Université de Montréal

Model Reductions in MDG-based Model Checking

par

Jin Hou

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:

Model Reductions in MDG-based Model Checking

présentée par:

Jin Hou

a été évaluée par un jury composé des personnes suivantes:

Marc Frappier — président-rapporteur

Professor Eduard Cerny — directeur de recherche

Professor Xiaoyu Song — co-directeur de recherche

Abdelhamid El Mostapha — membre du jury

Wang Yuke — examinateur externe

Véronique Aubin — représentant du doyen de la FES

Résumé

En model-checking, la méthode utilisée pour vérifier si une implantation satisfait une spécification est basée sur l'exploration de l'espace d'états accessibles du modèle de l'implantation. Pour la plupart des systèmes, leur espace d'états est tellement grand qu'il est quasiment impossible de l'explorer complétement avec un outil de vérification de modèle. Ceci est reconnu comme le problème de l'explosion d'états. Les techniques les plus puissantes pour résoudre ce problème sont basée sur la réduction du modèle. Si l'on peut réduire le modèle original M à un modèle plus petit M' tel que M satisfait à la propriété P si et seulement si M' satisfait également à P, alors la vérification de propriété P peut être effectuée en utilisant M' et ainsi potentiellement éviter l'explosion d'états.

Dans cette thèse, nous nous concentrons sur les techniques de réduction pour résoudre le problème de l'explosion d'états dans le système de vérification basée sur le MDG (MDG : Multiway Decision Graphs). Le système adopte MDG pour représenter symboliquement l'ensemble d'états et les relations de transition des machines à états abstraits. Dû à la présence des variables abstraites et des symboles de fonctions non-interprétees, il n'y a pas d'opération préimage sur le MDG. Tous les algorithmes qui utilisent le calcul de préimage ne peuvent alors pas être appliqués dans notre cas.

Nous présentons deux techniques de réduction. L'une est basée sur la topologie des circuits et l'autre sur la dépendance fonctionnelle de la propriété à vérifier. Nous définissons le circuit *suffisant* et le PDG (Property Dependency Graph) pour une propriété P et prouvons que le circuit suffisant et le modèle réduit M' qui utilise toutes les variables d'état dans le PDG préservent fortement P, c'est-à-dire que $M' \models P \Leftrightarrow M \models P$.

Cependant, le circuit suffisant ou le modèle réduit utilisant toutes les variables d'état de PDG pourrait encore conduire au problème de l'explosion d'états. Nous prouvons que les modèles réduits qui utilisent un sous-ensemble des variables d'état dans le circuit suffisant ou PDG préservent faiblement P, c'est-à-dire que si le modèle réduit possède P, alors le modèle original aussi, mais si le modèle réduit ne possède pas P, le modèle original pourrait posséder P. Nous présentons des algorithmes de réduction itératifs pour enlever plus de variables d'état. Nous utilisons les portes à entrance multiple dans le circuit pour guider la réduction itérative basée sur la topologie du circuit. La recherche en profondeur et la recherche en largeur sur le PDG partitionné nous permet d'ajouter itérativement de variables d'état dans la méthode basée sur la dépendance fonctionnelle.

Les méthodes proposées dans cette thèse sont complètement automatiques sans l'intervention de l'utilisateur. Nous les avons intégrées dans un outil de vérification de modèle MDG, donc rendu capable de vérifier les designs réels. Nous avons effectué différentes expériences de validation et les résultats montrent que nos méthodes peuvent réduire de manière efficace les modèles et elles fonctionnent bien même quand les autres outils de vérification échouent.

Mots clés : Vérification formelle de propriétés, Graphe de décision avec chemins multiples (MDG), réduction de modèle, Graphe de dépendance, problème de l'explosion d'états

Abstract

In model checking, the method to verify that an implementation satisfies a specification is based on exploring the reachable state space of its model. For many systems the state space is extremely large and beyond the capacity of model checking, which is referred to as the problem of state explosion. The most powerful techniques to solve this problem are model reductions. If we can reduce the original model M to a smaller model M' such that property P holds on M if and only if P holds on M', then property checking can only be done by using M', which may often avoid the state explosion problem.

In this thesis we focus on the reduction techniques to solve the state explosion problem in MDG Model Checking. MDG Model Checking adopts Multiway Decision Graphs (MDG) to symbolically represent sets of states and transition relations of Abstract State Machines (ASM). Due to the presence of abstract variables and uninterpreted function symbols, there is no preimage operation in MDG. All reduction algorithms that use preimage computation cannot be applied to MDG Model Checking.

We present two reduction techniques. One is based on the topology of circuits and the other is based on the functional dependency of the property to be verified. We define the *sufficient* circuit and the Property Dependency Graph (PDG) for a property P and prove that the *sufficient* circuit and the reduced model M' using all state variables in the PDG strongly preserve P, i.e., $M' \models P \Leftrightarrow M \models P$.

However, the *sufficient* circuit or the reduced model using all state variables in PDG may still lead to the state explosion problem. We prove that reduced models using a subset of the state variables in the *sufficient* circuit or PDG weakly

preserve P, i.e., if P holds on the reduced model then P holds on the original model, but if P fails on the reduced model it may not fail on the original one. We present iterative reduction algorithms to remove more state variables. We use multiple fanin gates in the circuit to guide the iterative reduction based on circuit topology, and depth-first search and breadth-first search on partitioned PDG to iteratively add more state variables in the functional dependency method.

Our methods are completely automatic without user guidance. We have integrated them in the MDG model checking tool and thus made it capable of verifying realistic designs. We carried out experiments on a number of benchmarks and the results illustrate that our methods can reduce models efficiently and work well even when other verification tools fail.

Keywords: formal verification, model checking, Multiway Decision Graph (MDG), model reduction, dependency graphs, state explosion problem

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Acknowledgments

The completion of this dissertation is the results of many years of hard work and the assistance of many people. First and foremost, I would like to thank Prof. Eduard Cerny for his constructive technical advice, financial support and constant encouragement throughout my doctoral studies. Prof. Cerny has given me many suggestions, comments and corrections in the writing of this dissertation. It has been a great experience for me to be a member of his research team.

Thanks to Prof. Xiaoyu Song for his help, financial support, valuable suggestions and continuous inspiration. Prof. Song introduced me to the Formal Verification group at the University of Montreal and thus made it possible for me to enter this wonderful field. Thanks to Yi Feng and Ying Zhang for their friendship and help. Thanks to Michel, the administrator of the VLSI Laboratory in the Dept. IRO, for his help while I was using software tools and equipment. Thanks to all Faculty and Staff, my friends and fellow graduates at the University of Montreal for their help and support. I have enjoyed studying and working with them. Although the names are far too numerous to list here, I would like to thank everyone for making my years here enjoyable.

At last, I would like to give my special thanks to my parents, my husband and my daughter, my sister and my brother in law, for their love, support and continuous encouragement throughout my life.

Chapter 1 Introduction

1.1 Motivation and Goal

Hardware systems are much larger than before and their complexity continues to grow. Locating and correcting design errors can be a time consuming and expensive process. Traditionally, simulation has been the main debugging technique. Ideal simulation needs to simulate all possible input patterns for checking the correctness of the system, which is impossible in practice. Typically a much reduced subset of the input patterns is simulated. The critical and intractable problem in simulation is that it is hard to find an effective simulation sequence that is sufficient to expose any incorrect behavior of the system.

An alternative method to simulation is formal verification. Formal verification overcomes the weakness of non-exhaustive simulation by proving the correspondence between some abstract specification and the design. It is like a mathematical proof in some sense. Just as correctness of a mathematically proven theorem holds regardless of the particular values that it is applied to, correctness of a formally verified hardware design holds regardless of its input values. Thus, consideration of all cases is implicit in a methodology for formal verification. Moreover, since the high-level description of a design at the early design stage can be used in formal verification, the design errors can be caught early, and thus a lot of money and time can be saved.

Formal verification means formally establishing that an implementation satisfies a specification. The implementation refers to the hardware design to be verified. The representation of an implementation can be a network of transistors/gates, finite-

state machines, description in logic, etc. The specification refers to the properties to which correctness is to be determined. It can be expressed as finite-state automata, ω-automata, or logic, e.g., temporal logic, first-order predicate logic and higher-order logic.

Formal verification methods fall into two classes: theorem proving and FSM (Finite State Machine)-based methods. With theorem proving, an implementation and its specification are often expressed as first-order logic or higher-order logic formulas. Their relationship equivalence/implication is regarded as a theorem to be proven within the logic system using axioms and inference rules. The ability to define appropriate theories and to reason about them using a common set of inference rules provides a unifying framework within which all kinds of verification tasks can be performed. But the generality increases the complexity. Most of the theorem-proving systems today are semi-automated, and require much effort on the user part in developing specifications of each component and guiding the proving process. Thus it can only be used by experts.

FSM-based methods model implementations as finite state machines. FSM-based verification is based on state enumeration by reachability analysis which starts from initial states, repeats applying the transition relations to determine the next states, until all reachable states are visited. For equivalence checking, specifications are also represented by FSMs. Two FSMs are equivalent if they produce the same outputs for every possible input sequence. For model checking proposed by Clarke and Emerson [CE81] and Queille and Sifakis [QS81], specifications are represented as a set of properties expressed by formulas in a temporal logic. Validity of the properties is checked at each reachable state of the FSM representing the implementation. If for all reachable states the formulas are true, then the properties hold in the implementation. FSM-based methods can be applied fully automatically, and also they can produce state sequences as counterexamples when verification fails. This makes it possible to diagnose bugs

in designs. Due to the above advantages, FSM-based methods are used more and more in industry to verify complex designs [Ben01][XCSCLP99][JQK97] [XCS97][BM97][BLPV95][BD94][CYF94] [MS91].

The major problem of FSM-based verification is the *state explosion problem*, the number of states for a system is too large to check exhaustively within the limited time and memory available. Therefore, this thesis is studying and proposing methods for alleviating the state explosion problem.

1.2 Related Work

The efficiency of FSM-based verification depends heavily on the size of the reachable state space. The larger the reachable state space is, the more time and memory it takes to verify a system. An explicit representation of the set of reachable states is exponential with the number of state components in a circuit which limits the applications of FSM-based methods to large systems. Burch, Clarke, McMillan, Berthet, Coudert et al [BCMDH90][BCMD90][BCL91_2] [McM92][BCMDH92][BCLMD94] explored a method called *Symbolic Model Checking* to alleviate the state explosion problem by using characteristic functions to symbolically represent sets of states and transition relations. Reduced Ordered Binary Decision Diagrams (ROBDDs) developed by Bryant [Bry86] to represent Boolean functions are used to represent the characteristic functions of sets and relations. Thus a large state space may be stored in a relatively small memory for many practical functions. Operations on ROBDDs can execute transitions on a large set of states at the same time.

Although using ROBDDs to perform an implicit enumeration of the state space has enlarged the useful domain of model checking, the theoretical complexity is still exponential (in the number of state and input variables), and there are circuits that require an exponential number of ROBDD nodes to represent them. Since the

variables are Boolean and an individual variable is needed for every bit of data, ROBDDs are not adequate for verifying circuits with large datapaths. They are primarily useful in verifying control paths. More efficient methods are needed to verify datapaths as well as to verify the interactions between the datapath and the control parts of a design. A new method using Multiway Decision Graphs (MDGs) to represent the relations and sets of states was proposed by Corella, Cerny, Song, Zhou, et al [ZSCC94][CLCZS95][ZSCC95][ZSCCL95][CZSLC97]. MDGs can efficiently represent formulas of a many-sorted first order logic with a distinction of abstract and concrete sorts. In an MDG, a data can be represented by a single variable of abstract sort, and a data operation can be represented by an uninterpreted function symbol. Thus the verification using MDGs is independent of the data path width, which greatly increases the range of circuits that can be verified.

Since ROBDD and MDG must obey a set of well-formedness conditions to be a canonical representation, the variables in ROBDDs and MDGs must be totally ordered. Different orders may produce different sizes of the graphs, and a bad order can result in the state explosion problem. To find the optimal order is an NP-complete problem [THY93]. There are many heuristic algorithms to find good orders [CDM00] [CYB97] [RG97] [PS95] [Rud93] [FDH93] [FFM93] [BBFS93] [CZJYT92][FMK91][ISY91].

Many finite state systems are composed of multiple processes running in parallel. Verifying the specification on the whole system often leads to the state explosion problem. Since most designs have a modular structure, it seems natural to decompose the specification of the whole system into properties of its modules, and verify the properties of the modules separately. If we can prove that the module properties collectively imply the specification of the entire system, then we prove the soundness of this method, which is referred to as compositional verification [McM92][Lon93] [GL94][McM97][TB97][Kai93].

Besides the methods introduced above, there are many methods based on model reductions. If we can reduce model M to a smaller M' such that if M' satisfies property P then M satisfies P and vice verse, we can only use M' to verify P in order to avoid state explosion. We refer to this relation as *strong* preservation of P and express it as $M' \models P \Leftrightarrow M \models P$. The relation of strong preservation of P limits the freedom of reductions and in many cases the resulting reduced model is still too big to be handled. For proving a property P we can often find a smaller reduced model M' such that M' satisfies P implies that M satisfies P. We refer to this implication as *weak* preservation of P and express it as $M' \models P \Rightarrow M \models P$. However in this case if a property fails on the reduced model, it may not fail on the original one.

One reduction method is based on abstract interpretation [CGL92][CGL94] [Dam96]. This method relies on the user to provide an abstract mapping from an original state to an abstract state, and an abstract interpretation for every operation in the system. If the abstraction is appropriate, the smaller abstract state space can be used to verify the properties of the system. This method is not automatic and also the correctness of the abstraction needs to be proven.

Another method is input elimination [SLH98] which reduces the size of the model by existential quantification of the inputs. A deterministic system with free inputs can be transformed into a non-deterministic one without inputs such that they are bisimulation equivalent. All CTL* formulas are then strongly preserved [CGL92]. Model reduction can be viewed as input elimination by transforming the state variables to be eliminated into input variables and then existentially quantifying them out.

Yet another reduction method is a homomorphic reduction in language containment tests [Kur87][Kur90][Kur92][Kur97]. To verify that a system satisfies a given property is to test whether the ω-regular language associated with

the reduced system is contained in the language associated with the property. The reduced automata are derived from the original ones through co-linear automaton homomorphisms. The Cadence FormalCheck tool [Bell98] is based on such a language containment test.

Another reduction method is to exploit symmetry in the structure of the system [CFJ96][ES93][GS97][ID96][Ip96][PB99]. The structural symmetry induces an equivalence relation between states. For verifying the equivalence classes, we need to explore only one state per each class. In [CFJ96] the symmetry of a finite state system was formally described, and it was proven that a formula in CTL* is preserved if all atomic propositions in the formula are invariant under the symmetry group. In [ID96][Ip96] the scalarset data type was added to the Murφ description language for specifying symmetry. Usually it is used to eliminate symmetrically selectable registers or individual bits in a word. Symmetry reduction is also used in the SMV (Symbolic Model Verifier) system [McM98].

Partition refinement is still another method for model reductions [KS83][Fer90] [Dam96][DGG93][LY92][FV98]. For a given system, its state space is partitioned into sets of states, and each set is an abstract state in the reduced model. If the reduced model is bisimular to the original system, then the properties in CTL* are strongly preserved.

Generally there are no universal solutions to the state explosion problem. All the above heuristics and solutions work only for certain classes of problems. Usually, a combination of some of these methods is needed.

1.3 Scope of the Thesis

We use and have further developed the MDG model checker [XCSCM98][Xu99] [ZSCC94][ZSCCL95]. It can verify properties expressed by a subset of a 1st-order

ACTL. The state explosion problem still limits the tool. Our work is to partially solve this problem and make MDG a practical verification tool. In the preceding section we surveyed many techniques for approaching this problem. However, not all of these methods can be applied to MDG model checking, and also not all of them are efficient and automatic. From our experience of using formal verification tools we realized how important it is to have the automatic reduction feature in these tools. Thus we focus on developing automatic and efficient reduction algorithms that can be used in MDG model checking as well as in other tools.

Since there are abstract variables and uninterpreted functions in MDG, data can be represented by one abstract variable, and thus the bit symmetry reduction is not needed. We tried a state splitting algorithm [Dam96][DGG93], but we encountered several problems. First, in MDG the original property is transferred to a circuit as an additional state machine and a simplified property. The transferred property itself is too simple to contain sufficient information for the splitting algorithm to carry out reductions. This is also the case when the property is directly encoded in the circuit. Second, due to the presence of abstract variables, there is no complement operation and also no conjunction operation of two MDG graphs having the same primary abstract variables. Without these basic operations, we cannot compute the preimage of a set of states that is used in the splitting algorithm, and thus we cannot use this method in MDG. In general all the partition refinement methods and other reduction methods based on finding a bisimulation relation using a preimage computation cannot be applied to MDG.

We first propose a simple way to verify the property by using a reduced circuit obtained by topology analysis of the original circuit. In the MDG model checker, the property P to be verified is transferred to an additional circuit and a simplified property, e.g., **AG**(*flag* = 1) is then verified. We begin from the signal *flag*, the output of the circuit, and search back to the inputs. If a part of the circuit cannot be reached that means it is isolated from the connected part containing *flag*, this part will be removed. The verification is then done on the reduced model. Furthermore,

we can consider the circuits with multiple fanin gates, e.g., *AND/NAND/OR/NOR/ XOR*-gates. In many cases only a part of the design connected to some inputs of multiple fanin gates can generate the result **AG**(*flag* = 1). We present a heuristic algorithm to iteratively select the input branches of the multiple fanin gates in a design. Each time we use the selected part of the design to verify the property and reduce the other state variables to primary inputs. If the property holds on the reduced model, it also holds on the original model, since the other state variables are considered as free primary inputs, i.e., the reduced model represents a larger state space. If the property does not hold on the reduced model, another part of the circuit is used. Finally when the whole connected part containing *flag* is used, the property is strongly preserved.

In the above method we only consider the topology of the circuits. A connected part of a circuit may still contain some state variables that do not affect the value of *flag*. We try to remove these variables by considering the functional dependency. We define the *property dependency graph* (PDG) and the *noncorrelated sets*. We have proved that the resulting abstract system constructed by using all the state variables in the PDG is the least model regardless the initial states that strongly preserves P, and consequently the abstract system constructed using only a subset of these variables weakly preserves P. Thus we can construct the first abstract system using the state variables appearing in the property. These are associated with the root of the PDG. Then we search the PDG to progressively add state variables to construct abstract systems on which P is verified. The critical thing is how to select state variables. We partition the nodes in a PDG by finding noncorrelated sets. The resulting graph is called *partitioned property dependency graph* (PPDG). We have developed two iterative algorithms that select state variables based on depth-first search and breadth-first search in the PPDG.

Using our method we can construct the abstract model by only considering the reachable abstract states. Since no preimage operation is needed, our methods can be used with the MDG model checking. Experimental results show that our

methods can do efficient model reduction even in cases where other tools fail [HC00] [HC00-2].

Contributions:

1. Two heuristic reduction algorithms based on the topology of circuits.
2. A proof that the abstract model constructed using all the state variables in the PDG is the least model regardless of the initial states that strongly preserves property P, and the abstract models constructed using a subset of these state variables weakly preserve P.
3. Two iterative reduction algorithms based on different search strategies in the PPDGs.
4. The integration of our algorithms in the MDG model checker that has extremely improved the behavior of this tool and made it usable on large designs.
5. Experiments on a number of benchmarks.

Outline of the thesis:

In Chapter 2 we review model checking techniques and temporal logics used in FSM-based verification. Symbolic methods and existing model checking tools are also introduced.

In Chapter 3 we review the theoretical foundations of model abstractions and reductions.

In Chapter 4 we introduce Multiway Decision Graphs (MDG) and L_{MDG} used in MDG model checking. We then explain the MDG model checking algorithms.

In Chapter 5 we give two heuristic reduction algorithms based on topological analysis of circuits.

In Chapter 6 we define a property dependency graph (PDG) and noncorrelated sets of state variables. We prove that a property is strongly preserved when it is verified by using all the state variables in its PDG, and weakly preserved when it is verified by using a subset of these state variables. We then show how to construct PDG in the MDG model checker.

In Chapter 7 we present two iterative reduction algorithms based on a depth-first search and a breadth-first search of a PPDG.

In Chapter 8 we describe the integration of our reduction algorithms in the MDG model checker.

In Chapter 9 we verify a number of benchmark designs using our system, SMV and FormalCheck, and then compare the results obtained.

In Chapter 10 we give conclusions of the thesis and outline the future directions of research.

Chapter 2 Model Checking and Temporal Logics

Model checking determines the validity of a specification with respect to a behavioral model of a system. The implementation is represented as an FSM and the specification as a set of properties expressed by formulas in a temporal logic. The validity of the properties is checked by exploring the reachable state space of the implementation FSM. If in all reachable states the formulas are true, then the properties hold on the implementation. In the following we will first introduce the basic concepts of temporal logics and the Computation Tree Logic (CTL), a widely used temporal logic in model checking. Then we will introduce symbolic model checking and the tools we have today.

Temporal logics [Pnue86] are a class of formal logics that allow reasoning about dynamically changing situations. They provide a formal system for describing how the truth values of assertions change over time without time being explicitly mentioned. There are four basic temporal operators: Always (**G**), Sometimes (**F**), Next-time (**X**), Until (**U**).

G*p* is true in state *s*, if *p* is true in all future states from *s*.

F*p* is true in state *s*, if *p* is true in some future states from *s*.

X*p* is true in state *s*, if *p* is true in the next state from *s*.

*p***U***q* 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.

Specification properties such as safety, liveness and precedence properties can be easily expressed in a temporal logic. Safety properties assert that nothing "bad" happens, represented as $M \models \mathbf{G}p$, which means *p* holds at all times on model *M*.

For example, a safety property could be that at an intersection the traffic lights of different directions cannot be green at the same time.

Liveness properties assert that eventually something "good" will happen, represented as $M \models p \Rightarrow \mathbf{F}q$, if p is initially true then q will eventually be true on model M. For example, a liveness property could be that if the traffic light is now red, it will turn to green in the future.

Precedence properties assert the precedence order of events, represented as $M \models p\mathbf{U}q$, which means on model M, p will hold until q becomes true. For example, a precedence property could be that if the traffic light is yellow now, it will stay yellow until it becomes red.

Temporal logics can be classified [1] into linear time temporal logics (LTTL) and branching time temporal logics (BTTL). In an LTTL, time is characterized as a single linear sequence events. In a BTTL, a branching view of time is taken, at any instant there are branching possibilities into the future. It is suitable for defining the semantics of non-deterministic programs. For example, if p represents the fact of a program terminating, then the inevitable termination is expressed by the formula $\mathbf{AF}p$ and a possible termination is expressed by $\mathbf{EF}p$. Here **A** means for all computation paths, and **E** means that there is a computation path.

