on Computer-Aided Design (ICCAD'95),* pp. 74-77, San Jose, USA, June 1995.

[56] R. K. Ranjan, W. Gosti, R. K. Brayton, A. Sangiovanni-Vincentelli. Dynamic reordering in a Breadth-First manipulation Based BDD Package: Challenges and Solutions. In *Proceedings of IEEE/ACM International Conference on Computer Design*, pp. 344-357, Austin, TX, October 1997.

[57] R. Rudell. Dynamic variable ordering for Ordered Binary Decision Diagrams, In *Proc. of IEEE/ACM International Conference on Computer-Aided Design*, pp. 42-47, Santa Clara, USA, June 1993.

[58] C. Scholl, S. Melchior, G. Hotz, and P. Molitor. Minimizing ROBDD Sizes of Incompletely Specified Functions by Exploiting Strong Symmetries. In European Design and Test Conference, pp. 229-234, 1997.

[59] C. Scholl, D. Moller, P. Molitor and R. Drechsler. BDD Minimization using Symmetries, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 81-100, Vol. 18, No. 2, February 1999.

[60] C. Seger. An Introduction to Formal Hardware verification. In *University of British Columbia Tech. Report 92-13,* 1992.

[61] E. M. Sentovich, K. J. Singh et. al. SIS: A system for Sequential Circuit Synthesis
http://www-cad.eecs.berkeley.edu:80/~ellen/Html/sis.html

[62] S. Tani, K. Hamaguchi and S. Yajima. The Complexity of the Optimal Variable Ordering of a Shared Binary Decision Diagram. *Technical Report 93-6,* Department of Information Science, Faculty of Science, University of Tokyo, December 1993.

[63] D. E. Thomas and P. R. Moorby. The Verilog Hardware Description Language. Fourth edition. Kluwer Academic Publishers, 1998.

[64] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *International Conference on Computer-Aided Design*, pp. 130-133, Santa Clara, USA, June 1990.

[65] H. J. Touati, R. K. Brayton and R. P. Kurshan. Testing Language Containment for ω-automata Using BDDs. In *Information and Computation*, Vol. 118, No. 1, pp. 101-109, 1995.

[66] Y. Xu. Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs, Ph.D thesis, D'IRO, Univeristy of Montreal, 1999.

[67] Y. Xu, E. Cerny, A. Silburt, A. Coady, Y. Liu and P. Pownall. Practical Application of Formal Verification Techniques on a Frame Mux/Demux Chip from Nortel Semiconductors. In *Proceeding Correct Hardware Design and*

Verification Methods (*CHARME'99*), Bad Herrenalb, Germany, pp. 110-124, September 1999.

[68] 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 *Proceedings of Conference on Computer-Aided Verification* (*CAV'98*), pp. 219-231, Vancouver, Canada, July 1998.

[69] Z. Zhou. Multiway Decision Graphs and Their Applications in Automatic Formal Verification of RTL Designs. PhD thesis, D'IRO, University of Montreal, 1997.

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

[71] Z. Zhou, X. Song, F. Corella, E. Cerny and M. Langevin. Description and Verification of RTL designs using Multiway Decision Graphs. In *Proceedings of the Conference on Computer Hardware Description Languages and their applications* (*CHDL'95*), pp. 575-580, Chiba, Japan. August, 1995.

[72] Z. Zhou, X. Song, S. Tahar, F. Corella, E. Cerny and M. Langevin. Formal Verification of the Island Tunnel Controller using Multiway Decision Graphs. In *Proceedings of International Conference on Formal Methods in Computer Aided Design (FMCAD'96),* pp. 233 - 247, Palo Alto, USA, 1996.

[73] LEDA VHDL System: Implementor's Guide – VHDL Intermediate Format. LEDA S.A. Meylan, France, 1994-1997.

[74] LEDA VHDL System: Implementor's Guide – Control/Data Flow Graph Generator. LEDA S.A. Meylan, France, 1994-1997.

[75] FormalCheck Web Page.
http:// www.cadence.com/datasheets/formalcheck.html

[76] FormalCheck User's Guide. Bell Labs Design Automation, Lucent Technologies, v. 2.1, 1997.

[77] Quintus Prolog Manual. Quintus Corporation, v. 3.1, 1991.

Appendix A. Verification of an Island Tunnel Controller

In this appendix we introduce the Island Tunnel Controller (ITC) example we conducted our experiments on in Chapter 4 and 5. We describe the behavioral description of Island Tunnel Controller in Section A.1. In Section A.2, we explain the verification of the ITC model.

A.1 The Island Tunnel Controller

The ITC controls the traffic lights at both ends of a tunnel based on information collected by sensors installed at both ends of the tunnel, a single lane tunnel connecting the mainland to an island, as shown in Figure 4.13. At each end of the tunnel, there is a traffic light. There are four sensors for detecting the presence of vehicles: one at the tunnel entrance (*ie*) and one at the tunnel exit on the island side (*ix*), and one at the tunnel entrance (*me*) and one at the tunnel exit on the mainland side (*mx*).

In [25], the following constraint is imposed: "at most sixteen cars may be on the island at any time". The number "sixteen" can be taken as a parameter and it can be any natural number. The constraint can thus be read as follows: "at most n ($n \geq 0$) cars may be on the island at any time".

The specification of ITC using three communicating controllers and two counters proposed in [25] is shown in Figure A.1. The state transition diagrams are shown in Figure A.2. The island light controller (ILC) (Figure A.2(a)) has four states: green, entering, red and exiting. The outputs *igl* and *irl* control the green and red lights on the island side, respectively; *iu* indicates that the cars from the island side are currently using the tunnel, and *ir* indicates that the island is requesting the

tunnel. The input *iy* requests the island to yield control of the tunnel, and *ig* grants control of the tunnel. A similar set of signals is defined for the mainland light controller (MLC). The tunnel controller (TC) processes the requests for access issued by the ILC and MLC. The island counter and the tunnel counter keep track of the numbers of cars currently on the island and in the tunnel, respectively. At each clock cycle, the count *tc* of the tunnel counter is increased by 1 depending on signals *itc*+ and *mtc*+, or decremented by 1 depending on *itc*- and *mtc*-, unless it is already 0. The island counter operators in a similar way, except that the increment and decrement signals are *ic*+ and *ic*-, respectively.

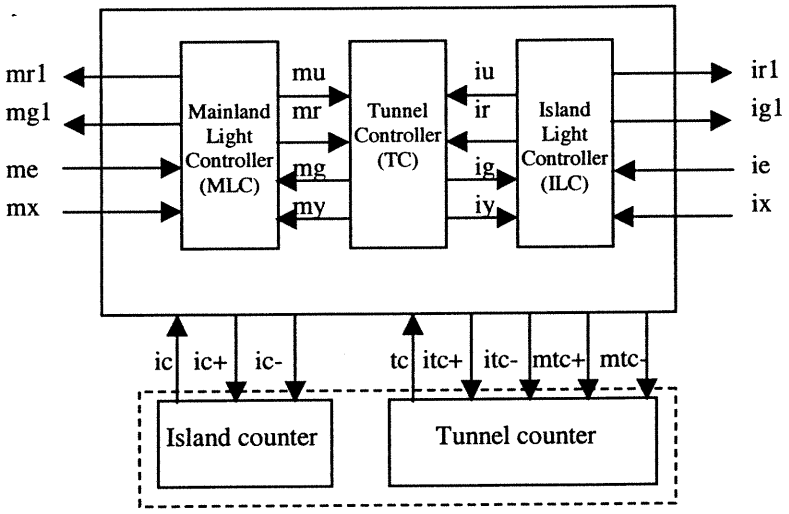
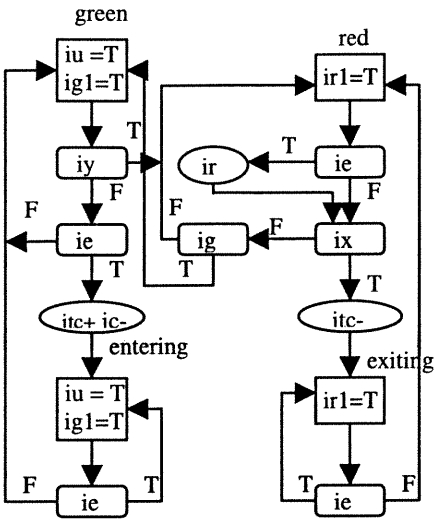
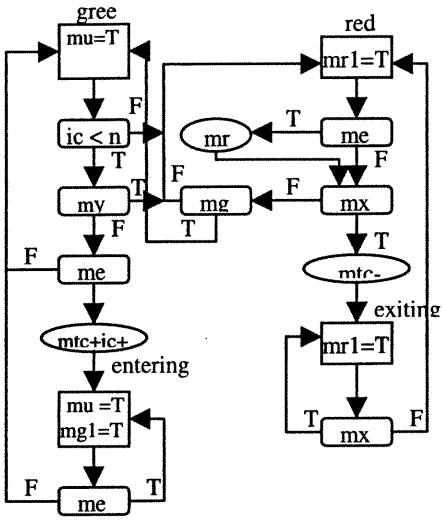