2.1 Computation Tree Logic (CTL)

The temporal logic CTL is a branching time temporal logic defined by Clarke and Emerson [CE81]. In CTL formulas there are path quantifiers and temporal operators. There are two path quantifiers: **A** denotes that something should be true for all "paths" starting from the current state, and **E** denotes that there exists a path starting from the current state having some property. There are five temporal operators in CTL: **G** ("always"), **F**("eventually"), **X** ("next time"), **U** ("until"), and

R ("releases"). They are used to describe the ordering of events along the path or paths indicated by the **A** or **E**. The meanings of **G**, **F**, **X**, **U** have been defined previously. A path satisfies $p\mathbf{R}q$ if q is true at the current state, and remains true up to and including the first state where p is true. p is not required to hold eventually, but when it does, it releases the requirement that q is true. The difference between *Req***U***Ack* and *Ack***R***Req* is that acknowledgment must occur in *Req***U***Ack* but may never occur in *Ack***R***Req*.

Syntax

CTL restricts the formulas in such a way that the linear time operators must be immediately preceded by a path quantifier, and linear time operators cannot be combined directly with propositional connectives. There are two types of formulas in CTL: state formulas that are true in a specific state and path formulas that are true along a specific path. The syntax of CTL formulas is as follows [CGL94] [CGL96]:

For state formulas:

1. Every atomic proposition is a state formula.
2. If f and g are state formulas, then so are $\neg f, f \vee g$ and $f \wedge g$.
3. If f is a path formula, then $\mathbf{A}(f)$ and $\mathbf{E}(f)$ are state formulas.

For path formulas:

1. If f is a state formula, then f is also a path formula.
2. If f and g are path formulas, then $\neg f, f \vee g, f \wedge g$, $\mathbf{G}f$, $\mathbf{F}f$, $\mathbf{X}f$, $f\mathbf{U}g$ and $f\mathbf{R}g$ are path formulas.

Semantics

Formally, CTL formulas are interpreted relatively to a transition system called a Kripke structure. A Kripke structure $M = (S, R, L)$ is a tuple of the following form:

- S is a finite set of states;
- R is a total binary relation on states and represents possible transitions;
- L: $S \rightarrow 2^{AP}$ is a function that labels each state with a set of atomic propositions true in that state. AP is the set of atomic proposition names.

As its name suggests, CTL interprets temporal formulas over structures that resemble infinite computation trees. A computation tree is formed by starting from a designated state called initial state in a Kripke structure and unwinding the graph (S, R) into an infinite tree. The computation tree illustrates all the possible executions starting from the initial state. The semantics of CTL given below are equivalent to the semantics with respect to the infinite tree.

A path in a Kripke structure M starting from s_0 is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. We let $\pi^i = s_i, s_{i+1}, \ldots$ be a suffix of π. If f is a state formula, the notation $M, s \models f$ means that f is true at state s in the structure M. If f is a path formula, the notation $M, \pi \models f$ means that f holds along path π in the structure M. The truth of a CTL formula is defined inductively as follows:

1. $(M, s) \models p$ iff $p \in L(s)$, where p is an atomic proposition
2. $(M, s) \models \neg f$ iff $(M, s) \not\models f$
3. $(M, s) \models f \vee g$ iff $(M, s) \models f$ or $(M, s) \models g$
4. $(M, s) \models \mathbf{E} f$ iff there exists a path π starting from s such that $\pi \models f$
5. $(M, \pi) \models f$ iff s is the first state of π and $s \models f$
6. $(M, \pi) \models \neg f$ iff $\pi \not\models f$
7. $(M, \pi) \models f \vee g$ iff $(M, \pi) \models f$ or $(M, \pi) \models g$
8. $(M, \pi) \models \mathbf{X} f$ iff $\pi^1 \models f$
9. $(M, \pi) \models f \mathbf{U} g$ iff $\exists k \geq 0$ such that $(M, \pi^k) \models g$, and $\forall i, 0 \leq i < k$, $(M, \pi^i) \models f$
10. $(M, \pi) \models f \mathbf{R} g$ iff for all $k \geq 0$, if for every $i < k$ $(M, \pi^i) \not\models f$ then $(M, \pi^k) \models g$

CTL* [CE86][CES86] is an extension of CTL and 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*∧**F***q*) is a basic modality of CTL*.

Sometimes we want to restrict the logics CTL* and CTL so that they cannot express the existence of a specific path in the Kripke structure. The *Universal CTL** (or ACTL*) and *Universal CTL* (or ACTL) are obtained by eliminating the existential path quantifier from the logic CTL* and CTL respectively [GL91]. To ensure that existential path quantifiers do not arise via negation, in ACTL* and ACTL negations can only be applied to atomic propositions. Thus a formula in ACTL* and ACTL can include only the universal quantifiers over paths.

Fixpoint characterization of CTL

CTL properties can be characterized as fixpoints of appropriate continuous functions. This allows us to use the standard fixpoint algorithm to determine the set of states of a given model in which a CTL formula is true, and thus have efficient algorithms for model checking.

For a finite Kripte structure $M = (S, R, L)$, to obtain the fixpoint characterization, we identify each CTL formula f with $\{s \mid s \models f\}$, the set of states in which f is true. Then *false* represents the empty set and *true* represents the complete set of states S, and any formula f represents a subset of S. Let 2^S be the power set of S (the set of all subsets of S), $\subseteq$ be the order of set inclusion. $(2^S, \subseteq)$ forms a complete lattice. Let τ: $2^S \rightarrow 2^S$ and it is monotonic: if $S_1 \subseteq S_2$, then $\tau(S_1) \subseteq \tau(S_2)$. According to Tarski's theorem [Tar55], τ has a least and a greatest fixpoint with respect to

inclusion order. A fixpoint of τ is a set of states S' such that $\tau(S') = S'$. A least fixpoint is denoted by $\mathbf{lfp}Y.\tau[Y]$ and a greatest fixpoint is denoted by $\mathbf{gfp}Y.\tau[Y]$.

There is a standard algorithm for computing the least and the greatest fixpoint of a monotonic function. It starts with empty set *false* and the whole set *true* respectively, and repeats applying the function τ on the last set until a fixpoint is reached. This procedure will terminate in at most $|S| + 1$ iterations.

Computing the least {or greatest} fixpoint:

Y := false; {or Y := true}
do
  Y' := Y; $Y := \tau(Y)$;
until $Y' = Y$;
return Y

Clarke and Emerson proved that each of the basic CTL operators can be characterized as a least or a greatest fixpoint of an appropriate predicate transformer.

$\mathbf{AF}\,f = \mathbf{lfp}Y.\,[f \vee \mathbf{AX}Y]$

$\mathbf{EF}\,f = \mathbf{lfp}Y.\,[f \vee \mathbf{EX}Y]$

$\mathbf{AG}\,f = \mathbf{gfp}Y.\,[f \wedge \mathbf{AX}Y]$

$\mathbf{EG}\,f = \mathbf{gfp}Y.\,[f \wedge \mathbf{EX}Y]$

$\mathbf{A}[f\,\mathbf{U}\,g] = \mathbf{lfp}Y.\,[g \vee (f \wedge \mathbf{AX}Y)]$

$\mathbf{E}[f\,\mathbf{U}\,g] = \mathbf{lfp}Y.\,[g \vee (f \wedge \mathbf{EX}Y)]$

$\mathbf{A}[f\,\mathbf{R}\,g] = \mathbf{gfp}Y.\,[g \wedge (f \vee \mathbf{AX}Y)]$

$\mathbf{E}[f\,\mathbf{R}\,g] = \mathbf{gfp}Y.\,[g \wedge (f \vee \mathbf{EX}Y)]$

Clarke, Emerson and Sistla [CES86] 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 $O(\text{length}(f)\times(|S|+|R|))$.

2.2 Symbolic Model Checking

In the original implementation of model checking algorithm, transition relations were represented explicitly by adjacency lists. This old method can only handle concurrent systems with small number of processes and states. E. Cerny et al proposed a new way to represent transition relations by using Boolean characteristic functions [Cer80][Cer77][CM77]. Burch, Clarke, Berthet, Coudert et al presented a new method for model checking which is called *Symbolic Model Checking* [BCMD90][BCM90] [BCL91_2][McM92][BCMDH92][BCLMD94]. Symbolic model checking uses Quantified Boolean Formulas (QBF) [AHU74] to represent sets and relations. QBF is an extension of propositional logic allowing quantifiers over propositional variables. The set operations such as union, intersection and image can be characterized in terms of Boolean operations. The well-developed techniques for manipulating Boolean formulas can thus be applied to CTL model checking. Since a Kripke structure is represented symbolically by Boolean formulas, there is no need to construct it as an explicit data structure. Hence the state explosion problem can be reduced.

A state of a concurrent system is generally modeled as a vector where each element represents one state bit of one component of the system. Thus a state of the system can be viewed as a truth assignment to a set of propositional variables $V=\{v_1,\ldots,v_n\}$. Under this interpretation, a set of states can be represented by a QBF formula. For example, if there are two state variables v_1 and v_2, then the formula $v_1 \wedge v_2$ represents the set of states in which v_1 is true and v_2 is true.

For representing a binary relation with a QBF formula, we let the variables $V=\{v_1,\ldots,v_n\}$ represent the current state, and the variables $V'=\{v_{1'},\ldots,v_{n'}\}$ represent the next state. The transition relation of the system can be represented by a boolean formula $R(V,V')$. The image S' of a set S is computed by the following QBF operations, where '$\leftarrow$' means substitution.

$S' = (\exists V.\ (S \wedge R(V,V'))) (V' \leftarrow V)$

For manipulating Boolean formulas automatically and efficiently, Reduced Ordered Binary Decision Diagrams (ROBDD) [Bry86] are used to represent Boolean formulas. The symbolic model-checking algorithm is implemented by a procedure *Check* that takes the CTL formula to be verified as argument and returns an ROBDD that represents the states satisfying the formula. If f is an atomic proposition, *check(f)* is the ROBDD representing the set of states satisfying f. If $f = f_1 \wedge f_2$ or $f = \neg f_1$, then *check(f)* is obtained using the algorithm *Apply* given by Bryant for computing 16 logical operations, with *check*(f_1) and *check*(f_2) as arguments. For example formulas of the form **EX***f*, **E**(*f***U***g*) and **EG***f* are handled as follows:

Check(**EX***f*) = *CheckEX*(*Check*(*f*))

Check(**E**(*f***U***g*)) = *CheckEU*(*Check*(*f*), *Check*(*g*))

Check(**EG***f*) = *CheckEG*(*Check*(*f*))

The procedure *CheckEX* is straightforward. It verifies if the current set of states has successors in which f is true. *CheckEU* is based on the least fixpoint characterization of the CTL operator **EU**, and *CheckEG* is based on the greatest fixpoint characterization of **EG**. We use Y^i to represent the set of states computed in the i-th iteration. It is easy to test for convergence by comparing the ROBDDs representing Y^{i-1} and Y^i.

2.3 Existing Model Checking Tools

With the progress of research in formal verification techniques, several model checking tools have been developed. Some of the well-known tools are SMV (Symbolic Model Verifier) [McM92][SMV], a system developed at Carnegie-Mellon University; VIS (Verification Interacting with Synthesis) [BHSSAC96][VIS], a system developed at University of California, Berkeley;

FormalCheck (COSPAN) [FC97][FC], a system developed at Bell Labs; and BlackTie [BT] developed at Verplex Inc.

The model accepted by SMV is written in the SMV or SV (Synchronous Verilog) language, and the properties to be verified are expressed in CTL. SV is designed to allow the description of finite state systems ranging from completely synchronous to completely asynchronous, and from detailed to the abstract. SMV uses an efficient ROBDD-based symbolic model checking algorithm.

VIS is a tool that integrates the verification, simulation, and synthesis of finite-state hardware systems. VIS operates on an intermediate format called BLIF-MV. VIS includes a compiler from a synthesizable subset of Verilog to BLIF-MV. It supports CTL model checking, language emptiness checking for Büchi automata, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis. Multi-valued decision diagrams (MDDs) that are an extension of BDDs are used to represent the functions over multi-valued variables.

FormalCheck is a model checker based on language containment. This method requires that the description of the system and the properties be represented by ω-automata, and it verifies the correctness of the system by checking that the language of the system is contained in the language of the property. The reduction algorithms and the refinement methods embedded in FormalCheck make the tool applicable to industrial-size designs [XCSCLP99].

Unlike the other tools listed here, BlackTie doesn't use any temporal logic or specific language for writing properties. The properties are written in the Verilog Hardware Description Language, which can shorten the learning curve for hardware designers and make the formal verification an easy-to-use design methodology. Another benefit is that the properties written for formal verification

can be used in simulation to continually monitor the expected behavior. BlackTie also automatically generates checkers for some simple properties.

In the LASSO laboratory at Université de Montéal a formal verification tool called the MDG model checker has been developed [ZSCC94][ZSCCL95][CLCZS95] [XCSCM98][Xu99][HC00]. This tool supports combinational and sequential equivalence checking, and property checking. A model is represented by an abstract state machine that allows abstract variables and uninterpreted functions. This makes the model checker capable of verifying circuits with large data path. Sets of states and transition relations are symbolically represented by Multiway Decision Graphs (MDGs). Design models are described using the language MDG-HDL, and properties are expressed by formulas in L_{MDG}, a first-order ACTL logic. We will give more detail in Chapter 4. We are still continuing to develop and improve this tool.

Summary

The advantages of model checking techniques are that they can be made completely automatic, and when the verification fails an error trace is given to help the designers to locate bugs. Due to the state explosion problem, model checking has not been used widely in industry. Further research on solving this problem is continuing. In particular effective model reduction techniques can extremely alleviate this problem. To find efficient model reduction methods is the subject of this thesis. In the next chapter we will introduce the basic theory of model reduction and some of the techniques.

Chapter 3 Model Abstraction and Reduction

In the previous chapter we introduced the basic theory of model checking. The critical problem of this technique is state explosion. As the complexity of real systems continues to grow, model checking with symbolic representation still cannot handle many of them. Abstraction is probably the most important technique for reducing the state explosion problem, since it is performed even before the original model is constructed that might be too big to fit into memory. However, we must establish a relationship between the abstract model and the original model such that correctness at the abstract level implies correctness for the original system. It is important that the verification methodology does not lead to false positive results. If $M' \models P \Rightarrow M \models P$, we say that M' *weakly preserves* P, i.e., if M' satisfies property P, then so does M, but if M' does not satisfy P, M may or may not satisfy P. While if $M' \models P \Leftrightarrow M \models P$, we say that M' *strongly preserves* P, i.e., both the positive and the negative results of verifying P on M' can be carried to M. In this chapter we introduce some theoretical basis for model abstraction and reduction, and the related literature. This will be used to prove our reduction algorithms in Chapter 6.

3.1 Simulation and Bisimulation

Definition 1. A labeled transition system is a tuple $M = < S, S_0, I, T, AP, L >$, where S is a set of states, $S_0 \subseteq S$ is a set of initial states, I is a finite set of inputs, T: $S \times I \rightarrow S$ is a transition relation, AP is the set of atomic propositions, L: $S \rightarrow 2^{AP}$

is a labeling function indicating which propositions are true in each state. We use $s \rightarrow_a s'$ to denote there is a transition from state s to state s' when the input is a.

Definition 2. Given two labeled transition systems $M = < S, S_0, I, T, AP, L >$ and $M' = < S', S_0', I, T', AP', L' >$. M and M' have the same sets of inputs and $AP' \subseteq AP$. A relation $H \subseteq S \times S'$ is a simulation relation between M and M' if and only if for all $s \in S$ and $s' \in S'$, if $H(s, s')$ then the following conditions hold:

1. $L(s) \cap AP' = L'(s')$

2. For every state $s_1 \in S$ and $T(s, s_1)$, there is a state $s_1' \in S'$ such that $T'(s', s_1')$ and $H(s_1, s_1')$.

We say that M' simulates M if there exists a simulation relation H such that for every initial state s_0 in M there is an initial state s_0' in M' for which $H(s_0, s_0')$. Clarke et al. [CGL96] proved that if M' simulates M then for every ACTL* formula f, $M' \models f \Rightarrow M \models f$, i.e., properties expressed in ACTL* are weakly preserved by M'.

Definition 3. Given two labeled transition systems M and M' similar to the above, but let $AP' = AP$. A relation $B \subseteq S \times S'$ is called a bisimulation relation between M and M' if and only if for all $s \in S$ and $s' \in S'$, if $B(s, s')$ then the following conditions hold:

1. $L(s) = L'(s')$
2. For every state $s_1 \in S$ and $T(s, s_1)$, there is a state $s_1' \in S'$ such that $T'(s', s_1')$ and $B(s_1, s_1')$.
3. For every state $s_1' \in S'$ and $T'(s', s_1')$, there is a state $s_1 \in S$ such that $T(s, s_1)$ and $B(s_1, s_1')$.

If there is a bisimulation relation B between M and M', we say M and M' are bisimular if and only if B satisfies two additional conditions:

1. For each initial state s_0 in M there is an initial state s_0' in M' for which $B(s_0, s_0')$.

2. For each initial state s_0' in M' there is an initial state s_0 in M for which $B(s_0, s_0')$.

Clarke et al [CGL92] proved that if M and M' are bisimular, then for every CTL* formula f, $M \models f \Leftrightarrow M' \models f$, i.e., the properties expressed in CTL* are strongly preserved by M'.

Clarke et al [CGL96] presented general algorithms for determining whether two transition systems are simular or bisimular. These algorithms can handle both deterministic and nondeterministic transition systems. To check the simulation relation between two transition systems M and M', they defined a sequence of relations, H_0^*, H_1^*, ... on $S \times S'$ as follows:

1. $H_0^*(s, s')$ if and only if $L(s) \cap AP' = L'(s')$;

2. $H_{n+1}^*(s, s')$ if and only if

   – $H_n^*(s, s')$, and

   $-\forall s_1 \in S.\ [R(s, s_1) \Rightarrow \exists s_1' \in S'.\ [R'(s', s_1') \ \&\ H_n^*(s_1, s_1')]]$

The procedure will terminate since the transition systems are finite. There is an n such that $H_n^* = H_{n+1}^*$ and H_n^* is the largest simulation relation H^* between M and M'. Thus M' simulates M if and only if for every initial state s_0 in M there is an initial state s_0' in M' such that $H^*(s_0, s_0')$.

For checking the bisimulation relation between two transition systems M and M', a sequence of relations, B_0^*, B_1^*,... on $S \times S'$ are defined as follows:

1. $B_0^*(s, s')$ if and only if $L(s) = L'(s')$;

2. $B_{n+1}^*(s, s')$ if and only if

   – $B_n^*(s, s')$, and

   $-\forall s_1 \in S.\ [R(s, s_1) \Rightarrow \exists s_1' \in S'.\ [R'(s', s_1') \ \&\ B_n^*(s_1, s_1')]]$ and

   $-\forall s_1' \in S'.\ [R'(s', s_1') \Rightarrow \exists s_1 \in S.\ R(s, s_1) \ \&\ B_n^*(s_1, s_1')]]$.

This procedure will terminate when $B_n^* = B_{n+1}^*$. B_n^* is the largest bisimulation B^* between M and M'. If for every initial state s_0 in M there is an initial state s_0' in M' such that $B^*(s_0, s_0')$, and additionally for every initial state s_0' in M' there is an initial state s_0 in M such that $B^*(s_0, s_0')$, then we say M and M' are bisimular.

3.2 Abstract Interpretation

For many cases, the properties that we are interested in involve fairly simple relationships among the data values in the system. Additionally real systems generally manipulate data in a well-structured way. If we can abstract the concrete elements to a small number of abstract elements such that the abstraction will not affect the verification result, then the state explosion problem can be reduced.

For example, if we want to check that $x_1 * x_2$ is positive or negative, we need not perform the multiplication at the concrete level of the two numbers. The only thing we need to be concerned with is the sign of the result. Thus we can abstract the individual operands to their signs $\{neg, pos\}$ by a mapping h: if $x \leq 0$ then $h(x) = neg$, if $x \geq 0$ then $h(x) = pos$. Then apply the rules of signs for multiplication $\overline{*}$: $neg \ \overline{*} \ pos = neg$, $neg \ \overline{*} \ neg = pos$, $pos \ \overline{*} \ pos = pos$.

Given a transition system $M = (S, S_0, I, T, AP, L)$ with variables ranging over a set of values D, to construct a reduced system M', we select an abstract domain A and a mapping h from D to A. This determines a set of abstract atomic propositions AP'. For the example shown above, the abstract atomic propositions are $s = neg$ and $s = pos$. Normally the abstract-level propositions in M' are less than those in the concrete system M, thus the complexity of verification is reduced. $M' = (S', S_0', I, T', AP', L')$ is constructed as follows:

1. $S' = \{h(s) \mid s \in S\}$

2. $s' \in S_0'$ iff there exist s such that $s' = h(s)$ and $s \in S_0$

3. AP' is determined by the abstract mapping h as described above

4. $L'(s') = s'$

5. $T'(s', s_1')$ iff there exist s and s_1 such that $s' = h(s)$, $s_1' = h(s_1)$ and $T(s, s_1)$

We can see that each abstract state is a set of concrete states that have the same labeling of abstract atomic propositions. It is easy to see that the reduced system M' simulates the original M, and the mapping h introduces a simulation relation $H = \{(s, s') \mid s \in S \text{ and } s' = h(s)\}$ between the two systems [CGL96].

When the abstract mapping h satisfies some conditions, $H = \{(s, s') \mid s \in S \text{ and } s' = h(s)\}$ is a bisimulation relation between M' and M. For two states s_1 and s_2, the abstract mapping h induces an equivalence relation $\sim$: $s_1 \sim s_2$ iff $h(s_1) = h(s_2)$. If all these equivalence relations $\sim$ are congruences for the primitive relations corresponding to the basic operations used in the program that is $(s_1 \sim s_2 \rightarrow (P(s_1) \Leftrightarrow P(s_2)))$, then M and M' are bisimular [CGL92].

3.3 Symmetry Exploited in Model Checking

Real systems often exhibit considerable symmetry, for example, it is easy to find symmetry in memories, registers, bus protocols and network protocols which have a lot of replicated structures. [CFJ96] [CFJ93] [ID96] [Ip96] [ES93] use symmetry to reduce the state space in model checking. In general, they abstract the original system by a mapping according to the symmetry in the system.

Definition of symmetry groups: Let G be a group of permutations, i.e., bijective mappings acting on the state space S of the transition system M defined above. A permutation $\sigma \in G$ is said to be a symmetry for M if and only if it preserves the transition relation T. That is σ should satisfy the following condition:

$(\forall s_1 \in S)(\forall s_2 \in S)((s_1, s_2) \in T \Rightarrow (\sigma s_1, \sigma s_2) \in T)$

G is a symmetry group for M if and only if every permutation $\sigma \in G$ is said to be a symmetry for M.

In the example shown below, the permutation σ is defined as: $s_0 \rightarrow s_0$, $s_1 \rightarrow s_2$, $s_2 \rightarrow s_1$, $s_3 \rightarrow s_3$. σ exchanges the states s_1 and s_2, but s_0 and s_3 are not affected. It is obvious that the transition remains the same. Hence σ is a symmetry of M.

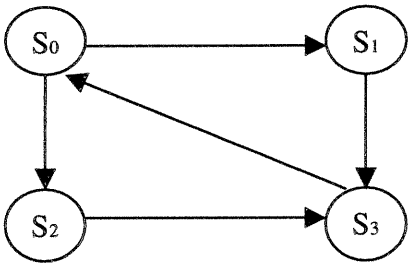


Figure 1. A symmetry example

If s is an element of S, then the orbit of s is the set

$$\theta(s) = \{t \mid (\exists \sigma \in G)(\sigma s = t)\}$$

Each orbit $\theta(s)$ is represented by $rep(\theta(s))$, a representative selected from it. The quotient model $M_G = (S_G, S_{G0}, I, T_G, AP, L_G)$ of M and G is defined as follows:

1. The state set is $S_G = \{\theta(s) \mid s \in S\}$, the set of the orbits of the states in S;
2. The transition relation T_G is $(\theta(s_1), \theta(s_2)) \in T_G$ *iff* $(s_1, s_2) \in T$;
3. The labeling function is $L_G(\theta(s)) = L(rep(\theta(s)))$.

G is an invariance group for an atomic proposition p if and only if the set of states labeled by p is closed under the application of all the permutations of G. That is the following condition holds:

$$(\forall \sigma \in G)(\forall s \in S)(p \in L(s) \Leftrightarrow p \in L(\sigma s))$$

Given a labeled transition M and a symmetry group G, if G is an invariance group for all the atomic propositions p occurring in a *CTL** formula f, then [CFJ96]

$$M, s \models f \Leftrightarrow M_G, \theta(s) \models f$$

The complexity of orbit calculations is as hard as the Graph Isomorphism problem. If a system has N equivalent components, i.e., N instances of one module, and each equivalent component has m state variables, the lower bound for the BDD representing the induced orbit relation θ is $2^{K/8}$ with $K = min(N, 2^m)$ [CFJ96].

Since the exponential complexity of computing the orbit relation, exploiting these types of symmetries in symbolic model checking is restricted to examples with a small number of components. An approach to avoid the computation of the orbit relation is given in [CFJ96], in which any subset instead of a unique state of an orbit can be used to represent this orbit. The image and preimage are computed in terms of the representatives instead of using the transition relations of the quotient model. That is, we can perform model checking on the quotient system without explicitly building the quotient model. We use *Img* to denote the image computation, and *Pre* to denote the preimage computation. For a set of states S, the image and preimage of S are defined as follows:

$$Img(S) = \{s' \mid \exists s \in S\ T(s, s')\}$$

$$Pre(S) = \{s \mid \exists s' \in S\ T(s, s')\}$$

A similar approach is presented by Ip and Dill [ID96] [Ip96]. In their work they also propose a new data type *scalarset* that was added to the description language to detect and exploit symmetries in the finite state system. A scalarset can only be accesssed through restricted set of operations that guarantee certain symmetries to hold on the state graph. SMV and Murφ verification tools adopt this reduction technique based on symmetry and scalarset data type.

3.4 Partition Refinement Methods

A *partition* ρ of a set S is a set of pairwise disjoint subsets of S whose union is all of S. The elements of ρ are called its blocks. If $\rho = \{B_1,\ldots, B_n\}$ and $\rho' = \{B_1',\ldots, B_n'\}$ are partitions of S, we say that ρ' is a *refinement* of ρ if and only if :

$$\forall B_i' \in \rho',\ \exists B_i \in \rho .\ (B_i' \subseteq B_i)$$

For a given transition system $M = \langle S, S_0, I, T\rangle$ and a partition $\rho = \{B_1,\ldots, B_n\}$ of S, we define a *quotient system* $M' = \langle S', S_0', I, T'\rangle$ as follows:

1. $S' = \{B_1,\ldots, B_n\}$, that is, every state in M' is a block in the partition ρ.
2. $\forall s_0' \in S_0',\ \exists s_0 \in S_0 .\ (s_0 \in s_0')$.
3. $\forall a \in I .\ (s_1' \xrightarrow{a} s_2' \in T' \Leftrightarrow (\exists s_1 \in s_1',\ \exists s_2 \in s_2' .\ (s_1 \xrightarrow{a} s_2 \in T)))$.

A transition $s_1' \xrightarrow{a} s_2'$ in the quotient system is *stable* if and only if the following condition is satisfied:

$$\forall s_1 \in s_1' .\ ((s_1 \xrightarrow{a} s_2 \in T) \wedge (s_2 \in s_2'))$$

That is, the transitions of each state in the block s_1' guarded by input a will lead to a state in the same block s_2'. The quotient system is stable if and only if all of its transitions are stable.

Let $\overline{\rho} = \{B_1,\ldots, B_n\}$ be a partition of the states in the transition system M. If the quotient system M' constructed by $\overline{\rho}$ is stable, M' is a bisimular of M [BFH90] [Fer90] [LY92].

Lee and Yannakakis [LY92] present an algorithm that computes bisimulations only on the reachable state space and computes the equivalence classes of the bisimulation relation rather than the bisimulation relation itself. The algorithm starts with an initial partition, then splits the blocks until there are no unstable arcs. This algorithm explores the graph and splits blocks simultaneously, combining the

forward inference of reachability information with the backward inference of equivalence information. The algorithm in [FV98] is a modification of [LY92] which adds a holding set to reduce the number of unreachable blocks retained during the processing. A holding set includes several unreachable blocks and is treated as a single equivalence class which improves memory usage.

Summary

In this chapter, we introduced the theoretical basis for model reduction. If a model M' simulates a model M, then for every ACTL* formula f, $M' \models f \Rightarrow M \models f$. If a model M' bisimulates a model M, then for every CTL* formula f, $M' \models f \Leftrightarrow M \models f$. The techniques of abstract interpretation, reductions based on symmetry, and partition refinement methods were also introduced. Not all these techniques are totally automatic and efficient. Usually in abstract interpretation and symmetry reduction, users need to provide some information to guide the verification tools. New methods that are more efficient and friendly are needed. We intend to develop some new reduction methods that can be implemented in MDG model checking. In the next chapter we will introduce MDG model checking. We will see why many existing methods cannot be adopted in this tool.

Chapter 4 Multiway Decision Graphs (MDGs) and Model Checking

In the previous chapters we introduced model checking and techniques to solve its critical problem of state explosion. With the appearance of ROBDD-based symbolic model checking techniques, the useful domain of model checking was increased considerably. Since ROBDD-based model checking requires a binary representation of the circuit, every individual bit of data signals must be represented by one boolean variable. When a circuit has a large datapath width, ROBDD-based verification methods may take too long or run out of memory. The reduction methods based on abstract interpretation and data symmetries take long time to compute the bisimulation relation. Do we have other methods to handle circuits with large data?

Corella, Cerny, Zhou and Song et al [CLCZS95][ZSCCL95][ZSCC94] developed new techniques based on the use of abstract variables to represent data and uninterpreted function symbols to represent data operations. These techniques can handle data path feedback and can verify interactions between data paths and control paths when data operations can be viewed as black boxes, i.e., the correctness does not depend on the meaning of those operations. As a consequence, Multiway Decision Graphs (MDGs) that incorporate abstract variables and uninterpreted functions were developed to represent and manipulate sets of states and transition relations. The technique of implicit state enumeration in the Boolean domain where ROBDDs are used was lifted to the domain of

abstract sorts where MDGs are used. The higher level of abstraction in MDGs makes it possible to verify circuits with large datapath widths.

4.1 A Many-sorted First-order Logic

Syntax

The formal logic used in MDG is a many-sorted first-order logic with a distinction between abstract sorts and concrete sorts. Concrete sorts can be enumerated finitely, while abstract sorts do not have an enumeration. The enumeration of a concrete sort α is a set of distinct constants of sort α. The constants occurring in enumerations are referred to as individual constants, while the constants of abstract sorts are referred to as generic constants. Variables of concrete sorts are used for representing control signals, and variables of abstract sorts are used for representing datapath signals.

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 $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, where $\alpha_1 \ldots \alpha_{n+1}$ are sorts. If α_{n+1} is an abstract sort then f is an abstract function symbol. Abstract function symbols are useful for modeling data operations of which we know the implementation to be correct. 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. Cross-operators are useful for modeling feedback signals from the datapath to the control circuitry. Both abstract function symbols and cross-operators can be uninterpreted or partially interpreted by conditional rewriting rules.

The function symbol $f(A_1, \ldots, A_n)$ may be structured which means $A_1, \ldots, A_n$ can be function symbols. We call structured function symbols as *terms* which are defined inductively as follows: a constant or a variable of sort α is a term of sort α; 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 that has no concrete subterms other than individual constants is said to be concretely-reduced. 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 a cross-term. An equation is an expression "$A_1 = A_2$" where A_1 and A_2 are terms of the same sort. The atomic formulas are the equations plus T (truth) and F (falsity). The formulas of the logic are built from the atomic formulas in the usual way using logical connectives and quantifiers.

Semantics

An interpretation is a mapping ψ that assigns a denotation to each sort, constant and function symbol, satisfying 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 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 $\psi(\alpha_{n+1})$. In particular, if $n = 0$, i.e., f is a generic constant of sort α_1, $\psi(f) \in \psi(\alpha_1)$.

Let X be a set of variables, a variable assignment with domain X compatible with an interpretation ψ is a function ϕ that maps every variable $x \in X$ of sort α to an element $\phi(x)$ of $\psi(\alpha)$. We write ${\phi^{\psi}}_X$ for the set of ψ-compatible assignments to the variables in X. The denotation of a term and the truth or falsity of a formula under an interpretation and a compatible variable assignment are defined as usual. We write $\psi, \phi \models P$ if a formula P denotes truth under an interpretation ψ and a ψ-compatible variable assignment ϕ to the variables in P, $\psi \models P$ if $\psi, \phi \models P$ for all such assignments ϕ, and $\models P$ if $\psi \models P$ 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 $\models P \Rightarrow Q$.

4.2 Multiway Decision Graph (MDG)

An 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. An MDG G represents a formula defined inductively as follows:

1. If G consists of a single leaf node labeled by a formula P which can only be T or F, then G represents P;
2. 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 $\bigvee_{1 \le i \le n} ((A = B_i) \wedge P_i)$.

MDGs satisfy a set of well-formedness conditions to turn MDGs into canonical representations that can be manipulated by efficient algorithms. The formulas represented by MDGs are a subset of the many-sorted first-order logic introduced in the previous section. We refer to these formulas as *Directed Formulas* (DFs).

A directed formula (DF) is a disjunction of conjunctions of equations. A well-typed equation is an expression "$A_1 = A_2$" where A_1 and A_2 are terms of the same sort. Given two disjoint sets of variables U and V, a directed formula is of type $U \rightarrow V$ if and only if (1) each equation is well-typed; (2) every abstract variable in V appears in the left-hand-side of the equations; (3) the abstract variables in U appear in the right-hand-sides of the equations or in the cross-terms in the left-hand-sides; (4) in each disjunct, the left-hand-sides of the equations are pairwise distinct. In a DF of type $U \rightarrow V$, V is referred to as the primary variables and U is referred to as the secondary variables. Abstract primary variables label MDG nodes, while secondary abstract variables label edges or appear as function arguments. Concrete variables can only label nodes. Two directed formulas are equivalent if and only if their MDG representations are isomorphic. In the

following we also use DFs to represent MDG graphs, and we do not make the distinction between them.

Similar to ROBDDs, MDGs can represent transition relations and sets of states. Since a variable assignment ϕ with domain V compatible with an interpretation ψ can be viewed as a vector of values, indexed by the variables in V, ${\phi^{\psi}}_V$ can be viewed as the cartesian product of the indexed family of sets $(\psi(\alpha_v))_{v \in V}$, where α_v is the sort of v. For a given interpretation ψ, a directed formula P of type $U \rightarrow V$ can be used to represent the transition relation $\{\phi \in {\phi^{\psi}}_{U \cup V} \mid \psi, \phi \models P\}$, or to represent the set of states $Set^{\psi}(P) = \{\phi \in {\phi^{\psi}}_V \mid \psi, \phi \models (\exists U)P\}$.

The basic operations of MDG are disjunction, conjunction, existential quantification, relational product (*RelP*), and pruning by subsumption (*PbyS*). Relational product *RelP*(MDGs, Vars, η) takes as arguments a set of MDGs, a set of variables *Vars*, and a variable renaming substitution η. It computes the conjunction of the MDGs, removes the variables by existential quantification, and applies the substitution. Since abstract variables occur in MDG, for the MDG ($x = c$) where x is an abstract variable, and c is a generic abstract constant, there is no MDG representing $\neg(x = c)$. Thus there is no negation operation in MDG. Then there is no complement operation in MDG. In MDG there is *PbyS* operation which approximates the difference of sets represented by MDGs. *PbyS*(G, H) takes two MDGs as arguments: G and H of type $X \rightarrow Y_1$ and $X \rightarrow Y_2$ respectively, and produces an MDG G' of type $X \rightarrow Y_1$. Suppose that the DF of G is of the form: $D_1 \vee \ldots \vee D_n$, and the DF of H is of the form: $B_1 \vee \ldots \vee B_n$. If there is a substitution θ with domain less than or equal to Y_2, we apply this substitution to H, If there exist D_i and $\theta(B_j)$ such that $D_i \wedge \theta(B_j) = D_i$ then we say D_i is subsumed by B_j. When doing the *PbyS*(G, H) operation, we remove every D_i which is subsumed by a disjunct of H from the DF of G, which is so-called pruning by subsumption. G' satisfies the relation: $\models G' \vee (\exists X)\, H \Leftrightarrow G \vee (\exists X)\, H$. Since an MDG represents a set, for every interpretation ψ, $(Set^{\psi}(G) \setminus Set^{\psi}(H) \subseteq Set^{\psi}(G') \subseteq Set^{\psi}(G)$.

4.3 An Abstract Description of State Machines and State Enumeration

An abstract description of a state machine (ASM) is a tuple $D = (X, Y, Z, Y', \eta, F_I, F_T, F_O)$ where:

1. X, Y and Z are pairwise disjoint sets of input, state and output variables. They can be of abstract sorts.
2. Y' is the set of next state variables, disjoint from $X \cup Y \cup Z$, and η is the function that maps each state variable to the corresponding next state variable. For convenience, we use the primed symbol v' to represent the next state variable of state variable v.
3. F_I is an MDG of type $U_0 \rightarrow Y$ that represents the set of initial states, where U_0 is a set of abstract variables disjoint from $X \cup Y \cup Z \cup Y'$.
4. F_T is an MDG of type $(X \cup Y) \rightarrow Y'$ that represents the transition relation.
5. F_O is an MDG of type $(X \cup Y) \rightarrow Z$ that represents the output relation.

For an interpretation ψ, there is only one state machine $M = (\phi^{\psi}_X, \phi^{\psi}_Y, \phi^{\psi}_Z, S_I, R_T, R_O)$ satisfying the description D. Since ϕ^{ψ}_X is the set of all ψ-compatible assignments to the variables in X, i.e., the set of all input vectors, thus it is the input alphabet of the state machine. Similarly ϕ^{ψ}_Y is the set of states of the machine and ϕ^{ψ}_Z the output alphabet.

1. $S_I = Set^{\psi}(F_I)$ is the set of initial states.
2. $R_T = \{(\phi, \phi', \phi'') \in \phi^{\psi}_X \times \phi^{\psi}_Y \times \phi^{\psi}_Y \mid \psi, \phi \cup \phi' \cup (\phi'' \circ \eta) \models F_T\}$ is the transition relation.
3. $R_O = \{(\phi, \phi', \phi'') \in \phi^{\psi}_X \times \phi^{\psi}_Y \times \phi^{\psi}_Z \mid \psi, \phi \cup \phi' \cup \phi'' \models F_O\}$ is the output relation.

Model checking is based on state enumeration. Here we show how reachability analysis can be performed on the abstract description $D = (X, Y, Z, Y', \eta, F_I, F_T, F_O)$ in MDG. The algorithm is shown in the following:

ReAn (D)

Initially $R := F_I$; $Q := F_I$; $K := 0$;

Loop

$K := K+1$;

$I := Fresh(X, K)$;

$N := RelP(\{I, Q, F_T\}, X \cup Y, \eta)$;

$Q := PbyS(N, R)$;

If $Q = F$ then return;

$R := PbyS(R, Q)$;

$R := Disj(R, Q)$;

End loop;

End *ReAn*

The variable K is the loop counter that is used to generate fresh abstract variables to denote values of abstract data inputs at the k-th iteration. $Fresh(X, K)$ builds a DF representing a conjunction of equations $v = v\#K$, one for each abstract input variable $v \in X$, where $v\#K$ is a fresh variable disjoint from $X \cup Y \cup Z \cup Y'$. Then $I := Fresh(X, K)$ represents the set of input vectors. Before the k-th iteration, $k > 0$, R represents the set of states reachable in less than k steps, and Q represents the frontier set, a subset of R containing at least all the states newly reached in the previous iteration. At the beginning, both R and Q are the initial states. At the K-th iteration, $RelP(\{I, Q, F_T\}, X \cup Y, \eta)$ computes the set of states N that can be reached in one transition from Q with input I. Then, $PbyS(N, R)$ computes the frontier set Q. If $Q = F$ then there are no new states reached, $R_{k-1} = R_k$, i.e., the fixpoint is reached, and the reachability analysis finishes. If $Q \neq F$, the set of states reachable in K iterations is computed. This is done by $R := PbyS(R, Q)$; $R := Disj(R, Q)$. Since Q was not computed earlier as an exact difference, R may

contain some disjuncts that are subsumed by Q. Removing these disjuncts from R before taking the disjunction of R and Q often produces a smaller resulting MDG.

4.4 The Specification Language L_{MDG}

In MDG model checking, the properties to be verified are expressed by formulas in L_{MDG} that defines a 1st-order branching-time temporal logic, a subset of which can be verified using our MDG model checker [Xu99][XCSCM98]. The atomic formulas of L_{MDG} are the constants True or False and equations of the form $t_1 = t_2$, where t_1 is an ASM-variable, t_2 is an ASM-variable, a constant, an ordinary variable or a function of ordinary variables. An ASM-variable is a variable appearing in the description of an ASM. An ordinary variable is not an ASM variable and is used to remember the past value of an ASM-variable in the specification of a property. If p, q are L_{MDG} formulas, then so are $!\, p$, $p \,\&\, q$, $p \mid q$, $p \rightarrow q$, **LET** $(v = t)$ **IN** p, **X**p, **A**p, **AG**p, **AF**p, **A**$(p$ **U** $q)$, **AG**$(p \rightarrow$ **F**$(q))$, **AG**$(p \rightarrow (q$ **U** $r))$. In the formula **LET** $(v = t)$ **IN** p, v is an ordinary variable and t is an ASM-variable. **LET** $(v_1 = t_i)$ &...& $(v_n = t_n)$ **IN** p is a shorthand for **LET** $(v_1 = t_i)$ **IN** ((**LET** $(v_1 = t_i)$ **IN** (...**LET** $(v_1 = t_i)$ **IN** p))). In L_{MDG} the existential path quantifier **E** is not allowed and negation can only be applied to propositional formulas over concrete variables. The semantics are defined on an abstract computation tree.

A path π in an abstract computation tree is an infinite sequence of states s_0, s_1, ... such that $R(s_i, s_{i+1})$ for $i \geq 0$. We use π_i to denote the suffix of π starting at s_i, $s, \sigma \models p$ to mean that the formula p is true at state s; $\pi, \sigma \models p$ to mean that the formula p is true on path π, and $Val_{\phi \cup \sigma}(t)$ to denote the value of term t under a ψ-compatible assignment ϕ to state, input and output variables and a ψ-compatible assignment σ to ordinary variables. The relation $\models$ is defined inductively as follows:

$s, \sigma \models t_1 = t_2$ iff $Val_{s \cup \sigma}(t_1) = Val_{s \cup \sigma}(t_2)$;

$s, \sigma \models ! p$ iff it is not the case that $s, \sigma \models p$;

$s, \sigma \models p \,\&\, q$ iff $s, \sigma \models p$ and $s, \sigma \models q$;

$s, \sigma \models p \mid q$ iff $s, \sigma \models p$ or $s, \sigma \models q$;

$s, \sigma \models p \rightarrow q$ iff $s, \sigma \models ! p$ or $s, \sigma \models q$;

$\pi, \sigma \models p$ iff $s, \sigma \models p$ and s is the first state of p;

$\pi, \sigma \models ! p$ iff it is not the case that $\pi, \sigma \models p$;

$\pi, \sigma \models p \,\&\, q$ iff $\pi, \sigma \models p$ and $\pi, \sigma \models q$;

$\pi, \sigma \models p \mid q$ iff $\pi, \sigma \models p$ or $\pi, \sigma \models q$;

$\pi, \sigma \models p \rightarrow q$ iff $\pi, \sigma \models ! p$ or $\pi, \sigma \models q$;

$\pi, \sigma \models \mathbf{X}\, p$ iff $\pi_1, \sigma \models p$;

$\pi, \sigma \models \mathbf{G}\, p$ iff $\pi_j, \sigma \models p$ for all $j \geq 0$;

$\pi, \sigma \models \mathbf{F}\, p$ iff $\pi_j, \sigma \models p$ for some $j \geq 0$;

$\pi, \sigma \models p\ \mathbf{U}\ q$ iff for some $k \geq 0$, $\pi_k, \sigma \models q$ and $\pi_j, \sigma \models p$ for all $k \geq j \geq 0$;

π , $\sigma \models$ **LET** $(v = t)$ **IN** p iff $\pi, \sigma' \models p$, where $\sigma' = (\sigma \setminus \{(v, \sigma(v))\}) \cup \{(v, Val_{s0 \cup \sigma}(t))\}$.

A property in L_{MDG} holds on an ASM if and only if the property is true for all the paths starting from the initial states in the abstract computation tree.

In the following, we use a memory as an example to illustrate how to use L_{MDG} to express properties.

Example 1: A memory unit cannot be read (*read* = 1) and written (*write* = 1) at the same time.

AG (! ((*read* = 1) & (*write* = 1)));

Example 2: After the *read* signal is set, the content *data10* of the memory unit at the specified *address* 10 will be fetched after one clock cycle.

AG (**LET** *v* = *data10* **IN** ((*read* = 1 & *address* = 10) → **X** (*data_out* = *v*)));

4.5 Construction of an ASM for the Property in L_{MDG}

In the MDG system, model checking is carried out as folllows: an additional ASM is built for a property represented by a L_{MDG} formula, composed with the original ASM of the design, and finally the appropriate algorithm is applied to verify a transferred property on the composite machine. Thus the verification of the original property on the original machine is changed to the verification of the transferred property on the composite machine. The transferred property is **AG**(*flag* = 1) for **AG**(*Next_let_formula*), **AF**(*flag* = 1) for **AF**(*Next_let_formula*), **A**((*flag*1 = 1) **U** (*flag*2 = 1)) for **A**((*Next_let_formula1*) **U** (*Next_let_formula2*)), where *flag*, *flag*1 and *flag*2 are boolean state variables of the additional ASM generated from the original property.