Figure A.1 The Island Tunnel Controller

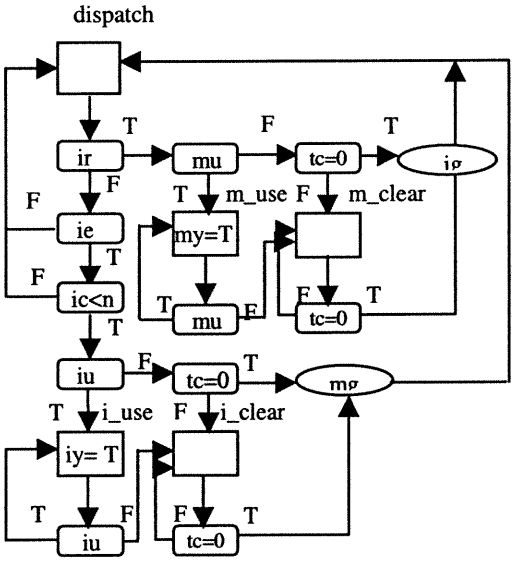
Both the island and the tunnel counters have each only one control state, *ready*, hence no control state variable is needed. We can use a concrete variable to represent the current counter number. The count *ic* (*tc*) is now assigned a concrete sort according to the counter width which is determined by the maximum number of cars that are allowed on the island. We suppose the number if cars that allowed on the island and in the tunnel equals 2^n where n is the counter width.



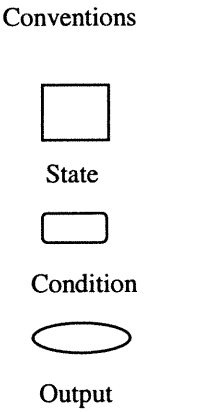
(a) Island light controller

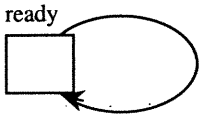


(b) Mainland light controller



(c) Tunnel controller





if (ic+=1) ∧ (ic-=0)
then ic' := ic+1;
else if (ic+=0) ∧ (ic-=0) ∧ (ic≠0)
then ic' := ic-1;
else ic' := ic;

(d) Island counter

if ((itc+=1) ∧ (mtc+=0) ∧ (itc-=0) ∧ (mtc-=0)) ∨
((itc+=0) ∧ (mtc+=1) ∧ (itc-=0) ∧ (mtc-=0))
then tc' := tc+1;
else if ((itc+=0) ∧ (mtc+=0) ∧ (itc-=1) ∧ (mtc-=0)) ∧
(tc≠0)) ∨ ((itc+=0) ∧ (mtc+=0) ∧ (itc-=0) ∧ (mtc-=1))
∧ (tc≠0))
then tc' := tc-1; else tc' := tc;
else ic' := ic;