Given a design which is represented by an ASM $M_D = (X_D, Y_D, Z_D, \eta_D, F_{ID}, F_{TD}, F_{OD})$, and a property P to be verified, we construct the additional ASM $M_P = (X_P, Y_P, Z_P, \eta_P, F_{IP}, F_{TP}, F_{OP})$ for the property expressed in L_{MDG}. The input variables of M_P are the ASM-variables of M_D that appear in the property, i.e., $X_P \subseteq X_D \cup Y_D \cup Z_D$. They represent the values at the current clock cycle. Let n be the maximum nesting number of **X** operators in the property. The set of state variables Y_P and the transition relation represented by the DF F_{TP} are constructed to remember the values of the input variables of M_P or the results of a comparison of the variables in the past n (or less than n) clock cycles. There are special state variables *flag*, *flag*1, *flag*2 in M_P which are used to indicate the truth values of the Next_let_formula one clock cycle earlier. There is no output from M_P, so there is no output relation either, i.e., $Z_P = \varnothing$, $F_{OP} = \varnothing$. The initial values of the state variables in M_P are set in such a way that they do not affect the result of verifying P on the original ASM.

After the additional ASM M_P is constructed, the composite machine M of the original ASM M_D and M_P is constructed. Let $M = (X, Y, Z, \eta, F_I, F_T, F_O)$ where

1. X is the set of the input variables of M, $X = X_D$.
2. Y is the set of state variables of M which contains both the variables in Y_D and Y_P, $Y = Y_D \cup Y_P$. The state space of M is a subset of $\Phi^{\Psi}_{Y_D} \times \Phi^{\Psi}_{Y_P}$ under each interpretation ψ, because M is a composite machine in which the states of M_P are derived from M_D, rather than the product machine of M_D and M_P.
3. Z is the set of the output variables of M_D, $Z = Z_D$.
4. F_I is a DF representing the set of initial states of M, $F_I = F_{ID} \wedge F_{IP}$.
5. F_T is a DF representing the transition relation of M, $F_T = F_{TD} \wedge F_{TP}$.
6. F_O is a DF representing the output relation of M, $F_O = F_{OD}$.
7. η is the function that maps each state variable of M to the corresponding next state variable, $\eta = \eta_D \cup \eta_P$.

For example, we construct the additional ASM (Figure 2) for the property of example 2 in section 4.4. The boldly lined components in Figure 2 are flip-flops.

AG (**LET** v = *data10* **IN** ((*read* = 1 & *address* = 10) → **X** (*data_out* = v)));

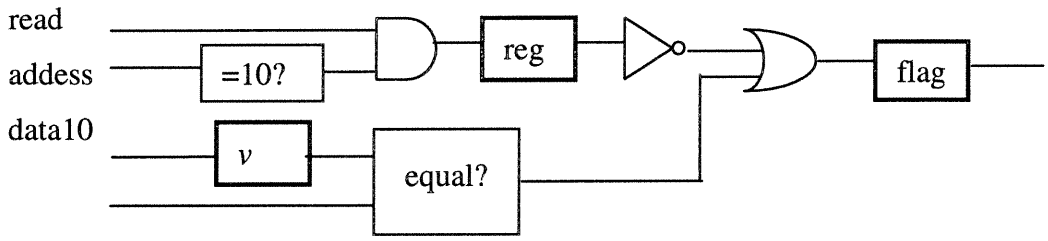


Figure 2 : An example of additional ASMs

4.6 Reduction Problems in MDG Model Checking

Although MDG model checking can use abstract variables and uninterpreted function symbols to represent sets of states and transition relations that enlarges the useful domain of MDG, the state explosion problem is still a bottleneck that prevents MDG from handling many real systems. Our work is to alleviate this problem and make MDG a practical verification tool.

As introduced in Chapter 1 and Chapter 3, there are many techniques to approach the state explosion problem. The techniques that attract us most are those based on model abstraction and reduction. If we can find an abstract model that normally is smaller than the original model, i.e., a reduced model, and this model simulates or bisimulates the original model, then we can verify properties on the abstract one. If an abstract model simulates the original model then ACTL properties are weakly preserved, while if an abstract model bisimulates the original one then any CTL* property is strongly preserved and we never need to use the original model in this situation, i.e., the state explosion problem can be reduced.

However, in general bisimulation equivalence can be verified in O(mn) [Mil80] for a labeled transition system with m transitions and n states which is exponential with the number of components in the design. For a large circuit, the method to compute a bisimulation relation is not feasible. Most of the solutions to avoid this computation are based on preimage and postimage computations. As mentioned earlier, MDG has no complement operation due to the presence of abstract variables and uninterpreted functions and no conjunct operation of two MDGs having the same abstract primary variables, and thus cannot compute preimages. All the methods that are based on preimage computation cannot be applied to MDG.

Symmetry reduction is often used on the systems that have a collection of components that may be replicated n times, such as symmetrically selectable

registers or individual bits in a word. These systems can often be proved correct without modeling the precise number of replicated components. Since MDG allows abstract variables, a datum with n bits can be represented by a variable *datan* of abstract sort. Thus bit symmetry reduction is not needed in MDG. Moreover, symmetry reduction is also based on computation of equivalence relations on states, and the efficient algorithms as presented in [CFJ96][ID96][IP96] are still based on preimage computation, thus they cannot be applied to MDG.

For the partition refinement methods introduced in [LY92][FV98][Dam96] [DGG93], preimage computation is also used, hence those methods cannot be used in MDG. In fact not all the methods are efficient even in the case of concrete variables. For example, the splitting method in [Dam96] [DGG93] splits states and then computes the transition relations of the refined model. It requires n^n computations of abstract transition relations for constructing an abstract model with n states. This makes the algorithm impractical even after state encoding. As reachability analysis cannot be done on the original model, computation of the abstract transition relations is also done for the unreachable states, which may be costly. Second, in MDG the original property is transferred to the circuit model as an additional state machine, the transferred property itself is thus very simple. The companion set becomes trivial and does not contain much information for carrying out reductions. This is also the case when the property is directly encoded in the circuit, i.e., saying that we should analyze the property first before constructing the additional state machine would not help.

Many reduction methods are not fully automatic. They need users to give a mapping between the concrete model and the abstract model. Only then they can construct the abstract model automatically. In some symmetry methods [ID96][Ip96], the user also needs to define the components that can be symmetrically reduced as *scalarset* to guide the tool to complete the reduction.

What we want is to provide users with an efficient and friendly verification tool that can do automatic reductions. We have to find some suitable reduction methods for improving our MDG model checker.

Summary

In this chapter we introduced Multiway Decision Graphs (MDG) and model checking based on MDG. The models under verification are represented by abstract state machines that may contain abstract variables and uninterpreted function symbols. This makes MDG model checking suitable for verifying the interaction between datapath and the control. However, the state space explosion problem critically limited the useful domain of MDG model checker. In this thesis we present two model reduction methods in the following chapters and have integrated them in MDG model checker and made this tool capable of verifying real industry designs. One is based on the topology of the circuits under verification and is discussed in Chapter 5. The other method is based on the functional dependency and is discussed in Chapter 6 and Chapter 7.

Chapter 5 Model Reductions Based on Circuit Topology

In the previous chapter we have introduced MDG model checking. Due to the particular characteristics of MDG, many existing reduction methods cannot be adopted in MDG model checking. From observation we know that many properties only specify the behavior of a part of a circuit under verification. If we can only use this part of a circuit to prove the property, then we may avoid the state explosion that may happen when using the whole circuit. In this chapter we present an automatic method based on circuit topology to find the sufficient model for verifying a given property. Furthermore we present am improved method based on the fanin information of gates.

5.1 A Reduction Algorithm Based on Circuit Topology

In the MDG model checker, the property to be verified is first transferred to an additional circuit and this circuit is connected to the original design circuit. Then the verification of the original property on the original design is transferred to the verification of the simplified property on the composite machine. For example, the property of the form **AG**(*Next-let-formula*) is transferred to the simplified property *P*: **AG**(*flag* = 1). The simplified property verifies the values of some *flag* signals. If we know which parts of the circuit influence the values of the flags, we can use only those parts to verify the simplified property. Intuitively, those parts of circuit in which signals propagate to the flags may influence their values. In the following

we will present an automatic method to find the parts of circuit that may influence the flags.

We start from the signals of the flags in the additional circuit representing the property to be verified, and search back the circuit in the reverse direction of signal propagation. If a signal or a component is reached during the searching, it is marked as "reached". The searching process terminates when it reaches the primary inputs or previously reached components. After the searching process terminates, the reached components of the circuit are those whose values propagate to the flags and thus may influence their values. The components not marked as "reached" cannot influence the values of the flags. We can use the part of the circuit containing all the reached components to verify the property, and remove the unreachable components. We call the so-constructed reduced system a *sufficient* model for P. Obviously the *sufficient* model strongly preserves property P.

The circuit that is unshaded in Figure 3 is a MinMax machine [CZSLC97] that has 2 input variables $X = \{r, x\}$ and 3 state variables $Y = \{c, rm, rM\}$, where r and c are of the boolean sort, and x, rm, and rM are of an abstract sort s. rm stores the smallest value of input x, and rM stores the greatest value of input x since last reset. When the machine is reset, rm is loaded by the maximal possible value *max* and rM is loaded by the minimum possible value *min*. Here *max* and *min* are generic constants. The smallest and greatest values are computed using an operator *leq*. *leq* is an uninterpreted cross-operator of type $s \times s \rightarrow B$. If $leq(a, b) = 1$, we say a is less than or equal to b.

We want to verify that if reset is on ($r = 1$) then rm will be loaded by the generic constant *max* in the next clock cycle. This property is expressed in L_{MDG} as follows:

$$P\text{: } \mathbf{AG}((r = 1) \rightarrow \mathbf{X}(rm = max));$$

The additional circuit constructed for this property is shown in the shaded area of Figure 3. We thus verify *P*': **AG**(*flag* = 1) on the resulting composite machine.

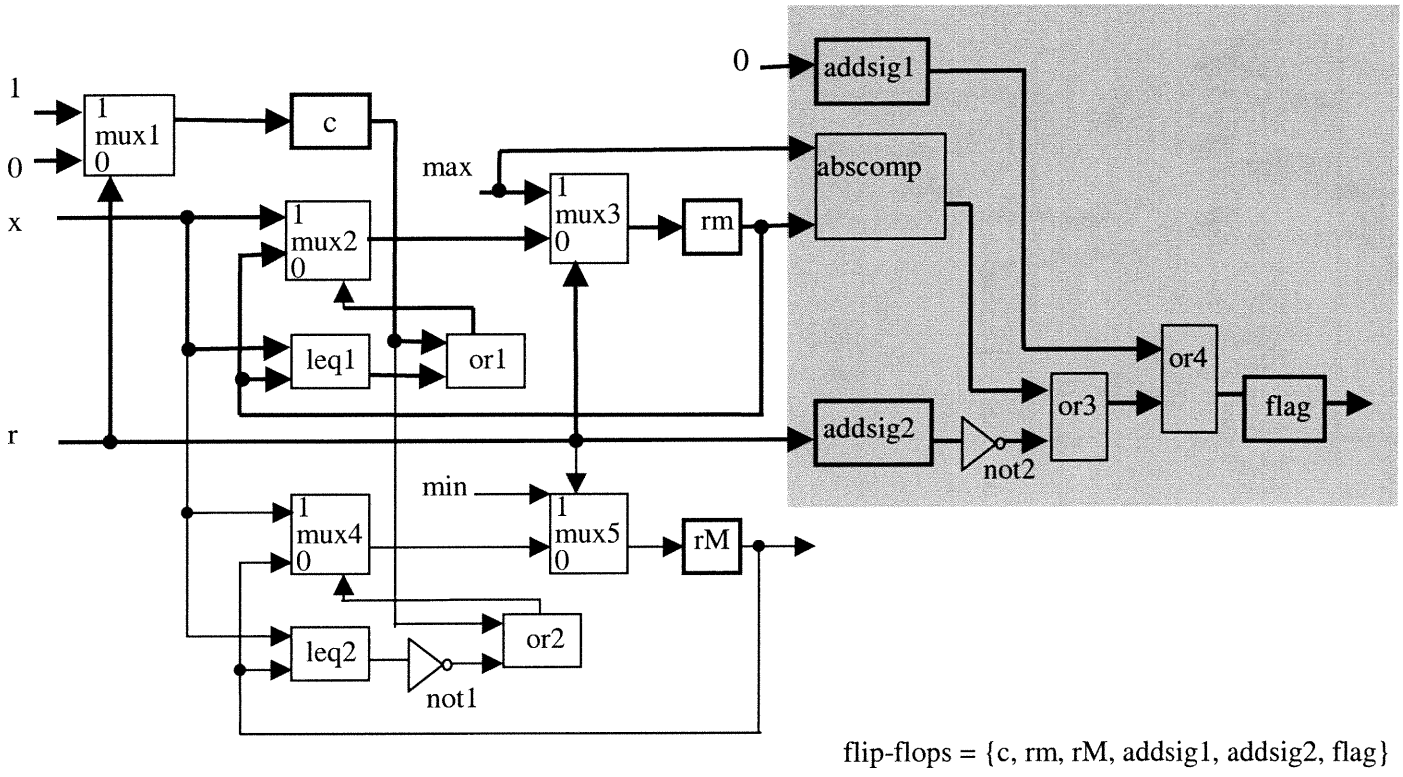


Figure 3. The MinMax machine and the additional circuit for P

Now, we start from *flag* and search the circuit in the reverse direction of signal propagation to find which parts of the circuit influence the value of *flag*. The thick lines in Figure 3 show the signals that propagate to *flag*. We find that the components {*or*4, *or*3, *not*2, *addsig*1, *abscomp*, *addsig*2, *rm*, *mux*3, *mux*2, *or*1, *leq*1, *c*, *mux*1} are reached, and the components {*mux*4, *mux*5, *leq*2, *or*2, *rM*, *not1*} are not reached. Since only the reached components influence the value of *flag*, thus only these parts are used to verify **AG**(*flag* = 1). The outputs of flip-flops are state variables. The property is verified on a reduced system that consists of the state variables {*flag*, *addsig*1, *addsig*2, *rm*, *c*}, and the state variable *rM* is removed.

The procedure *Reduced_model_checking(M, P)* shown below automatically finds the reachable parts of the circuit *M* according to the property *P* using the subprocedure *Find_reachable_part*, and removes the unreachable part of *M* from the transition system by the subprocedure *Remove_unreachable_part*. It constructs the transition model from the reduced circuit by the subprocedure *Construct_transition_relations*, and then verifies *P* on the reduced model using *Modelcheck*. Here *M* is the circuit description of the composite machine and *P* is the simplified property.

```
Reduced_model_checking(M, P)
Begin
   reached_part := Find_reachable_part(flags, M);
   M_S := Remove_unreachable_part(M, reached_part);
   T := Construct_transition_relations(M_S);
   Return Modelcheck(T, P);
End;

Find_reachable_part(flags, M)
Begin
    signals := flags;
    reached := ∅;
    While signals ≠ ∅;
     Begin
        reached := signals ∪ reached;
        comp := Find_component_connected_to_signals(signals, M);
        if comp ≠∅
        Begin
            reached := comp ∪ reached;
```

```
            fanins := Find_fanins(comp, M);
            signals := fanins \ reached;
        End;
        Else Return reached;
    End;
    Return reached;
End;
```

Figure 4. The reduction algorithm based on circuit topology

5.2 An Iterative Reduction Algorithm Considering the Fanins of Gates

For large designs, even if we use only the *sufficient* part of the circuit to construct the reduced transition system that influences the values of the flags that need to be verified, state explosion may still occur. In many cases the property can be verified on an even-more reduced model. We now consider the gates with multiple fanins that frequently appear in circuits. For the circuit shown below, the components {*flag*, r_0, r_1, r_2, r_3} are flip-flops, i.e., the state variables, and {x_1, x_2} are inputs. The initial values of *flag*, r_0, r_1, r_2, and r_3 are all 1. The property we want to verify is **AG** (*flag* = 1).

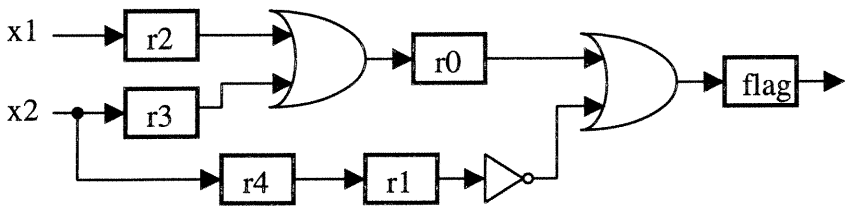


Figure 5. An example of circuit

In the circuit shown in Figure 5, the OR_2 gate has two fanins r_2 and r_3. Searching back the circuit from r_2 and r_3, we can see that there are no common predecessors of r_2 and r_3, the fanins of the OR_2 gate. Here we say s_1 a predecessor of s_2 if and only if the value of s_1 propagates to s_2. The values of r_2 and r_3 are thus not correlated. When we check the output of the OR_2 gate, we can keep r_2 as a state variable and change r_3 to a primary input or vice versa. In this example when r_2 is changed to a primary input, **AG** (*flag* = 1) holds on the reduced model. This gives us the following idea: partition the fanins of gates to sets $S_1,\ldots, S_n$ such that for any signal $s_1 \in S_i$, there exists $s_2 \in S_i$ having common predecessors with s_1, and there is no signal in S_j ($j \neq i$) having common predecessors with s_1. That is the values of the signals in different sets have no correlation. Each time we can select one set and constructs the reduced model by using the state variables connected to this set and changing the state variables connected to the other sets to primary inputs. If the reduced model satisfies the property then verification terminates, otherwise another set is selected to construct a new reduced model. If all these reduced models do not satisfy the property then the more complete model *sufficient* is used.

The reduced model constructed by reducing some state variables to primary inputs weakly preserves the property in ACTL. It is easy to understand this. Since the reduced state variables are changed to primary inputs, their values are chosen nondeterministically in the reduced model, thus the reduced model represents a more general transition system and larger reachable state space than the original model. If the property holds on the reduced model then it holds on the original one. The detailed proof is similar to the proof of Theorem 2 in Chapter 6, and is not included here.

The iterative reduction algorithm based on reducing inputs of gates is shown in Figure 6. When the procedure *Verify_circuit_topology*(*M*, *P*, *flags*) is called, its subprocedure *Find_fanin* starts from the flags, and searches the circuit in the

reverse direction of signal propagation until primary inputs or previously reached signals or gates. The input signals of currently reached multiple fanin gates are partitioned into sets of signals such that each set has no common predecessors with other sets by procedure *Partition_fanin*. If there are more than one set, then one set is selected, and *Find_unused_vars* finds the state variables that are not predecessors of this set. *Change_circuit* reduces these state variables to primary inputs. *Construct_transition_ relations* constructs the reduced transition model and *Modelcheck* verifies the property on the reduced model. If the result is success, i.e., the property holds on the reduced model, then verification terminates and returns success. Otherwise, another set of fanin signals is selected, and the above procedure is repeated. After every set of fanin signals has been selected to construct the reduced models and all these models do not satisfy the property, or if all currently found fanin signals belong to one set, *Verify_circuit_topology*(*M*, *P*, *fanins*) is called, and the algorithm continues to search backward in the circuit to find other multiple fanin gates, and the above procedure is repeated. When the search reaches the primary inputs and all the reachable parts of the design, the reduced model is the final *sufficient* model.

```
Verify_circuit_topology(M, P, sigs)
begin
   fanins := Find_fanin(sigs, M);
   if fanins ≠ ∅
   begin
      sets := Partition_fanin(fanins, M);
      if sets has more than 1 set
       begin
          while sets ≠ ∅
           begin
              Select one set S from sets;
              Remove S from sets;
```

```
            unuse_vars := Find_unused_vars(S, M);
            M_r := Change_circuit(unused_vars, M);
            T := Construct_transition_relations(M_r);
            result := Modelcheck(T, P);
            if result = success Return success;
          end
        end
        Verify_circuit_topology(M, P, fanins);
    end
    else begin
      M_S := Find_sufficient_circuit(M);
      T := Construct_transition_relations(M_S);
      result := Modelcheck(T, P);
      Return result;
    end
end
```

Figure 6. An iterative reduction algorithm based on the fanins of gates

For the circuit in Figure 5, to verify the property **AG** (*flag* = 1), our algorithm starts from *flag*, searches backward in the circuit and finds the gate OR_1. Since the fanin signals of OR_1 have the common predecessor signal x_2, they cannot be partitioned. Then the algorithm searches further and finds the gate OR_2. The two fanin signals of OR_2 have no common predecessor signals and are partitioned to two sets $\{r_2\}$ and $\{r_3\}$. First $\{r_2\}$ is selected and r_3 is reduced to a primary input, but the property check fails. Then $\{r_3\}$ is selected and r_2 is reduced to an input. This time the property holds on the reduced model constructed by using the state variables $\{flag, r_0, r_1, r_3, r_4\}$. The experimental result is shown below.

Without reduction:

```
=== Circuit statistics ===

Total components: 13
Total signals:    15
  Abstract signals: 0
  Concrete signals: 15  which is equivalent to 15 boolean signals
Total state variables:  6
  Abstract state variables:  0
  Concrete state variables: 6 which is equivalent to 6 boolean
  variables


=== Performance statistics ===

  Total time spent:
     Run time : 0.160 seconds ; System time : 0.030 seconds ; Real
     time : 0.583 seconds .
  State variable coverage : 6 , 100% of all state variables.
  Nodes: 123;   Compound Terms: 1.
  Memory usage: 1189144 bytes.
  Garbage_collection 6 times: 0.040 seconds; 813636 bytes freed.
```

With reduction:

```
=== Circuit statistics ===

Total components: 11
Total signals:    15
  Abstract signals: 0
  Concrete signals: 15  which is equivalent to 15 boolean signals
Total state variables:  5
  Abstract state variables:  0
  Concrete state variables:  5  which is equivalent to 5 boolean
  variables
```

```
=== Performance statistics ===

  Total time spent:
     Run time : 0.100 seconds ; System time : 0.020 seconds ; Real
     time : 0.212 seconds .
  State variable coverage : 5 , 83% of all state variables.
  Nodes: 96;    Compound Terms: 1.
  Memory usage: 985472 bytes.
  Garbage_collection 1 times: 0.010 seconds; 280144 bytes freed.
```

The circuit statistics tells us the information about the circuit that was used to verify the property. From the result we can see that without our reduction algorithm the MDG tool verified the property on the original circuit using 13 components and 6 state variables, while using the reduction algorithm the MDG tool verified the property on the reduced model using 11 components and 5 state variables. The performance statistics tells us the information about the time, state variable coverage (i.e., the number of state variables that were used and its percentage of the total state variables in the original design), the number of MDG nodes, memory usage, and etc. We can see that using our reduction algorithm the number of MDG nodes, memory usage, cpu time and run time are also decreased.

Summary

In this chapter we presented heuristic reduction algorithms that are based on the circuit topology. Beginning from the signal flags, our algorithms search through the circuit in the reverse direction of signal propagation and find all the signals and components that control the flags. This model constructed using all the reached signals and components is called the *sufficient* circuit, meaning that this part of the circuit is sufficient to verify the property. The property is strongly preserved. We

can obtain further reduction by iteratively reducing some input signals of multiple fanin gates. The method is completely automatic without any user guidance.

The reduction based on circuit topology may include some unnecessary state variables. The *sufficient* model may not be the least model that preserves the property strongly. In the subsequent chapters we will present reduction algorithms based on functional dependency and we will then obtain the least model regardless the initial states that strongly preserves the property.

Chapter 6 Model Reductions Based on the Property Dependent State Variables(DV_P)

In the previous chapter we introduced a reduction method based on circuit topology. However, there still may exist a situation that a signal is connected to *flag*, but its value cannot influence *flag*. The *sufficient* circuit preserves the property strongly, but it is not necessarily the smallest model. We want to find the least model that strongly preserves the property, and iterative reduction methods that can further reduce state variables from the original model. In this chapter we present a reduction method based on functional dependency. Given a property P we search for the so-called property dependent state variables DV_P and construct the reduced model to verify P using only the individual transition relations of DV_P. We prove that the abstract system constructed by the transition relations of DV_P is the least model that we can obtain without reachability analysis on the original machine that strongly preserves P, and the abstract system constructed by a subset of DV_P weakly preserves P. This method is different from those that compute an equivalence relation using preimage operations. Hence this method is particularly useful for MDG, although it can be used in other model checking tools as well.

6.1 Definitions

Given a transition system $M = (Y, Y', X, T)$, let $Y = (y_1, \ldots, y_n)$ be the set of state variables, $Y' = (y_1', \ldots, y_n')$ be the set of corresponding next state variables. We use the primed symbol y' to represent the next state variable of the state variable y. Let $X = (x_1, \ldots, x_m)$ be the set of input variables, and T the transition relation. If f_i is the next state function of y_i', then $y_i' = f_i(Y, X)$. The transition relation of the

state variable y_i is $T_i(Y, X, y_i') \Leftrightarrow (y_i' = f_i(Y, X))$. The transition relation of the entire model can be expressed as a conjunction of the individual transition relations of the state variables [BCL91]:

$$T(Y, X, Y') = T_1(Y, X, y_1') \wedge T_2(Y, X, y_2') \wedge \ldots \wedge T_n(Y, X, y_n') \qquad (1)$$

Definition 1. Let $ddv(y_i)$ be the set of direct determining variables of y_i. It includes all variables $v \in Y \cup X$ such that $f_i|_{v=a} \neq f_i|_{v=b}$ for some $a \neq b$, where $f_i|_{v=x}$ is the cofactor of f_i for $v = x$. Let $dv(y_i)$ denote the set of determining variables of y_i that is defined recursively as follows: $dv(y_i) = ddv(y_i) \cup \{dv(y_j) \mid y_j \in ddv(y_i) \setminus y_i\}$. Then, let $ddsv(y_i)$ be the set of direct determining state variables of y_i, $ddsv(y_i) = ddv(y_i) \setminus X$, and $dsv(y_i)$ denote the set of determining state variables of y_i, $dsv(y_i) = dv(y_i) \setminus X$. The variables that are not in $dv(y_i)$ are called *don't care* variables of y_i. For a set of state variables $SetV = \{y_1, \ldots, y_k\}$, we have $ddv(SetV) = ddv(y_1) \cup \ldots \cup ddv(y_k)$, $dv(SetV) = dv(y_1) \cup \ldots \cup dv(y_k)$, $ddsv(SetV) = ddv(SetV) \setminus X$, $dsv(SetV) = dv(SetV) \setminus X$.

Definition 2. Let P be the property to be verified, and $Y_P = (y_1, \ldots, y_k)$ be the state variables appearing in P. The set of determining variables of the P is $dv(Y_P) = dv(y_1) \cup \ldots \cup dv(y_k)$, and the set of determining state variables is $dsv(Y_P) = dv(Y_P) \setminus X$. Let DV_P be the set of property dependent state variables of P, and $DV_P = Y_P \cup dsv(Y_P)$.

6.2 Property Preservation on Reduced Models

In this section, we prove that for verifying a property P it is sufficient to use the individual transition relations of the variables in DV_P to compute the transition system. We prove that the so-constructed system is the least model not requiring reachability analysis on the original machine that preserves P strongly. We also

prove that a reduced model constructed using a subset of DV_P containing the state variables in P preserves P weakly.

For the transition system M defined above, and a property P to be verified, we reduce all the *don't care* state variables and construct the reduced machine $M_R = (Y_R, Y_R', X, T_R)$, where Y_R is the set of the property dependent state variables DV_P, Y_R' contains the corresponding next state variables, X is the set of input variables, and T_R is the conjunction of the individual transition relations of the state variables in DV_P. Let $DV_P = \{y_1, \ldots, y_k\}$.

$$T_R(Y_R, X, Y_R') = T_1(Y_R, X, y_1') \wedge \ldots \wedge T_k(Y_R, X, y_k') \qquad (2)$$

In formula (2), T_1 to T_k are the same as those in formula (1). Since $y_1', \ldots, y_k'$ are not dependent on y_{k+1} to y_n, then $T_1(Y, X, y_1') = T_1(Y_R, X, y_1'), \ldots, T_k(Y, X, y_1') = T_k(Y_R, X, y_1')$. The transition relations of the *don't care* variables, T_{k+1} to T_n, are removed from T_R, which means that the *don't care* state variables are reduced.

From the definition of M and M_R, we can see that M represents a state machine in which each state can be represented by a characteristic predicate over the state variables $\{y_1, \ldots, y_n\}$, denoted by $C(y_1, \ldots, y_n)$, while M_R represents a state machine in which each state can be represented by a characteristic predicate over the state variables in DV_P, denoted by $C(y_1, \ldots, y_k)$. Obviously, each state in M_R represents a set of states in M. Thus the set of the states in M_R can be viewed as a partition of the state space of M, and each state of M_R can be viewed as a block of the partition. The initial states in M_R are those blocks that contain some initial state in M, i.e., $C_{R0}(y_1, \ldots, y_k)$ represents an initial state of M_R if and only if there exists an initial state of M, denoted by $C_0(y_1, \ldots, y_n)$ and $C_{R0}(y_1, \ldots, y_k) \wedge C_0(y_1, \ldots, y_n) = C_0(y_1, \ldots, y_n)$. We will prove that the transition relation T_R makes the reduced system M_R a stable quotient system of M.

Given two states s_1 and s_2 in M, represented by $C_1(y_1, \ldots, y_n)$ and $C_2(y_1, \ldots, y_n)$ respectively, belong to the two different states B_1 and B_2 of M_R, represented by

$C_1(y_1, ..., y_k)$ and $C_2(y_1, ..., y_k)$ respectively, if there is a transition from s_1 to s_2 guarded by input value $a(X)$ in M, we will see that there is a transition from B_1 to B_2 guarded by $a(X)$ in M_R. Since the predicate over a set of variables is a conjunct of the atomic propositions of the variables, we can use $C_2(y_1, ..., y_k) \wedge C_2(y_{k+1}, ..., y_n)$ to represent $C_2(y_1, ..., y_n)$. During the following computation $\{y_{k+1}, ..., y_n\}$ can be quantified earlier since $y_1, ..., y_k$ do not depend on $\{y_{k+1}, ..., y_n\}$ [BCL91].

$\exists_{y1, ..., yn, X} [C_1(y_1, ..., y_n) \wedge a(X) \wedge T_1(Y,X,y_1') \wedge ... \wedge T_n(Y,X,y_n')]] = C_2(y_1', ..., y_n')$

$\Rightarrow \exists_{y1, ..., yk, X} [T_1(Y,X,y_1') \wedge ... \wedge T_k(Y,X,y_k') \wedge [\exists_{yk+1, ..., yn} [C_1(y_1, ..., y_n) \wedge a(X) \wedge$

$T_{k+1}(Y,X,y_{k+1}') \wedge ... \wedge T_n(Y,X,y_n')]]] = C_2(y_1', ..., y_k') \wedge C_2(y_{k+1}', ..., y_n')$

$\Rightarrow \exists_{y1, ..., yk, X} [T_1(Y,X,y_1') \wedge ... \wedge T_k(Y,X,y_k') \wedge C_1(y_1,..., y_n) \wedge a(X) \wedge C_2(y_{k+1}',..., y_n')]$

$= C_2(y_1', ..., y_k') \wedge C_2(y_{k+1}', ..., y_n')$

$\Rightarrow \exists_{y1, ..., yk, X} [T_1(Y,X,y_1') \wedge ... \wedge T_k(Y,X,y_k') \wedge C_1(y_1, ..., y_n) \wedge a(X)] = C_2(y_1', ..., y_k')$

Since $T_1, \ldots, T_k$ do not depend on y_{k+1} to y_n, then

$\exists_{y1, ..., yk, X} [T_1(Y,X,y_1') \wedge ... \wedge T_k(Y,X,y_k') \wedge C_1(y_1, ..., y_n) \wedge a(X)]$

$= \exists_{y1, ..., yk, X} [T_1(Y_R,X,y_1') \wedge ... \wedge T_k(Y_R,X,y_k') \wedge C_1(y_1, ..., y_k) \wedge a(X)]$

$\Rightarrow \exists_{y1, ..., yk, X} [T_1(Y_R,X,y_1') \wedge ... \wedge T_k(Y_R,X,y_k') \wedge C_1(y_1,..., y_k) \wedge a(X)] = C_2(y_1',..., y_k')$

Next we will prove the opposite direction: for any transition $B_i \rightarrow_a B_j$ in M_R, there is a transition $s_1 \rightarrow_a s_2$ in M such that $s_1 \in B_i$ and $s_2 \in B_j$. Let $C_i(y_1, ..., y_k)$ represent B_i, $C_j(y_1, ..., y_k)$ represent B_j, and $a(X)$ represent the current values of the input variables X. The transition $B_i \rightarrow_a B_j$ is represented by the following formula:

$\exists_{y1, ..., yk, X} [C_i(y_1, ..., y_k) \wedge a(X) \wedge T_1(Y,X,y_1') \wedge ... \wedge T_k(Y,X,y_k')] = C_j(y_1, ..., y_k)$,

after renaming y_1' to $y_1, \ldots,$ and y_k' to y_k

Now we compute the post image of B_i in M.

$post_a(B_i) = post_a(C_i(y_1, ..., y_k))$

$= \exists_{y1, ..., yn, X} [C_i(y_1, ..., y_k) \wedge a(X) \wedge T_1(Y,X,y_1') \wedge ... \wedge T_n(Y,X,y_n')]]$

$= \exists_{y1, ..., yk, X} [T_1(Y,X,y_1') \wedge ... \wedge T_k(Y,X,y_k') \wedge [\exists_{yk+1, ..., yn} [C_i(y_1, ..., y_k) \wedge a(X) \wedge$

$T_{k+1}(Y,X,y_{k+1}') \wedge ... \wedge T_n(Y,X,y_n')]]]$

$= \exists_{y1, \ldots, yk, X} [T_1(Y,X,y_1') \wedge \ldots \wedge T_k(Y,X,y_k') \wedge C_i(y_1, \ldots, y_k) \wedge a(X) \wedge C(y_{k+1}', \ldots, y_n')]$

$= C_j(y_1', \ldots, y_k') \wedge C(y_{k+1}', \ldots, y_n')$

$= C_j(y_1, \ldots, y_k) \wedge C(y_{k+1}, \ldots, y_n)$ after renaming y_1' to y_1, …, and y_n' to y_n

$\subseteq C_j(y_1, \ldots, y_k)$

That is, we get the post image of B_i in M that is a set of states represented by $C_j(y_1, \ldots, y_k) \wedge C(y_{k+1}, \ldots, y_n)$. From the above we can see that when there is a transition $B_i \rightarrow_a B_j$ in M_R, then there exists some transition in M from a state of M that belongs to B_i to a state of M that belongs to B_j. Since $post_a(B_i) \subseteq B_j$, the transition $B_i \rightarrow_a B_j$ is stable, and since any transition in M_R is stable, M_R is stable too. That is, M_R is a stable quotient system of M, and M_R and M are bisimular. The formulas in CTL* with atomic propositions over DV_P are thus strongly preserved by M_R. From the above proof, theorem 1 follows.

Theorem 1. The reduced model M_R constructed using the transition relations of the state variables in DV_P strongly preserves property P.

Now, we will prove that the reduced model M_R constructed by using all the state variables in DV_P is the least system that bisimulates the original model regardless the initial states. Suppose that DV_P has k state variables $\{y_1 \ldots y_k\}$, and there exists a smaller reduced model M_1 that is obtained by using k-1 state variables in DV_P, i.e., reducing it only by one state variable of M_R. Let y_k be the state variable that is reduced, and M_1 is constructed using $\{y_1 \ldots y_{k-1}\}$. In the following we will prove that M_1 does not bisimulate M_R.

Let B_i be a state in M_1, represented by $C_i(y_1, \ldots, y_{k-1})$. Let $B_i \rightarrow_a B_j$ be any transition in M_1. Obviously B_i represents one set of states in M_R. Suppose that b_1 and b_2 are two states of M_R that belong to B_i. Let $C_{b1}(y_1, \ldots, y_k)$ and $C_{b2}(y_1, \ldots, y_k)$ represent b_1 and b_2 respectively. Then $C_{b1}(y_1, \ldots, y_{k-1}) = C_{b2}(y_1, \ldots, y_{k-1}) = C_i(y_1, \ldots, y_{k-1})$ and $C_{b1}(y_k) \neq C_{b2}(y_k)$. Let b_1' and b_2' be the next states of b_1 and b_2, represented by $C_{b1}'(y_1, \ldots, y_k)$ and $C_{b2}'(y_1, \ldots, y_k)$ respectively. Let y_h be the state variable in $\{y_1 \ldots$

$y_{k-1}\}$ whose value is determined directly by y_k. Since the value of y_k is different in state b_1 and b_2, i.e. $C_{b1}(y_k) \neq C_{b2}(y_k)$, the value of y_h' is thus different in b_1' and b_2'. Then $C_{b1}'(y_1, ..., y_{k-1}) \neq C_{b2}'(y_1, ..., y_{k-1})$. That is, b_1' and b_2' belong to two different sets that correspond to two states in M_1. Since $post_a(B_i) \subseteq B_j$ does not exist, the transition $B_i \rightarrow_a B_j$ in M_1 is not stable. Hence M_1 does not bisimulate M_R. From the above proof, we have the following Theorem 2.

Theorem 2. The abstract model M_R constructed by the transition relations of the state variables in DV_P is the least reduced model that bisimulates the original system regardless the initial states.

There still may be reduced models that strongly preserve the property and that are smaller than the one described in Theorem 2. This is because we do not consider specific initial states there. To consider them, however, would require removing unreachable states and necessitate carrying out reachability analysis on the original system which we want to avoid.

According to Theorem 1, for a given property P, the reduced transition system M_R constructed by the transition relations of all the variables in DV_P strongly preserves P. But the resulting M_R may still be too large. Can we do better reduction than M_R? In the following we will prove that the reduced model M_r constructed by using the transition relations of a subset of DV_P weakly preserves P.

We construct M_r in a similar way as M_R. Let $M_r = (Y_r, Y_r', X_r, T_r)$ where $Y_r = \{y_1, ..., y_m\}$ is the set of state variables, a subset of DV_P, $m \leq k$, Y_r' is the set of the corresponding next state variables, X_r is the set of input variables and $X_r = X \cup (DV_P \setminus Y_r)$. The state variables in DV_P but not selected in Y_r are reduced to primary inputs. The transition relation T_r is a conjunction of the individual transition relations of the variables in Y_r. The initial states in M_r are those blocks that contain

an initial state in M, i.e., $C_{r0}(y_1, \ldots, y_m)$ represents an initial state of M_r if and only if there exists an initial state of M, denoted by $C_0(y_1, \ldots, y_n)$ and $C_{r0}(y_1, \ldots, y_m) \wedge C_0(y_1, \ldots, y_n) = C_0(y_1, \ldots, y_m)$.

Let b_1 be a state of M_R represented by $C_1(y_1, ..., y_k)$ and B_1 be a corresponding state of M_r represented by $C_1(y_1, ..., y_m)$ such that b_1 belongs to B_1. If there is a transition from b_1 to b_2 guarded by input value $a(X)$ in M_R, we have the following formula:

$$b_2 = \exists_{y1, ..., yk, X} [C_1(y_1, ..., y_k) \wedge a(X) \wedge T_1(Y,X,y_1') \wedge ... \wedge T_k(Y,X,y_k')]]$$

Let B_2 be the next state of B_1 by the transition from B_1 guarded by $a(X)$ in M_r. Then we have:

$$B_2 = \exists_{y1, ..., yk, X} [C_1(y_1, ..., y_m) \wedge a(X) \wedge T_1(Y,X,y_1') \wedge ... \wedge T_m(Y,X,y_m')]]$$

If b_2 belongs to B_2, then M_r simulates M_R, and the property expressed in ACTL* with atomic propositions over Y_r is weakly preserved [CGL96]. In the following we will see that b_2 belongs to B_2.

$$b_2 = \exists_{y1, ..., yk, X} [C_1(y_1, ..., y_k) \wedge a(X) \wedge T_1(Y,X,y_1') \wedge ... \wedge T_k(Y,X,y_k')]]$$

$$\subseteq \exists_{y1, ..., yk, X} [C_1(y_1, ..., y_k) \wedge a(X) \wedge T_1(Y,X,y_1') \wedge ... \wedge T_m(Y,X,y_m')]], m \leq k$$

$$\subseteq \exists_{y1, ..., yk, X} [C_1(y_1, ..., y_m) \wedge a(X) \wedge T_1(Y,X,y_1') \wedge ... \wedge T_m(Y,X,y_m')]] = B_2$$

Theorem 3. The abstract model M_r constructed by using any subset of DV_P containing the state variables in P weakly preserves an ACTL* property P.

It is easy to understand Theorem 3. Since all other state variables in DV_P but not in Y_r are reduced to primary inputs, they are nondeterministic in M_r, M_r thus represents a more general transition system than M_R. If the property holds on M_r, the property holds on M_R too, and then it holds on M. If the property does not hold on M_r, it may or may not hold on M_R and M. Hence, the reduced transition system M_r constructed by a subset of DV_P weakly preserves P.

In MDG model checking, a property expressed in L_{MDG} is transferred to an additional circuit and a simplified property. The verification of the property on the

original machine is transferred to the verification of the simplified property on the composite machine. Since the simplified property is a subset of ACTL, the above theorems apply to MDG model checking.

According to Theorem 3 we know where we can start a reduction. According to Theorem 1 and 2, we know that the reduced model M_R is the least model that we are sure that property P is strongly preserved without constructing the original transition system. Thus we know where we can stop. This is used in the iterative reduction algorithms presented in Chapter 7.

6.3 Construction of *ddv* Hash Table in MDG

In MDG model checking, all the sets of states and transition relations are represented by Directed Formulas (DFs) that are directly transformed to MDG graphs. To find the functional dependency of the property on the system, we need to find the direct determining state variables of the state variables appearing in the property, and then recursively find other such variables. It is easy to get direct determining variables in MDG, since there are no redundant nodes and no redundant subgraphs. The variables appearing in the MDG representing the transition relation of the state variable v are thus the direct determining variables of v, i.e. $ddv(v)$. By scanning all the MDGs representing the individual transition relations, we can construct a hash table of direct determining variables for all state variables in the design. Then we can get the direct determining state variables, determining variables, and determining state variables of any state variable and any set of state variables. It means that we can get the property dependent state variables DV_P of property P.

Given the circuit description file, order file and algebric file of the design, and a property to be verified, first we use the "next" command to construct the composite machine and the simplified property. Then we compile the circuit and

build the individual transition relations and from them we build the hash table of direct determining variables for every state variable. One individual transition relation is represented by a compound term *R(NextStVar*, MDG). One MDG is represented by the term *graph*(_, *NodeKind*, *NodeLabel*, _, _ , *SubGraphs*, *SecVars)* where "_" represents the terms we do not use here, *NodeKind* is concrete, abstract or cross-term, *NodeLabel* is the label of the node, *SubGraphs* is the immediate subgraphs, and *SecVars* is a set containing all the secondary variables in the graph, i.e., the variables appearing on the edges or in the cross-terms. We can find the set of primary variables in an MDG by searching the node in the *graph* and recursively searching the nodes in the *SubGraphs*. *ddv*(v) is the set of primary variables and secondary variables excluding v', the next state variable of v in the MDG representing the transition relation of v. The procedure *construct_ddv_table* shown below constructs *ddv(v)* for a state variable v. The arguments are *circuit_file* that is the circuit description file of the composite machine defining the circuit, the initial states, and the mapping function η from next state variables to state variables, *order_file* defining the order of all variables and uninterpreted function symbols of the composite machine, and *alg_file* that is the algebric file defining the sorts, functions and rewriting rules used in the composite machine.

```
Construct_ddv_table(circuit_file, order_file, alg_file)
Begin
    Transitions := Compile_circuit_construct_individual_transitions(
                        circuit_file, order_file, alg_file);
    While Transitions ≠ ∅
    Begin
        Choose a transition R ∈ Transitions;
        Transitions := Transitions \ R;
        v' := R(NextStVar);
```

```
        v := η (v');
        G := R(MDG);
        While G ≠ T
        Begin
            primary := Find_primary_variables(∅, G);
            ddv(v) := G(SecVars) ∪ (primary \ v');
        end
    end
    Return ddv;
end

Find_primary_variables(primvar, graph)
Begin
    primary := primvar;
    If (graph(NodeKind) ≠ cross-term)
    Begin
      primary := primary ∪ graph(NodeLabel);
    end
    For each subgraph ∈ graph(SubGraphs) begin
        If subgraph ≠ T begin
            primary := Find_primary_variables(primary, subgraph);
        end
    end
Return primary;
end
```

Figure 7. Procedure Construct_ddv_table in the MDG model checker

Compile_circuit_construct_individual_transitions checks if there are any syntax errors in the description file and constructs the individual transition relations for

every state variable. It also constructs the database which contains the table of input signals, the table of abstract input signals, the table of the pairs (state variable, its next state variable), the table of the initial values of state variables, and the table of the order of variables, etc.

After the individual transition relations are constructed, individual transition relations are selected one at a time to compute *ddv*(*v*). *Find_primary_variables (primvar, graph)* searches the MDG representing the individual transition relation of *v*, and finds all the primary variables appearing in the MDG. If a node is not a cross-term then its labeling variable is a primary variable. *Find_primary_variables* starts from the root node and makes depth-first search in the MDG. If a node is reached, *Find_primary_variables* is recursively revoked to find the variables labeling the nodes in the subgraphs. After we obtain the set of the primary variables and secondary variables in an MDG and remove the next state variable from this set, we then get all the direct determining variables of *v*. After the direct determining variables of all the state variables are obtained, the hash table *ddv* is constructed.