(e) Tunnel counter

Figure A.2 The state transition diagrams of the Island Tunnel Controller

We can also use an abstract state variable *ic* (*tc*) to represent the current counter number. At each clock, the count is updated according to the control signals. In this abstract description, the count *ic* (*tc*) is of abstract sort *wordn* for n-bit words. The control signals (*ic*+, *ic*-, etc.) are of sort bool with the enumeration {0,1}. The uninterpreted function *inc* of type *wordn* → *wordn* denotes the operation of increment by 1, and *dec* of the same type denotes Decrement 1. The cross-term *equz* (*tc*) represents the condition "*tc* =0" and models the feedback from counter to the control circuitry; *equz* is a cross-function symbol of type *wordn* → *bool*. Each of the controllers can have a single control state variable which takes all the possible states as its values. Thus, the enumeration of those states constitutes the (concrete) sort of the variables.

Let *is*, *ms*, and *ts* be the control state variables of the three controllers ILC, MLC and TC, respectively, and let *is'*, *ms'* and *ts'* be the corresponding next state variables. We define a concrete sort *mi_sort* having the finite enumeration {green, red, entering, exiting}. The variables *is* and *ms* (and also their next state variables *is'* and *ms'*) are then assigned to be of this sort *mi_sort*. Similarly, we let variables *ts* and *ts'* to be of sort *ts_sort* which has the enumeration {dispatch, i_use; m_use; i_clear, m_clear}. All other control signals (*ie*, *ix*, *me*, *mx*, etc) are of sort *bool*. The condition "*ic* < *n*" is represented by the cross- term *lessN*(*ic*), where the uninterpreted cross-function *lessN* of type *wordn* → *bool* represents the operation "< *n*".

A.2 Verification

Property checking is useful for verifying that a specification satisfies certain requirements. We list below three properties (invariants) that we verified and also provide their CTL (Computational Tree Logic) formulas:

P1: The lights on the island side and the mainland side cannot be green at the same time.

AG (!((igl=1) & (mgl=1))).

P2: The tunnel counter is never signaled to increment simultaneously by ILC and MLC.

AG (!((itc+ =1)&(mtc+ = 1))).

P3: The island counter is never signaled to increment and decrement simultaneously.

AG (!((ic- = 1) & (ic += 1).

We verified P1, P2, P3 on ITC with various counter widths ranging from 4 to 10 bits. We also use an abstract variable to describe the counter. We applied our static and dynamic ordering algorithms on this example. The experiments results can be found in pages 67 and 88.

Appendix B Verification of an ATM Switch Fabric

In this appendix we present the Fairisle 4×4 ATM (Asynchronous Transfer Mode) switch fabric we conducted our experiments on in Chapter 4 and 5. The device was in use for real applications in the Cambridge Fairisle network [17], designed at the Computer Laboratory of the University of Cambridge. In Section B.1, we introduce the Fairisle ATM Switch fabric model. In Section B.2, we give its hardware description. In Section B.3, we explain the property checking of ATM model.

B.1 The Fairisle ATM Switch Fabric

The 4×4 Fairisle switch consists of three types of components: the input port controllers, the output port controllers and the switch fabric, as shown in Figure 4.14. An (Fairisle) ATM cell consists of a header (one-byte tag containing routing information as shown in Figure 4.15) and a fixed number of octets. Cells are switched from input ports to output ports according to the header.

The behavior of the switch is cyclical. In each cycle or frame, the input port controllers synchronize incoming data cells, append control information in the front of the cells, and send them to the fabric. The fabric waits for cells to arrive, strips off the header, arbitrates between cells destined to the same port, sends successful cells to the appropriate output port controllers, and passes acknowledgments from the output port controllers to the input port controllers.

If different port controllers inject cells destined for the same output port controller (as indicated by the *route bits* in the header) into the fabric at the same time, then only one will succeed. The others must retry later. The routing tag also includes

priority information (*priority bit*) that is used by the fabric for arbitration which takes place in two stages. High priority cells are given precedence before the remaining cells. The choice between cells of the same priority is made on a round-robin basis. The input controllers are informed of whether their cells were successful using acknowledgment signals. The fabric sends a negative acknowledgment to the unsuccessful input ports, but passes the acknowledgment from the requested output port to the successful input port. The port controllers and the switch fabric all use the same clock, hence bytes are received synchronously on all links. They also use a higher-level cell frame clock – the frame start signal f_s. It ensures that the port controllers inject data cells into the fabric synchronously so that the routing tags arrive at the same time. If no input port raises the active bit throughout the frame then the frame is inactive - no cells are processed. Otherwise it is active.