In the following we show the state machine of the *MinMax* example from Figure 3, and then draw the MDGs representing the individual transition relations of the state variables *c*, *rm* and *rM*. We will see that the direct determining variables of the state variables can be obtained from the MDGs and the reduced model can thus be constructed. The state machine of the *MinMax* machine is as follows:

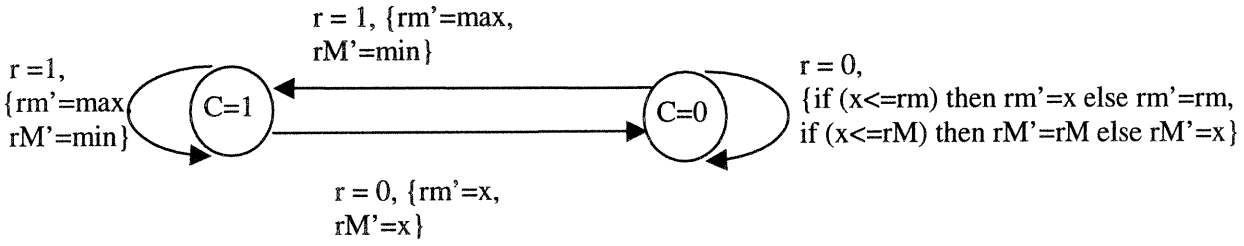


Figure 8. The MinMax state machine

The individual transition relations of the state variables *c*, *rm*, and *rM* are shown in Figure 9. From the transition T_c, T_{rm} and T_{rM}, we can get the sets of the direct determining variables of the state variables *c*, *rm* and *rM* respectively, i.e., $ddv(c) = \{r\}$, $ddv(rm) = \{r, x, c, rm\}$, and $ddv(rM) = \{r, x, c, rM\}$.

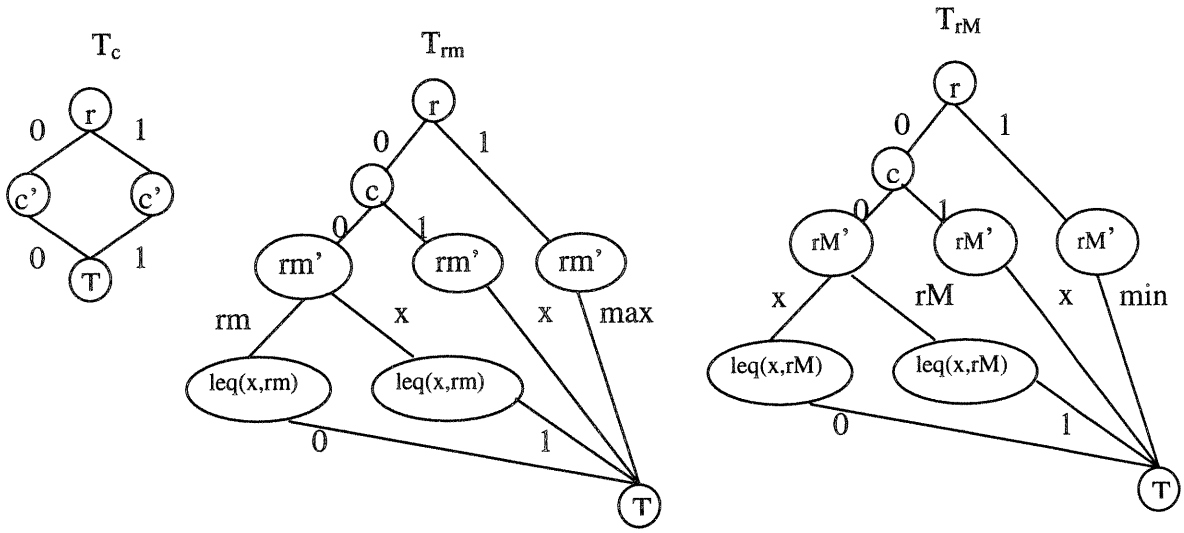


Figure 9. MDGs of the individual transition relations of MinMax

For example, we wish to verify that if $r = 1$ then *rm* will be loaded the generic constant *max* in the next clock cycle. This property expressed by L_{MDG} is as follows:

$$P\text{: } \mathbf{AG}\ ((r = 1) \text{ -> } \mathbf{X}\ (rm = max));$$

The additional circuit constructed for this property is as shown in Figure 10. After composing the additional circuit with the original design, we verify **AG** (*flag* = 1) on the composite machine.

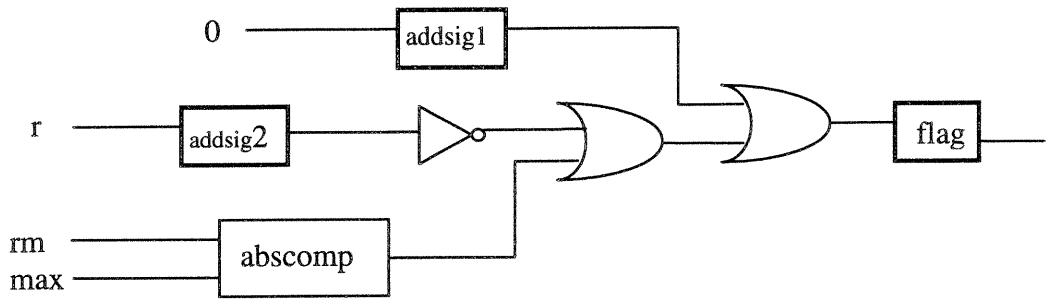


Figure 10. The additional circuit for **AG** $((r = 1) \rightarrow \mathbf{X}\ (rm = max))$;

In the additional circuit, there are three state variables {*flag*, *addsig*1, *addsig*2}, and their initial values are all 1. The transition relations of the three state variables are represented by the following Directed Formulas (DFs):

T_{flag}: $(addsig1=1 \wedge flag'=1)$

$\vee(addsig1=0 \wedge (addsig2=0 \vee (addsig2=1 \wedge abscomp(rm,max)=1)) \wedge flag'=1)$

$\vee(addsig1=0 \wedge addsig2=1 \wedge abscomp(rm, max)=0 \wedge flag'=1)$

$T_{addsig1}$: $(addsig1'=0)$

$T_{addsig2}$: $(r=0 \wedge addsig2'=0) \vee (r=1 \wedge addsig2'=1)$

Since the DFs are directly transformed to MDGs, the variables appearing in the DF representing the transition relation of the state variable v are its direct determining variables. From T_{flag}, $T_{addsig1}$ and $T_{addsig2}$, we get $ddv(flag) = \{addsig1, addsig2, rm\}$, $ddv(addsig1) = \varnothing$, and $ddv(addsig2) = \{r\}$.

We have the lists of direct determining variables for all the state variables in the composite machine. Beginning from state variable *flag*, we can recursively compute the property dependent state variables as $DV_P = \{flag, addsig1, addsig2, rm, c\}$. We can see that to verify P, the state variable rM is not needed. We can construct the reduced system by only using the transition relations of DV_P, and the property is strongly preserved.

Summary

In this chapter, we defined the property dependent state variables DV_P for property P and proved that the reduced model constructed by using all the state variables in DV_P strongly preserves P. Furthermore, the reduced models constructed by a subset of DV_P preserve P weakly. We also showed how to get the direct determining variables of the state variables using MDG. In the next chapter we will present two iterative reduction algorithms based on the theorems in this chapter.

Chapter 7 Iterative Reduction Algorithms Based on Depth-first and Breadth-first Search of PPDG

From Theorem 1 in Chapter 6, for a given property P we can only use the transition relations of the property dependent state variables in DV_P to build the reduced system to verify P. P is strongly preserved in that case. However, this reduced system may still produce large state space. From Theorem 3 it follows that we can use a subset of the transition relations of DV_P to construct the reduced system, however, the property may now be only weakly preserved. This leads us to consider an iterative reduction method. First we can select a small subset of DV_P and use the transition relations of the state variables in the subset to build a reduced system, and then check if P holds on it. If the answer is "yes", the procedure terminates and we can say P holds on the original machine. If the answer is "no", we can add more state variables in DV_P to construct another reduced machine and check P. We repeat this procedure until all the individual transition relations of the variables in DV_P are used to construct the reduced machine. Then, the property P is strongly preserved.

The critical thing of the iterative reduction methods is how to select the subset of DV_P at each iteration step. If these subsets are not well selected, the reduced machines cannot satisfy the property P, it may thus take many iteration steps and longer execution time. In the worst case all the reduced systems constructed by the selected subsets of DV_P cannot satisfy P and this leads to all the state variables in

DV_P being eventually used. We want to avoid this. In the following we will introduce our iterative reduction algorithms that are based on the *property dependency graph* (PDG) and *noncorrelated sets*.

7.1 Definition of PDG and Noncorrelated Sets

We can view the dependency of property P on the determining variables as a directed graph, called property dependency graph (PDG). The definition of PDG is given below.

Definition 1: We define the graph PDG = (V, E, L), where V is the set of nodes, E is the set of edges, and L is the set of labels of the nodes. Let Y_P be the set of the state variables appearing in property P. The root of PDG is labeled by Y_P. Each of the other nodes in the graph is labeled by one determining variable of Y_P that can be a state variable or an input. There is an edge $v \rightarrow w$ in PDG if and only if $w \in ddv(v)$, i.e., w is a direct determining variable of v.

According to the definition, PDG might be a cyclic graph. We define the *levels* in PDG such that the root is at level 0, the nodes that can be reached by n edges along the shortest paths from the root are at level n.

After we find the direct determining variables $ddv(v)$ for each state variable v in M, we build the PDG starting from the root labeled by the set of the state variables appearing in P. Obviously all the variables in the PDG form the set of determining variables of P, all the state variables in the PDG represent the property dependent state variables DV_P, and any subgraph starting from any node in PDG represents the determining variables of this node.

For the composite machine of the *MinMax* example shown in Figure 3, we obtain the direct determining variables of each state variable from their individual transition relations as follows:

$ddv(c) = \{r\}$, $ddv(rm) = \{r, x, c, rm\}$, $ddv(rM) = \{r, x, c, rM\}$.

$ddv(flag) = \{addsig1, addsig2, rm\}$, $ddv(addsig1) = \varnothing$, $ddv(addsig2) = \{r\}$.

Beginning with *flag*, we can construct the property dependency graph for the simplified property *P'*: **AG** (*flag* = 1) as shown below, where *r* and *x* are inputs and the other variables are state variables.

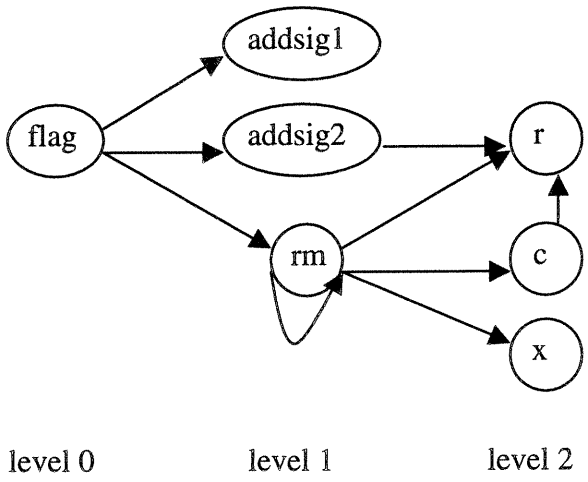


Figure 11. PDG of property P of the *MinMax* machine

From the PDG we can see that the state variable *rM* in *MinMax* machine is not in the graph. It is a *don't care* variable of the property to be verified, and thus it can be removed from the reduced machine. All the variables in the PDG are determining variables of *P*, and the set of state variables in PDG is exactly the set of property dependent state variables DV_P. In this example, we can see that DV_P = {*flag*, *addsig*1, *addsig*2, *rm*, *c*}.

When the PDG and the set of property dependent state variables DV_P have been obtained, we want to select a subset of DV_P to build the reduced system. If the subset is well selected then the verification can terminate much faster. In our method we select a so-called *noncorrelated* set at each iteration step. The definition of noncorrelated sets is given next.

Definition 2: Let S be the set of state variables of M, and Let $S_1 \subset S$ and $S_2 \subset S$ be two disjoint subsets, $S_1 \cap S_2 = \varnothing$. If $dv(S_1) \cap dv(S_2) = \varnothing$, which means that S_1 and S_2 have no common determining variables, then S_1 and S_2 are *noncorrelated.*

In our example, the two sets (*addsig*1) and (*addsig*2, *rm*) in Figure 11 are noncorrelated, since dv(*addsig*1) = (*addsig*1) and dv(*addsig*2, *rm*) = (*addsig*2, *rm*, *r*, *c*, *x*) have no common determining variables.

From the definition we can see that if two sets S_1 and S_2 are noncorrelated then the variables in $S_1 \cup dv(S_1)$ have no influence on the values of the variables in $S_2 \cup dv(S_2)$, and vice versa. In PDG, the corresponding subgraphs are disjoint. The basic idea of our reduction method is to partition the state variables of PDG into noncorrelated sets that cannot be further partitioned, and each time select one set to construct the reduced model.

7.2 An Algorithm for Finding Noncorrelated Sets

We start from any variable $v_1 \in S$ and add it to S_1. If a variable $v_j \in S \setminus S_1$ and $dv(v_j) \cap dv(S_1) \neq \varnothing$, then v_j is added into S_1. Repeat this procedure until we cannot find a variable in $S \setminus S_1$ having a common determining variable with S_1. Thus we find the first set S_1 of correlated variables. We start from another variable $v_2 \in S \setminus S_1$ and find the second set S_2. We repeat this procedure until every variable in S has been considered. The resulting sets $S_1, \ldots, S_n$ are noncorrelated. The algorithm is as follows.

```
Partition_set(vars, PDG)
Begin
    noncorrelated_sets := ∅;
   While vars ≠ ∅
   begin
      Select v ∈ vars;
      vars := vars \ v;
      Si := Find_one_set(v, vars, PDG);
      noncorrelated_sets := Si ∪ noncorrelated_sets;
      vars := vars \ Si;
   end
   Return noncorrelated_sets;
end

Find_one_set(v, vars, PDG)
begin
   Si := {v};
   dv_Si := dv(v);
   While vars ≠ ∅
   begin
      Select w ∈ vars;
      vars := vars \ w;
      dv_w := dv(w);
      If dv_Si ∩ dv_w ≠ ∅ then
      begin
         Si := Si ∪ {w};
         dv_Si := dv_Si ∪ dv_w;
      end
```

```
    end
    Return (Si);
end
```

Figure 12. An algorithm for finding noncorrelated sets of state variables

Partition_set(vars, PDG) partitions the variables in *vars* to noncorrelated sets. Here, *vars* contains a subset of state variables in *DVp*, and PDG is implicitly represented by the hash table of direct determining variables of all the state variables in DV_P. First we select one variable *v* from the set *vars*. *Find_one_set(v, vars, PDG)* finds all the variables in *vars* that correlate to *v*. The resulting set is added to *noncorrelated_sets*. We remove this set from *vars*, and continue searching for the next set of correlated variables. When *vars* is empty, all variables in the initial *var* have been considered and the result is stored in *noncorrelated_sets*.

Find_one_set(v, vars, PDG) finds the variables in *vars* correlating to *v*. First it obtains the determining variables of *v* by searching the hash table *dv*, and adds them to *dv_Si*. Then it selects a variable *w* in *vars*, and finds the determining variables of *w*. If *w* and *v* have common determining variables, *w* is in the same set as *v*, add *w* to *Si* and add *dv*(*w*) to *dv_Si*. For the next variable in *vars*, we need to check if it has a determining variable included in *dv_Si*. If it has then it is added to *Si* and its determining variables are added to *dv_Si*. When *vars* is empty, all its variables have been checked and the correlated set including *v* has been found.

The algorithm *Construct_dv_table*(Y_P, PDG) that constructs the hash table *dv* of the determining variables for the state variables in PDG is shown in Figure 13.

```
Construct_dv_table(Y_P, PDG)
begin
   Y := Y_P;
    While Y ≠ ∅
     begin
        Select v ∈ Y;
         Y := Y \ v;
         If v is not marked as visited
         begin
              dv(v) := Find_dv(v, PDG);
          end
     end
end


Find_dv(v, PDG)
Begin
    Mark v as visited;
    dv(v) := ddv(v);
    ddsv := ddv(v) \ inputs;
   While ddsv ≠ ∅
   Begin
       Select w ∈ ddsv;
       ddsv := ddsv \ w;
       If w is not marked as visited then
       Begin
          dv(w) := Find_dv(w, PDG);
          dv(v) := dv(v) ∪ dv(w);
        End
```

```
    End
    Return (dv(v));
End
```

Figure 13. An algorithm to construct a *dv* table

Find_dv(v, PDG) finds the determining variables of v. It first finds the direct determining variables of v, i.e., $ddv(v)$, adds them to $dv(v)$, and then removes inputs from $ddv(v)$ to get the direct determining state variables of v which are stored in *ddsv*. Then it selects a variable w in *ddsv* to recursively compute the determining variables of w, i.e., $dv(w)$, and adds $dv(w)$ to $dv(v)$. After *ddsv* is empty, we get the determining variables of v, $dv(v) = ddv(v) \cup \{dv(w) \mid w \in ddsv(v)\}$. Since there may be cycles in PDG, we use a mark "visited" to label those variables that have been found before, and further search is done only for the unmarked variables.

Given the definitions of property dependency graphs and noncorrelated sets, we can define a partitioned property dependency graph (PPDG) needed for illustrating our iterative reduction algorithms. The root of PPDG is the same root as PDG, the nodes of level n in PPDG are labeled by the noncorrelated sets of variables labeling the nodes of level n in PDG. If there exists an edge from node v to node w in PPDG, then w contains one noncorrelated set of direct determining variables of set v. Our iterative reduction algorithms start from the variables in P, i.e., the root, and at each iteration step add its one noncorrelated set of state variables to construct a reduced system. This iterative procedure can be viewed as a search of PPDG. In the following we will introduce the iterative reduction algorithms based on two different search strategies: depth-first and breadth-first.

7.3 A Depth-first Iterative Reduction Algorithm

First we explain the basic idea behind our algorithms. For instance, in the property dependency graph given in Figure 14, y_1 is directly determined by y_2 and y_3, and the two subgraphs G_1 and G_2 are disjoint, which means that y_2 and y_3 have no common determining variables, $dv(y_2) \cap dv(y_3) = \varnothing$. Thus the values of y_2 and y_3 are uncorrelated. When we consider the influence of y_2 on y_1, we can leave y_3 as a primary input, or vice versa. For verifying the property $\mathbf{AG}(y_1=1)$, we use y_1 and the state variables in G_1 to construct a reduced model and change the state variables in G_2 to primary inputs. If P holds on this reduced model, the procedure terminates, otherwise we use y_1 and the state variables in G_2 to construct another reduced model and change the state variables in G_1 to primary inputs. If P still fails, then we use y_1 and all the state variables in G_1 and G_2 to verify P.

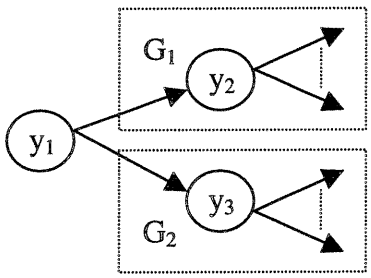


Figure 14. Example of a property dependency graph

The above procedure can be further refined, since using all state variables in a subgraph may not be necessary and may still produce a large state space. Beginning from the root of PDG, we search through the graph, partition the newly visited state variables into noncorrelated sets $S_1,\ldots, S_n$, and then select a set S_i, $1 \leq i \leq n$ and add it to the current set of state variables that was previously used to construct a reduced model. If the property fails on this model, we search the graph again beginning from S_i, and repeat the above procedure. We thus iteratively add state variables to construct a more and more complete model.

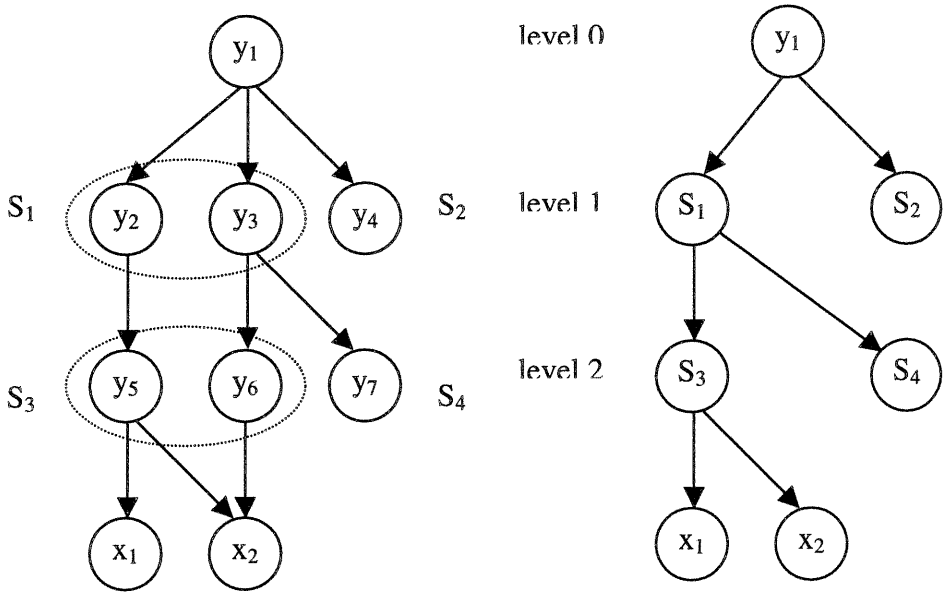


Figure 15. (a) A property dependency graph (PDG) (b) The corresponding PPDG

For the PDG shown in Figure 15 (a), y_1, ..., y_7 are state variables, x_1, x_2 are primary inputs, the property to be verified is **AG**($y_1 = 1$). Starting from the root y_1, we find $ddsv(y_1) = (y_2, y_3, y_4)$. The variables in $ddsv(y_1)$ can be partitioned to two noncorrelated sets $S_1 = (y_2, y_3)$ and $S_2 = (y_4)$. First we select S_1, and use $(y_7) \cup S_1$ to construct a reduced model. If P fails, we start from S_1 and find $ddsv(S_1) = (y_5, y_6, y_7)$. The variables in $ddsv(S_1)$ can be partitioned into two noncorrelated sets $S_3 = (y_5, y_6)$ and $S_4 = (y_7)$. Then we select S_3 and use $(y_7) \cup S_1 \cup S_3$ to construct a reduced model. If P still fails, since S_3 has no direct determining state variables, we select S_4 and use $(y_7) \cup S_1 \cup S_4$ to construct a reduced model. If P fails again, we use $(y_7) \cup S_1 \cup S_3 \cup S_4$ to construct a reduced model. If P still fails, we use $(y_7) \cup S_2$ to construct a reduced model. If P fails again, we use all the state variables in PDG, i.e., $(y_7) \cup S_1 \cup S_2 \cup S_3 \cup S_4$ to construct the final model that is now guaranteed to strongly preserve P. In Figure 15(b), the corresponding PPDG is also shown, in which a node represents a correlated set of variables. The above

procedure for searching a correlated set to be used in constructing a reduced model can be viewed as a depth-first search on PPDG.

The iterative reduction algorithm *Reduction_verify_depthfirst(M, P)* shown in Figure 16 accomplishes the above idea. Here M is the circuit model, and P is the property to be verified. Y_P contains the state variables appearing in P. Y' is the set of state variables that is used to compute the abstract model. DV_P is the set of property dependent state variables of P. *lastset* is the newest set added to Y'.

```
Reduction_verify_depthfirst(M, P)
Begin
    Y_P is the set of the state variables in P;
    dsv(Y_P) := Compute_dsv(Y_P, M);
    DV_P := Y_P ∪ dsv(Y_P);
    Y' := Y_P;
    Mark variables in Y' as visited;
    result := Verify_depthfirst(M, P, DV_P, Y', Y');
    If result == success
            print('Property checking succeeded');
    Else   print('Property checking failed');
End

Verify_depthfirst(M, P, DV_P, Y', lastset)
Begin
    M' := Reduce_model(M, Y');
    result := Modelcheck (M', P);
    If result == success
        Return success;
    Else if Y' == DV_P
        Return failure;
```

```
    Else
    Begin
      ddsv := Compute_ddsv(lastset, M);
      ddsv1 := Remove_visited_variables(ddsv);
      noncorrelated_sets := Partition_set(ddsv1, M);
      While noncorrelated_sets is not empty
      Begin
        Select newset ∈ noncorrelated_sets;
        noncorrelated_sets := noncorrelated_sets \ newset;
        Mark the state variables in the newset as visited;
        newY' := Y' ∪ newset;
        Verify_depthfirst(M, P, DVP, newY', newset);
      End
      dsv := Compute_dsv(lastset, M);
      newY' := Y' ∪ dsv;
      M' := Reduce_model(M, newY');
      result := Modelcheck (M', P);
      If result == success
      Return success;
      Else if newY' == DVP
      Return failure;
  End
End
```

Figure 16. A depth-first iterative reduction algorithm

The procedure *Compute_ddsv(Si, M)* computes all direct determining state variables of the set *Si*. *Compute_dsv(Si, M)* computes all determining state variables of *Si*. *Remove_visited_variables(ddsv)* removes the visited state variables from the set of the currently reached state variables, then further search from these

variables is prevented. *Reduce_model*(M, Y') reduces the other state variables except Y' to primary inputs. *Partition_set*(*Vars*, M) partitions the variables in *Vars* into noncorrelated sets.

The procedure *Verify_depthfirst* accomplishes the iterative reduction and model checking. It begins from the reduced model constructed by the variables in Y_P that are contained in the root of PDG. If property P holds on this model, then the verification finishes. Otherwise the algorithm checks if the abstract model was constructed using all the property dependent state variables DV_P. If yes, the algorithm terminates with a negative result. If not, it enlarges the model by adding more state variables. It begins with the last selected set, i.e., Y_P, and searches $ddsv(Y_P)$, the set of state variables of level 1 in PDG. The previously visited state variables are removed from $ddsv(Y_P)$ and the remaining variables are partitioned into sets of variables by *partition_set* such that in any pair of sets there are no common determining variables. The resulting noncorrelated sets are stored in *noncorrelated_sets* that is at level 1 here. Then a set S_1 in *noncorrelated_sets* is selected and added to Y' to construct the abstract machine, the other state variables become primary inputs. If the model does not satisfy P and does not contain all the variables from DV_P, the algorithm begins with the last selected set, i.e., S_1 and repeats the above procedure. If a set has no direct determining state variables, then another set in the same *noncorrelated_sets* is selected. If the *noncorrelated_sets* at level n is empty, that is, the determining state variables of any set in it have been used to construct a reduced model and none of these reduced models satisfy P, then all the determining state variables of all the sets in the *noncorrelated_sets* are used to construct a reduced model. If the property still fails, then the algorithm goes back to the *noncorrelated_sets* at level n-1, and selects another set.

For example, consider the circuit shown in Figure 17 where R_0, ..., R_5, R_{out} are registers, and a, b, c are free inputs.

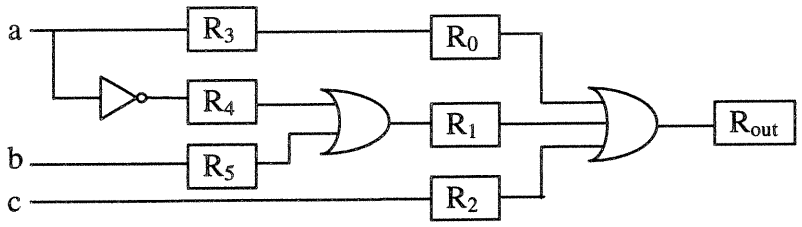


Figure 17. An example of circuit

If the initial values of all the state variables are 1, then the circuit has the property that the output R_{out} is always 1, **AG**(R_{out} = 1). The property dependency graph is shown in Figure 18.

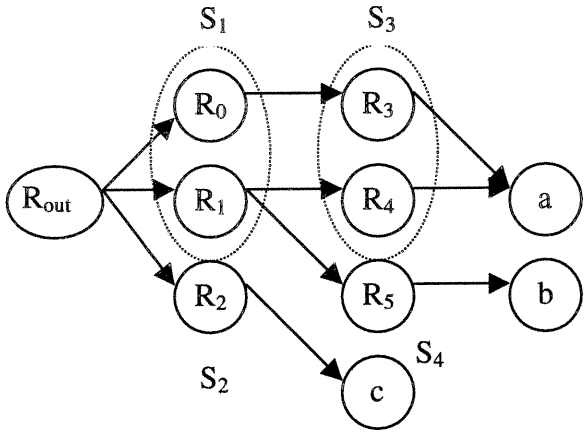


Figure 18. The PDG of the circuit shown in Figure 17

The procedure for verifying **AG**(R_{out} = 1) using our iterative reduction algorithm is as follows:

Iteration 1: The abstract model is constructed using only {R_{out}}, and the verification fails.

Iteration 2: The abstract model is constructed using {R_{out}, R_0, R_1}, and the verification fails.

Iteration 3: The abstract model is constructed using { R_{out}, R_0, R_1, R_3, R_4}, and the verification succeeds.

The property is verified with R_2 and R_5 eliminated.

We implemented the iterative reduction algorithm shown in Figure 16 in the MDG tool, the experimental results using MDG to verify **AG**(R_{out} = 1) on the circuit illustrated in Figure 17 are shown as follows.

MDG without reduction algorithm :

```
=== Performance statistics ===

  Compiling (loading+deriving all the relations) took:
     Run time : 0.210 seconds ; System time : 0.020 seconds ; Real
     time : 0.429 seconds .
  Building the initial state set MDG took:
     Run time : 0.000 seconds ; System time : 0.000 seconds ; Real
     time : 0.003 seconds .
  Property checking took:
     Run time : 0.060 seconds ; System time : 0.000 seconds ; Real
     time : 0.170 seconds .
  Total time spent:
     Run time : 0.270 seconds ; System time : 0.020 seconds ; Real
     time : 0.602 seconds .
  State variable coverage : 8 , 100% of all state variables.
  Nodes: 234;   Compound Terms: 1.
  Memory usage: 1275776 bytes.
  Garbage_collection 1 times: 0.010 seconds; 173164 bytes freed.
```

MDG with reduction algorithm :

```
=== Performance statistics ===

  Compiling (loading+deriving all the individual relations) took:
     Run time : 0.170 seconds ; System time : 0.010 seconds ; Real
     time : 0.379 seconds .
  Constructing ddp and dp tables took :
     Run time : 0.000 seconds ; System time : 0.010 seconds ; Real
     time : 0.009 seconds .
  Iterative reduction verification took :
     Run time : 0.280 seconds ; System time : 0.010 seconds ; Real
     time : 0.642 seconds .
  Total time spent:
     Run time : 0.450 seconds ; System time : 0.030 seconds ; Real
     time : 1.030 seconds .
  State variable coverage : 6 , 75% of all state variables.
  Nodes: 134;   Compound Terms: 1.
  Memory usage: 1069912 bytes.
  Garbage_collection 7 times: 0.040 seconds; 986400 bytes freed.
```

Our iterative reduction algorithm in MDG eliminated R_2 and R_5 automatically and verified the property using 75% of all the state variables, we can see that there are

134 MDG nodes, much less than without reduction (234). The memory usage is also less. For this small example, the property can be easily verified on the original system without reduction and the run time is smaller since the reduction method takes several iterations. But for large systems, iterative reduction methods use less memory and less time and help verify properties that cannot be verified by the original systems, especially in the cases that the original systems lead to the state explosion.

We also used FormalCheck and SMV to verify $\mathbf{AG}(R_{out} = 1)$. We selected the Iterated algorithm in FormalCheck and the property was verified with all the state variables used, as shown in the "Reduction Manager" window. When we eliminated R_2 and R_5 manually by setting them free in the Reduction Manager window, the property was verified faster. When we selected automatic reduction in SMV, all the state variables were used to verify the property with 238 BDD nodes. When we manually eliminated R_2 and R_5 by setting them free in the "Abstraction" window, the property was verified with fewer BDD nodes (186). From the comparison we can see that our reduction algorithm can automatically find the state variables that could be reduced while other tools fail to find them in this case.

The algorithm shown in Figure 16 randomly selects one set from the *noncorrelated_sets*. For a large design, this set may contain many state variables, and using it to construct the abstract system may still lead to state explosion. We modified the algorithm to select the smallest set in the *noncorrelated_sets* at each step. We added one procedure *Sort_sets(noncorrelated_sets)* to sort the sets in increasing order of their sizes and store the result in a list. Each time the algorithm selects the set at the head of this list. That is, the smallest unused set is added to build the abstract machine. This may reduce the chance of the state explosion and avoid the situation that a property cannot be verified by the reduced model using a large set in the *noncorrelated_sets* while it could be verified by the reduced model using a smaller set.

7.4 A Breadth-first Iterative Reduction Algorithm

In the preceeding section we introduced the reduction algorithm based on a depth-first search of the partitioned property dependency graph. In this section, we present an iterative reduction algorithm based on a breadth-first search of the PPDG. The algorithm *Reduction_verify_breadthfirst(M, P)* is shown below. Here M is the circuit model and P the property to be verified. Y_P contains the state variables appearing in P. Y' is the set of state variables that is used to compute the abstract model. DV_P is the set of property dependent state variables of P. *lastset* is the newest set added to Y'.

```
Reduction_verify_breadthfirst(M, P)
Begin
    Y_P is the set of the state variables in P;
    dsv(Y_P) := Compute_dsv(Y_P, M);
    DV_P := Y_P ∪ dsv(Y_P);
    Y' := Y_P;
    Mark the state variables in Y' as visited;
    result := Verify_breadthfirst(M, P, DV_P, Y', Y');
    If result == success
        print('Property checking succeeded');
    Else  print('Property checking failed');
End

Verify_breadthfirst(M, P, DV_P, Y', lastset)
Begin
    M' := Reduce_model(M, Y');
    result := Modelcheck (M', P);
```

```
    If result == success
        return success;
    Else if Y' == DVP
        return failure;
        Else
        Begin
            ddsv := Compute_ddsv(lastset, M);
            ddsv1 := Remove_visited_variables(ddsv);
            noncorrelated_sets := Partition_set(ddsv1, M);
            listofsets := Sort_sets(noncorrelated_sets);
            While listofsets is not empty
            Begin
                newset := head of listofsets;
                listofsets := listofsets \ newset;
                Mark the state variables in the newset as visited;
                newY' := Y' ∪ newset;
                M' := Reduce_model(M, newY');
               result := Modelcheck (M', P);
               If result == success
                  return success;
               Else if Y' == DVP
                  return failure;
            End
            newY' := Y' ∪ ddsv1;
            Verify_breadthfirst(M, P, DVP, newY', ddsv1);
        End
End
```

Figure 19. A breadth-first iterative reduction algorithm

Reduction_verify_breadthfirst uses the variables in Y_P that appear in P to construct the first abstract machine. If P is satisfied then the procedure terminates. If P is not satisfied and not all of DV_P have been used, the direct determining state variables of Y_P are obtained by *Compute_ddsv* and partitioned into noncorrelated sets by *Partition_set*. The noncorrelated sets are sorted in an increasing order of their sizes, and the result is stored in *listofsets*, the list at level 1. Each time the smallest set in the remaining *listofsets* is selected to construct the abstract machine to verify P. If P cannot be proved on the abstract machines by selecting any individual set in *listofsets*, then all the sets in *listofsets* are used together to construct the reduced machine to verify P. If P is still not satisfied, the direct determining state variables of all variables in the sets in *listofsets* are obtained and partitioned into noncorrelated sets that are stored in the list at level 2. Then the set in the list at level 2 is iteratively selected and the above process is repeated. Verification terminates when property P is satisfied or all state variables in DV_P are used.

If we use the breadth-first reduction algorithm to verify P: $\mathbf{AG}(R_{out} = 1)$ on the circuit in Figure 17, referring to the PDG in Figure 18, the iterations are as follows:

Iteration 1: $Y' = \{R_{out}\}$, property P fails.

Iteration 2: $Y' = \{R_{out}, R_2\}$, property P fails.

Iteration 3: $Y' = \{R_{out}, R_0, R_1\}$, property P fails.

Iteration 4: $Y' = \{ R_{out}, R_0, R_1, R_2\}$, property P fails.

Iteration 5: $Y' = \{ R_{out}, R_0, R_1, R_2, R_5\}$, property P fails.

Iteration 6: $Y' = \{ R_{out}, R_0, R_1, R_2, R_3, R_4\}$, property P succeeds.

The verification succeeds after six iterations and one state variable R_5 is reduced. For this example, the iterative reduction algorithm based on the depth-first search of PPDG gets the better result, i.e., the verification succeeded after 3 iteration steps and two state variables R_2, R_5 were reduced.

It is hard to say which algorithm is better. For the *MinMax* example, we will see that the breadth-first reduction algorithm works better. The partitioned property dependency graph (PPDG) of Figure 11 is shown in Figure 20.

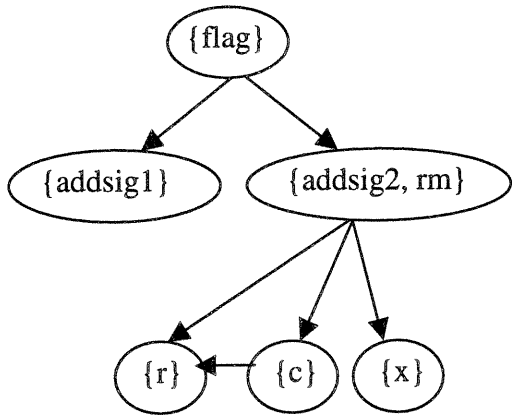


Figure 20. PPDG of *MinMax* example

If we use the algorithm based on the breadth-first search of PPDG to verify property P:**AG**(*flag* = 1), the verification proceeds as follows:

Iteration 1: Y' = {*flag*}, property P fails.

Iteration 2: Y' = {*flag*, *addsig*1}, property P fails.

Iteration 3: Y' = {*flag*, *addsig*2, *rm*}, property P fails.

Iteration 4: Y' = {*flag*, *addsig*1, *addsig*2, *rm*}, property P succeeds.

The verification succeeds after four iterations and two state variables {*c*, *rM*} are eliminated.

If we use the iterative reduction algorithm based on depth-first search of PPDG to verify the property P: **AG**(*flag* = 1), the verification proceeds as follows:

Iteration 1: Y' = {*flag*}, property P fails.

Iteration 2: Y' = {*flag*, addsig1}, property P fails.

Iteration 3: Y' = {*flag*, *addsig*2, *rm*}, property P fails.

Iteration 4: Y' = {*flag*, *addsig*2, *rm*, *c*}, property P fails.

Iteration 5: Y' = {*flag*, *addsig*1, *addsig*2, *rm*, *c*}, property P succeeds.

The verification succeeds after five iterations and one state variable *rM* is reduced. For this example, the iteration reduction algorithm based on the breadth-first search of PPDG reduced more state variables and ran faster.

In general, since there are many different designs, it is hard to say which algorithm is better. We provide users with the two algorithms, one based on the depth-first search of PPDG, and the other one based on the breadth-first search of PPDG. There are generally more iteration steps before all the state variables in DV_P are included in the depth-first search than in the breadth-first search algorithm. That is, the depth-first search adds state variables more slowly and may succeed with fewer variables, however, the iterations also take time. Hence more experience is needed on large models to see where either algorithm performs better.

When a property should be falsified, i.e., there is a bug in the design, the iterative reduction algorithms may take more time. Since the abstract models constructed using a subset of property dependent state variables of P only weakly preserve P, when P is falsified on an abstract model, the algorithms add more state variables iteratively until all variables in DV_P are used. The good thing is that the state variables not in DV_P are still reduced in this situation and this may avoid state explosion that could happen using the whole design.

7.5 Complexity Analysis of the Algorithms

We introduced two iterative reduction algorithms based on functional dependency of the property. First, the property dependency graph is constructed that is implicitly represented by a hash table *ddv* of the direct determining variables of the state variables. Second, the hash table *dv* for the state variables in PDG is constructed. Third, the PDG is partitioned to PPDG. Fourth, reduced models are constructed by iteratively selecting sets of state variables based on two search strategies on PPDG.

The algorithm for constructing the hash table *ddv* is shown in Figure 7. The time to construct *ddv* is the time to find all primary variables in the MDGs representing the transition relations. Since the primary variables label MDG nodes, to find all the primary variables is to visit every node in MDGs. Thus the complexity of constructing *ddv* table is $\theta(N_{MDG})$, where N_{MDG} is the number of MDG nodes and in the worst case it is 2^N, where N is the number of state variables and inputs in the original model.

The algorithm for constructing the hash table *dv* is shown in Figure 13. The time to construct *dv* is the time to visit all the state variables in the PDG, i.e., $\theta(N_{PDG})$. Here N_{PDG} is the number of nodes in PDG, that is, the number of property dependent state variables in DV_P. In the worst case, N_{PDG} is the number of the state variables in the original machine.

The iterative reduction algorithms are shown in Figure 16 and Figure 19. The PPDG is implicitly represented by the noncorrelated sets and is constructed iteratively during reduction iterations. When a new set of state variables in PDG is reached, these variables are partitioned into noncorrelated sets by the procedure *Partition_set* shown in Figure 12. The time to partition a set is $\theta(N_{set})$, where N_{set} is the number of the state variables in this set. In the worst case when all the state variables in PDG are reached, PPDG is completely constructed. Since each state variable in PDG is visited only once, the time to construct PPDG is $\theta(N_{PDG})$. In the worst case, N_{PDG} is the number of the state variables in the original machine.

The two iterative reduction algorithms are based on a depth-first search or a breadth-first search on PPDG. In both of these two methods, in the worst case when all the property dependent state variables are necessary for verifying the property, the time to iteratively select the sets of state variables is $\theta(N_{PPDG})$, where N_{PPDG} is the number of the nodes in PPDG. In the worst case, N_{PPDG} is the number

of the state variables in the original model. When one set of variables is selected to construct the reduced model, the other state variables are changed to primary inputs and the time of reducing the model is $\theta(N_{stvar})$, where N_{stvar} is the number of state variables to be reduced. In the worst case, N_{stvar} is of the same order as the number of state variables in the original model.

The total time for iteratively constructing the reduced models is thus $\theta(N_{MDG}) + 2 \times \theta(N_{PDG}) + \theta(N_{PPDG}) \times \theta(N_{stvar})$. In the worst case the time is $\theta(2^N)$, where N is the number of state variables and inputs in the original model.

Summary

In this chapter, we introduced two iterative reduction algorithms based on function dependency. We defined a property dependency graph (PDG) and noncorrelated sets of state variables, and then we defined the partitioned property dependency graph (PPDG). The algorithms construct reduced models starting from the state variables appearing in the property. If the reduced model does not satisfy the property, then a set of state variables is selected to construct a more detailed model. If the reduced model satisfies the property, the verification is finished. If all intermediate reduced models do not satisfy the property, then the model constructed using all the state variables in PDG is used. There are two iterative reduction algorithms depending on the search strategy of the PPDG, depth-first and breadth-first. In the next chapter we will introduce the implementation of our reduction algorithms in the MDG model checker.

Chapter 8 Integration of Reduction Algorithms with MDG Model Checker

In the previous chapters we have introduced the reduction algorithms based on circuit topology or functional dependency. In this chapter we will introduce some implementation issues relative to the integration of our reduction algorithms in the MDG model checker.

8.1 Implementation of the Reduction Algorithms

The MDG model checker verifies properties expressed in L_{MDG}. This tool has two separate subsystems. One is the property compiler and the other one is the model checking engine. The property compiler accepts a design written in the MDG-HDL language and a property in L_{MDG}, and constructs the additional ASMs for the property by translating the formula into an MDG-HDL net list. It then combines the additional circuit with the original design. Then model checking engine accepts the composite machine and the simplified property and verifies the simplified property on the composite machine.

The reduction algorithms are integrated with the model checking engine. Reduction options are provided for users to decide if they want to use reduction and which reduction algorithm they prefer. The reduction methods 1, 2, and 3 are the reductions based on the depth-first search of PPDG, breadth-first search of PPDG and the circuit topology respectively. There are 14 types of properties accepted by the model checker. The user must select the type of the property. Then the main model checking procedure revokes the appropriate subprocedures to verify the property. The overall flow chart of the integration is shown in Figure 21.