Figure B.1 shows a block diagram of a 4×4 switch fabric. The inputs to the fabric consist of the cell data lines, the acknowledgments that pass in the reverse direction, and the frame start signal f_s which is the only external control signal. The outputs consist of the switched data, and the switched and modified acknowledgment signals. The switch fabric is composed of an arbitration unit, an acknowledgment unit and a dataswitch unit. The arbitration unit reads the headers, makes arbitration decisions when two or more cells are destined for the same output port, passes the result to the other modules using grant signals and controls the timing of the other units using output disable signals. The dataswitch performs the actual switching of data from an input port to an output port according to the most recent arbitration decision. The acknowledgment unit passes appropriate acknowledgment signals to the input ports. Negative acknowledgments are sent until arbitration is completed.

All the design units were repeatedly subdivided until eventually the logic gate level was reached, providing a hierarchy of components. The design had a total of

441 logic gates with two or more inputs and flip-flops. The switch fabric was built on a 4200 gate equivalent Xilinx FPGA.

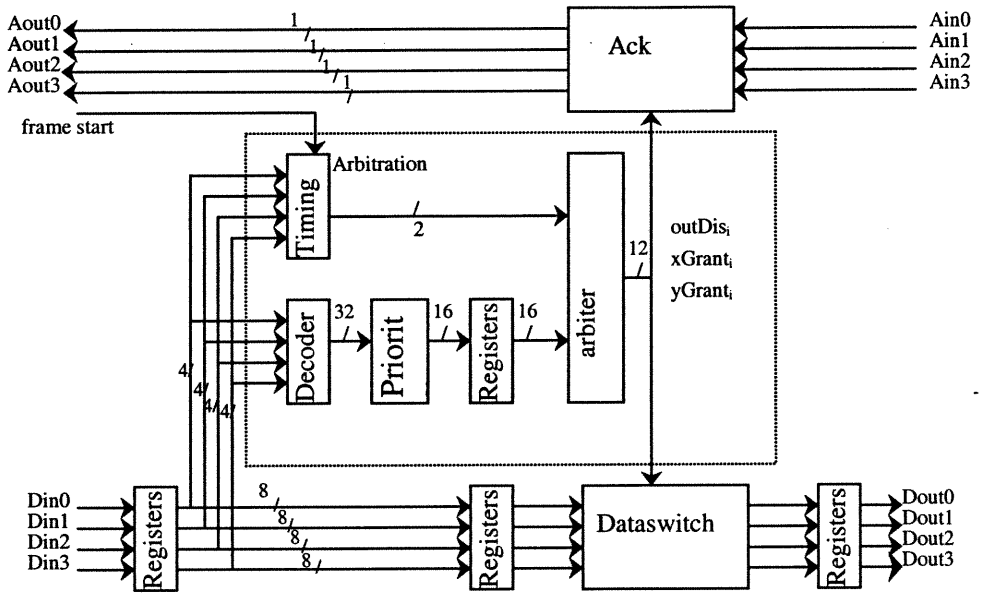


Figure B.1 The block diagram of the Fairisle ATM switch fabric

B.2 Hardware Description

B.2.1 Gate and RT Implementation

Figure B.2 described the RTL implementation of the fabric based on the gate-level description by describing the dataswitch using multiplexors instead of logic gates. The data signals Din_i / $Dout_i$ (i = 0, 1, 2, 3) are modeled as n-bit words and assigned an abstract sort *wordn*. The control fields contained in the cell headers, i.e. active, priority and route bits, are extracted from the abstract data signals using cross-operators (functions) act, pri of type *wordn* → *bool* and rou of type *wordn* →

word2 (*word2* = {0, 1, 2, 3}) respectively. The MDG-ASM model is thus obtained by compiling the abstract description of the RTL implementation.

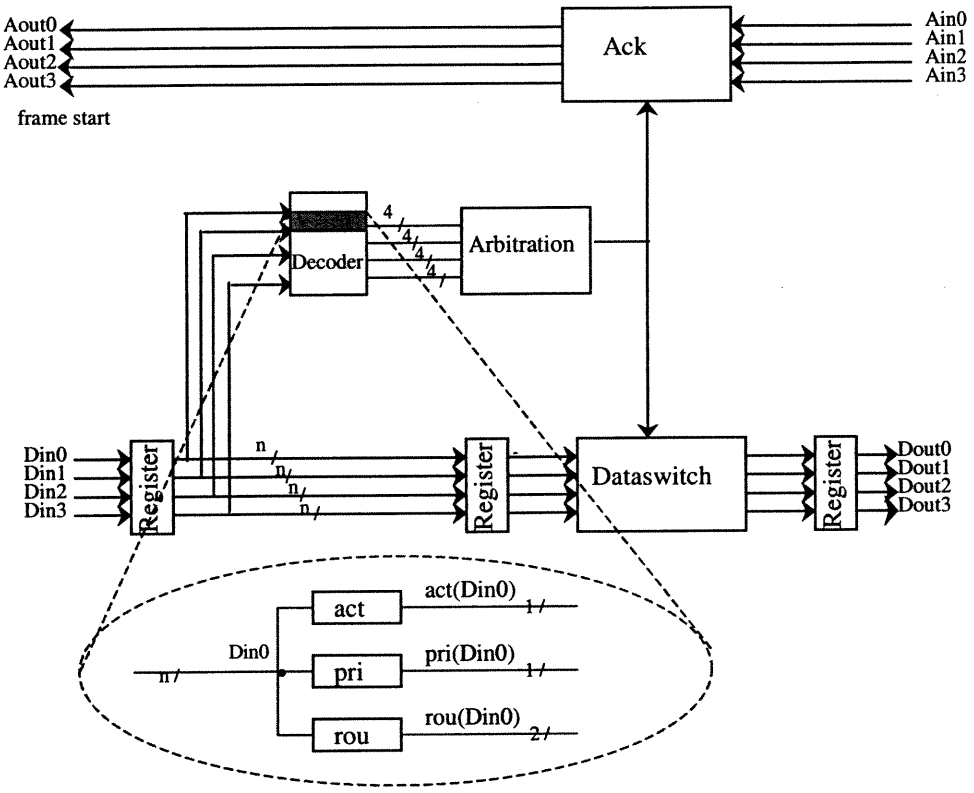


Figure B.2 Model abstraction of the switch fabric

B.2.2 Behavioral Specification

This specification was developed independently of the actual hardware design and includes no restrictions on the frame and cell lengths and the word width. It reflects the complete behavior of the fabric under the assumption that the environment maintains certain timing constraints on the arrival of the frame start signal and the cell headers.

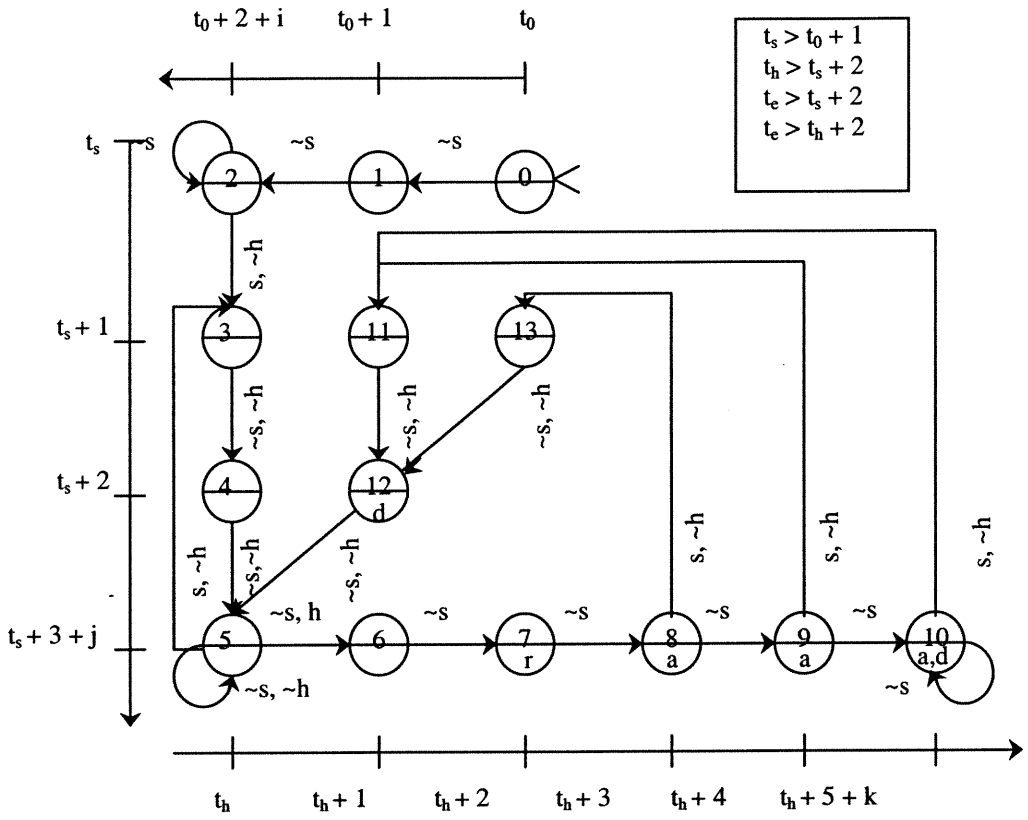


Figure B.3 The Timing of the Fairisle switch fabric

The state machine as shown in Figure B.3 defines the exact timing relationship in clock cycles between inputs and output responses. The symbols *s* and *h* denote the arrival of the frame start signal f_s and of the headers, respectively, and the symbol '~' denotes negation. There are 14 states. States 0, 1 and 2 along the time axis t_0 describe the initial behavior of the switch fabric. States 3, 4 and 5 along the time axis t_s describe the processing when the f_s signal arrives. States 6 to 13 along the time axis t_h represent the behavior after the arrival of headers. In the loops on states 2 and 5, the fabric awaits the f_s signal and the hearders, and the loop on state 10 represents the data transfer cycles. The positive integers *i*, *j* and *k* are used to expand the three time axes t_0, t_s and t_h, respectively. The symbols *r*, *a* and *d* inside the circles indicate the operation taking place in the states: round-robin arbitration, the output of acknowledgments and the output of data. The absence of those

symbols means that there is no operation taking place and that the default value is output. The operations are defined by separate state machines. The data signals Din_i and $Dout_i$ (i = [0..3]) are of an abstract sort *wordn* , and the acknowledgement signals Ain_i and $Aout_i$ (i=[0..3]) are of sort *bool*. The state variable c is of a concrete sort having the enumeration [0..13]. The same function symbols (act, pri and rou) as those in the RTL implementation are used to extract the control information in the headers.

B.3 Property Checking of ATM model

We use the time points t_s, t_h and t_e to denote the start of a frame, the start of an active cell, and the end of a frame (which is the start of the next frame), respectively. Using these time points, we can state several properties which reflect the behavior of the switch fabric. The properties we verified on the switch fabric are :

P1: From $t_s + 3$ to $t_h + 4$, the default value is put on the data output port 0.

P2: From t_s +1 to t_h +2, the default value is put on the acknowledgment output port 0.

P3: From $t_h + 3$ to t_e, if input port 0 chooses output port 0 with the priority bit set in the routing tag, and no other input ports have their priority bits set, the value on $Aout_0$ will be the input of Ain_0.

P4: From t_h +5 to t_e +2, if input port 0 chooses output port 0 with the priority bit set in the header and no other input ports have their priority bits set, the value on $Dout_0$ will be Din'_0 which is the input of Din_0 of 4 clock cycles earlier.

We verified P1 to P4 on the ATM model using our static and dynamic reordering algorithms. The experiment results can be found in pages 69 and 87.