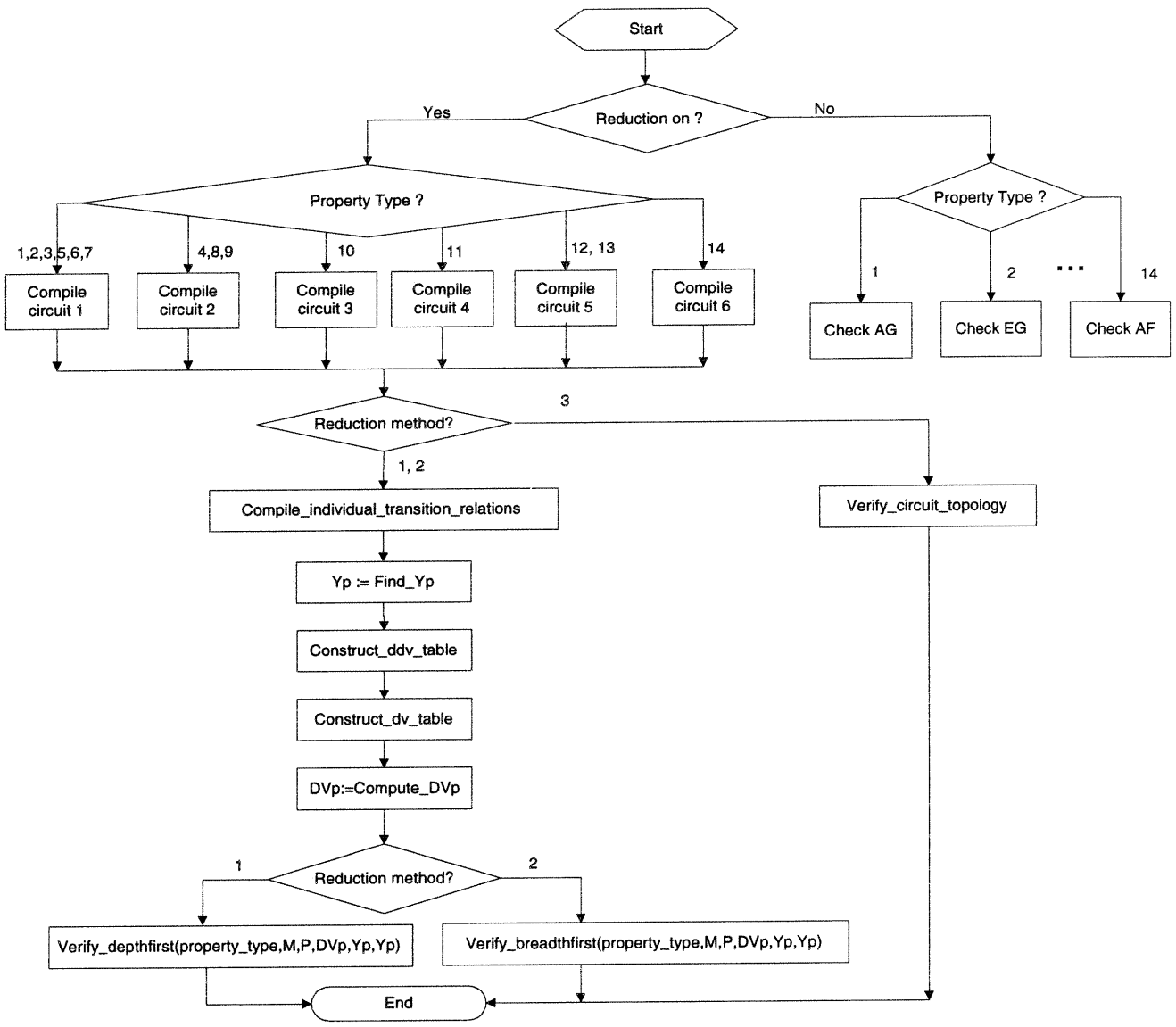


Figure 21. Flow chart of MDG Model checking main procedure

If "no reduction" is selected, then the appropriate model checking algorithm without reduction is used according to the type of the property. If "reduction" is selected, the circuit is compiled and a database is constructed using different procedures for different types of properties. If the reduction method 1 or 2 is selected, the MDGs of the individual transition relations are constructed. The state variables Y_P in the additional circuit representing the property P are found by name matching. Then the hash tables of direct determining variables and determining variables are constructed, and the set of the property dependent state variables DV_P

is obtained. According to the selection of the depth-first search or the breadth-first search, the appropriate iterative reduction algorithm is used. If the reduction method 3 is selected the iterative reduction algorithm based on circuit topology is used. The flow charts of *Verify_depthfirst*, *Verify_breadthfirst* and *Verify_circuit_topology* are shown in Figure 22, Figure 23 and Figure 24 respectively.

The procedure *Find_*Y_P is used to find the state variables in the additional ASM for a property P. In next section we will explain why we use these state variables as Y_P rather than the flags in the simplified property. Since all the signals in the additional ASM are named by words beginning with '*addedsignal*' or '*v*' or '*flag*', which are reserved key words, it is easy to find the state variables in the additional ASM that form Y_P. The first abstract machine is constructed using the variables in Y_P, and then noncorrelated sets of state variables are added iteratively to construct more detailed abstract machines.

The procedures *Verify_depthfirst* and *Verify_breadthfirst* are similar to those in Chapter 7, and the procedure *Verify_circuit_topology* is similar to the iterative reduction algorithm in Figure 6, except that they have one more augument *property_type*, and the procedure *Modelcheck* also has *property_type* as one augument. *Modelcheck*(*property_ type,* M_r, *P'*) completes property checking on the reduced machine M_r using the appropriate algorithm depending on the type of the property.

When the MDG model checker reads in the composite machine, it produces a database that includes the tables of output variables, input variables, abstract input variables, the pairs of state variables and the corresponding next state variables, initial signals, initial values, and initial variables. This database is useful for constructing the transition system and for completing the model checking. When the reduction algorithm eliminates some state variables, this database needs to be updated. This is completed by the procedure *Reduce_model* and *Change_circuit*.

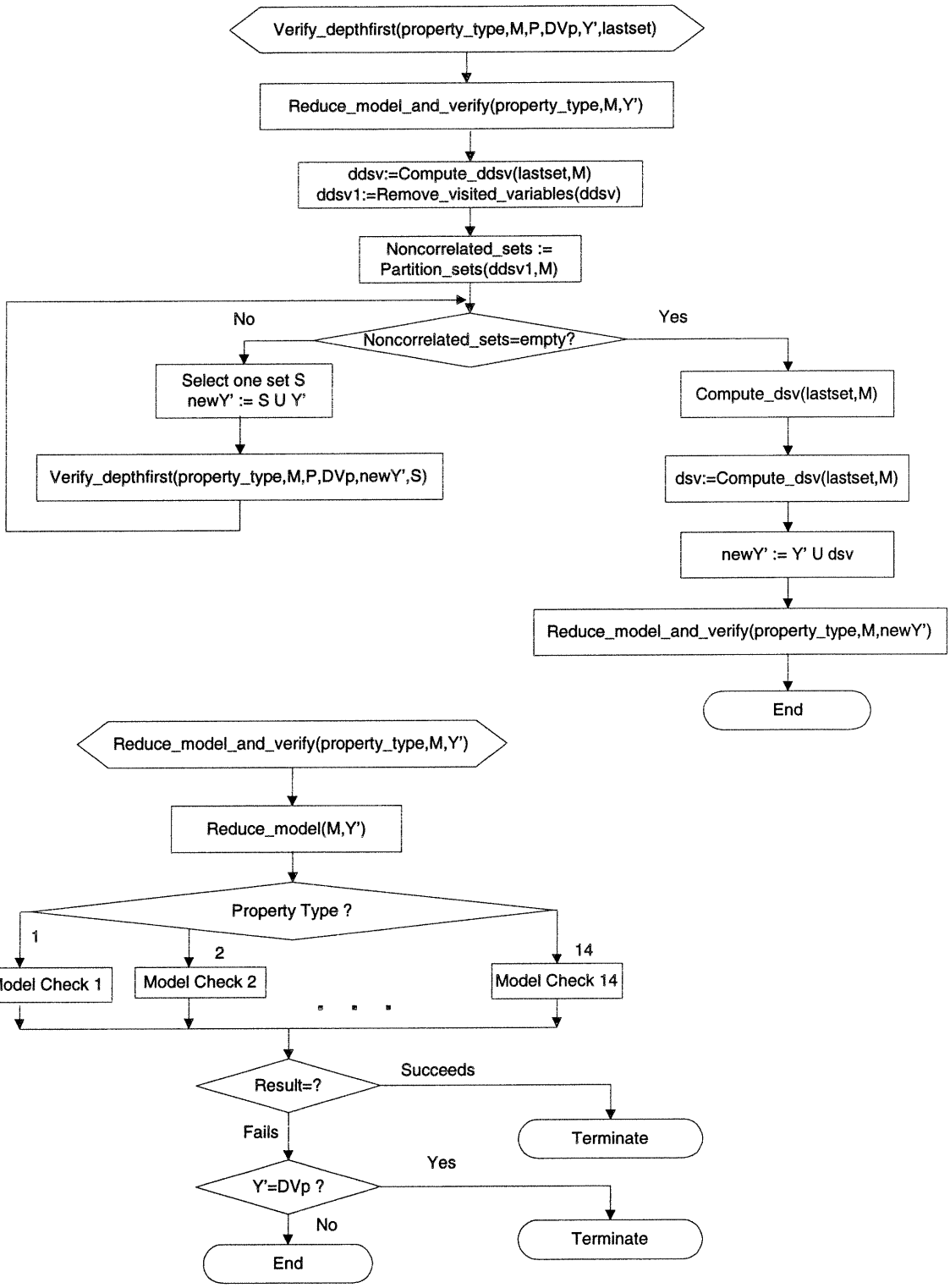


Figure 22. Flow chart of procedure *Verify_depthfirst* in MDG model checker

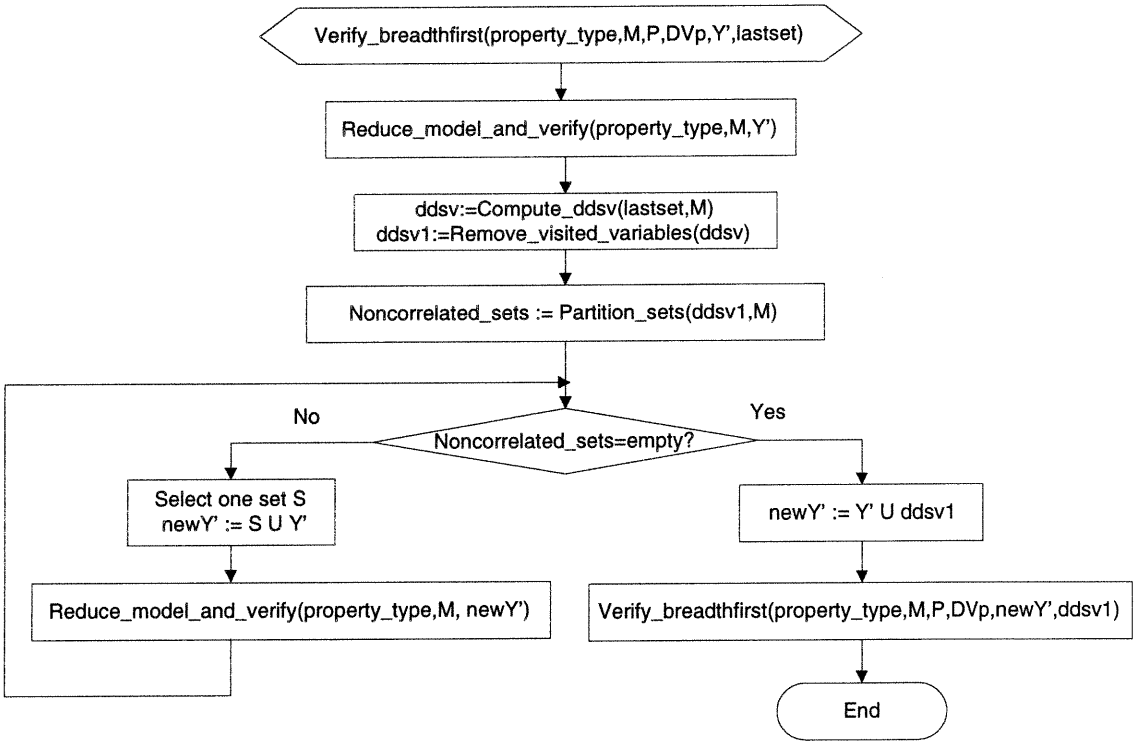


Figure 23. Flow chart of procedure *Verify_breadthfirst* in MDG model checker

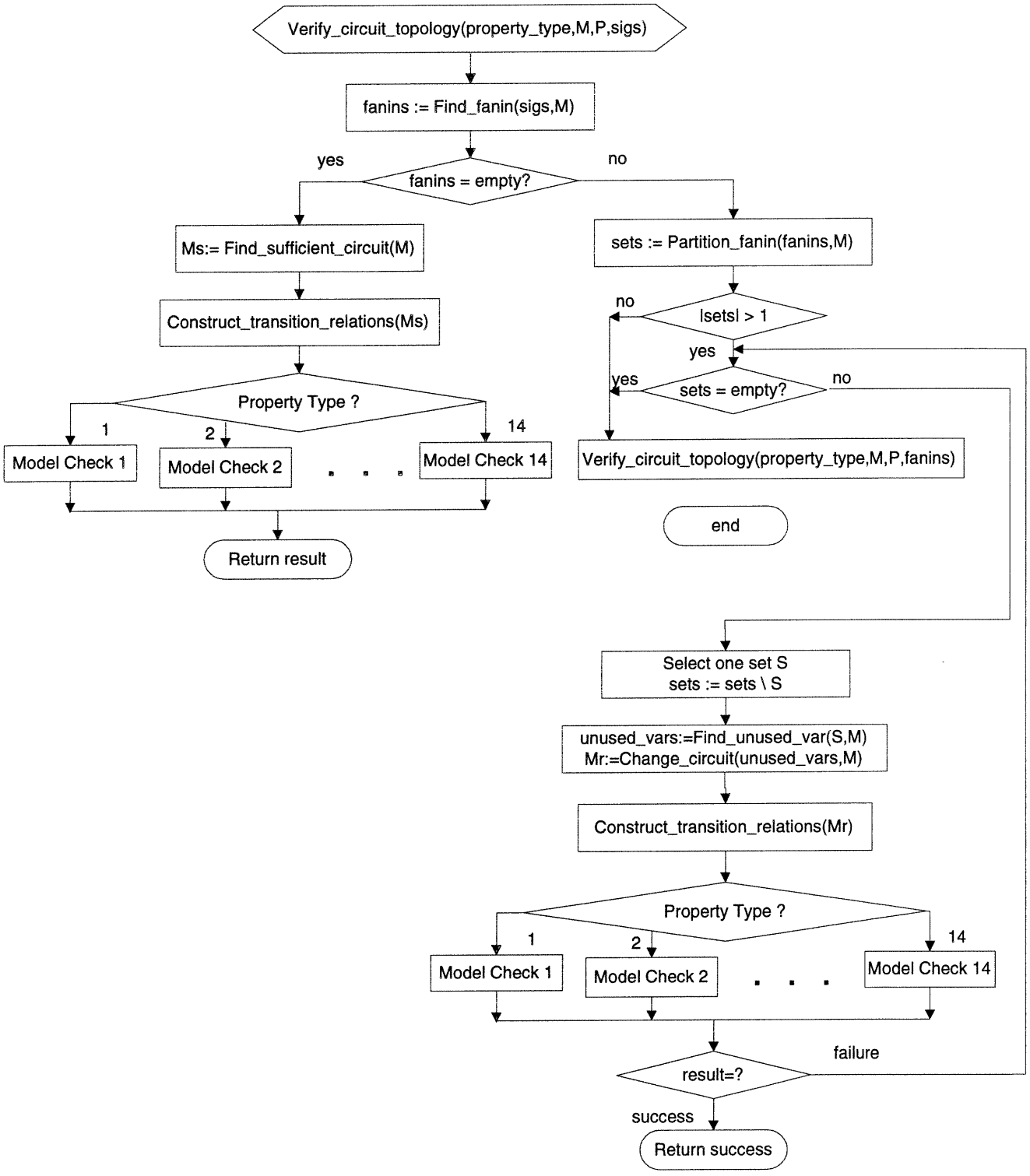


Figure 24. Flow chart of procedure *Verify_circuit_topology* in MDG model checker

8.2 Selection of the Starting State Variables

In this section we want to emphasize that a suitable selection of the starting state variables Y_P for building reduced models can lead to more efficient reductions. We will explain why we use the state variables in the additional ASM of property P as Y_P to construct reduced models for the MDG model checker.

Given a design and a property to be verified, we want to use the information provided by the property to reduce the original model. In the MDG model checker, an additional ASM machine represents the original property, and the verification of the original property on the design is transferred to the verification of a simplified property on the composite machine. Since the transferred property is too simple, we cannot get from it much information about the original property. Because of the particular way to generate the additional ASM for a property, all the added signals are necessary for the flag being true and they are in different correlated sets. If we start from the flag in the simplified property, at the next iteration only a subset of the added signals is used, and the flag check should fail. The reduction verification using a breadth-first search of PPDG takes more iteration steps to finish, but in this case the reduction algorithm based on a depth-first search cannot eliminate any state variable in DV_P. Therefore, we start with the state variables of the composite machine appearing in the additional ASM that directly represents the original property. This makes the iterative reduction algorithms terminate faster and eliminate state variables more efficiently. We will illustrate this using the MinMax example.

In the MinMax example, the additional ASM representing the property $\mathbf{AG}((r = 1) \rightarrow \mathbf{X}(m = max))$ is shown in Figure 3. The state variables appearing there are {*flag*, *addsig*1, *addsig*2, *m*}. The short prefix *addsig* represents the original prefix *addedsignal*. The property dependency graph of this example is shown in Figure

11. The way to define the initial values of *addsig*1 and *flag* in this example guarantees that the value of *flag* is 1 in the first two clock cycles, and thereafter the value of *flag* is determined by the functional parts of the circuit. Without *addsig*1 the property **AG**(*flag* = 1) should fail. Using the reduction algorithm based on a depth-first search, at first iteration {*flag*} is used to construct the abstract machine and *P* fails. The direct determining state variables of *flag* are found which are {*addsig*1, *addsig*2, *rm*}. They are partitioned into two noncorrelated sets, {*addsig*1} and {*addsig*2, *m*}. {*addsig*1} is used with *flag* at iteration 2 to construct the abstract machine, *P* fails. {*addsig*2, *rm*} is then used with *flag* to verify *P*, but *P* still fails. Then *c* is added at iteration 4, but *P* fails again. Finally at iteration 5 all the state variables in PDG are used, *P* is satisfied on the final abstract machine with the *don't care* state variable *rM* eliminated.

On the other hand, if we use the state variables appearing in the property ASM {*flag*, *addsig*1, *addsig*2, *m*} to construct the reduced machine at the first iteration, the property **AG**(*flag* = 1) is satisfied and the verification is finished with state variables *c* and *rM* eliminated. Comparing with the reduction that starts from {*flag*}, the reduction that starts from the state variables in the additional ASM can eliminate more state variables in less time. The experimental results of verifying **AG**((*r* = 1) → **X**(*m* = *max*)) on the *MinMax* machine are as follows:

```
Iteration 1 :
Used state variables: [m, flag, addedSignal1, addedSignal2]

=== Checking_AG succeeded ===

=== Circuit statistics ===

Total components: 14
Total signals:    19
  Abstract signals: 4
  Concrete signals: 15  which is equivalent to 15 boolean signals
Total state variables:  6
  Abstract state variables:  2
  Concrete state variables:  4  which is equivalent to 4 boolean
  variables
```

```
=== Performance statistics ===

  Compiling (loading+deriving all the individual relations) took:
     Run time : 0.180 seconds ; System time : 0.000 seconds ; Real
     time : 0.475 seconds .
  Constructing ddv and dv tables took :
     Run time : 0.010 seconds ; System time : 0.000 seconds ; Real
     time : 0.010 seconds .
  Iterative reduction verification took :
     Run time : 0.090 seconds ; System time : 0.000 seconds ; Real
     time : 0.250 seconds .
  Total time spent:
     Run time : 0.280 seconds ; System time : 0.000 seconds ; Real
     time : 0.735 seconds .
  State variable coverage : 4 , 67% of all state variables.
  Nodes: 224;   Compound Terms: 9.
  Memory usage: 1126760 bytes.
  Garbage_collection 14 times: 0.150 seconds; 2421448 bytes freed.
```

Summary

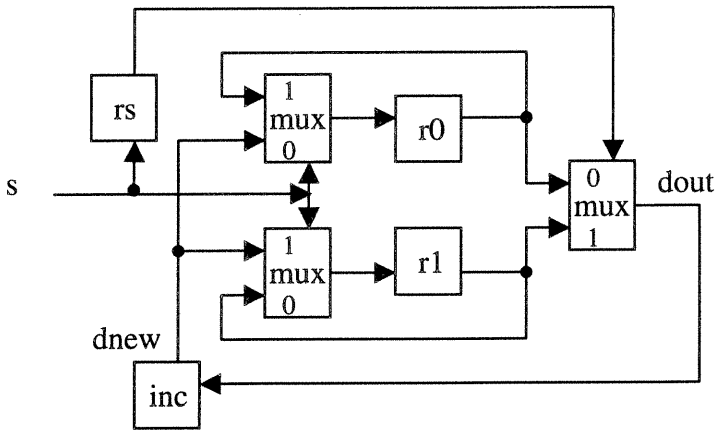
Our iterative reduction algorithms were implemented in the MDG model checker. We have carried out several experiments using MDG with and also without model reductions. These experiments showed that our reduction algorithms enlarge the useful domain of the MDG tool. Although our reduction algorithms are particularly useful for the MDG model checker, since they need not compute a bisimulation relation or use a preimage operation, they can be used in other tools as well. We used FormalCheck and SMV to compare the performance of reduction. The experiments showed that our reduction algorithms can reduce the model more efficiently on a large number of designs. In the next chapter we study two such cases.

Chapter 9. Case Studies

In this chapter we consider a very common circuit in data processing and digital telecommunication designs, and a benchmark design called the Island Tunnel Controller. We use the MDG model checker, FormalCheck and SMV to verify a number of properties. The experimental results show that our reduction algorithms have largely improved the behavior of the MDG model checker and can achieve efficient model reduction where other tools fail.

9.1 A Common Data Processing Circuit

Consider the example discussed in [XCSCLP99] and shown in Figure 25. The structure of the circuit is quite common in data processing circuits. The appropriate context (set of registers, memory data, etc.) is selected based on the control signals (address of the memory, etc.), processing is carried out on the selected context, and then the modified context is stored in the same memory element. It is also quite common in telecommunication circuits in which channel or link numbers select the corresponding registers to be updated. This structure can be easily enlarged by adding more registers and increasing the size of the registers.



flip-flops: {rs, r0, r1}, operation: inc

Figure 25. A data processing circuit

In Section 9.1.1, all the signals in the circuit are defined at the concrete Boolean level. We use three verification tools MDG, FormalCheck and SMV to verify the model with different numbers and sizes of registers. The experimental results are then discussed. Since in MDG there are abstract variables and uninterpreted function symbols, we give an abstract description of the circuit in Section 9.1.2 and verify it using MDG. The experimental results are again discussed. All of the experiments were carried out on a 333 MHz Sun Ultra 10 workstation with 1GB of memory. In the following tables, the symbol '-' means that the verification did not terminate.

9.1.1 Property Checking on a Concrete Model

All the signals in the circuit shown in Figure 25 are defined as Boolean. The registers are defined to have a certain number of bits. Property P_1 states that if s is 0, rs is 0, and the value of register $r0$ is 0 in the current clock cycle, then the value of $r0$ will be 1 in the next clock cycle. Property P_2 states that if s is 0, rs is 0 and the value of $r0[0]$ is 0 in the current clock cycle, then the value of $r0[0]$ will be 1

in the next clock cycle. We verified these two properties on the models with different numbers and sizes of registers.

Table 1 shows the results obtained using the MDG model checker. We can see that without our reduction algorithm, the tool can only verify the models having two registers with widths less than 20 bits. The reduction algorithm has significantly increased the useful domain in this case. When the number of registers is increased to 12 and the width is increased to 28 bits, P_1 and P_2 can still be verified using our reduction algorithm.

Table 1: Experimental results with MDG

Property	Register No.& Width	No reduction				Reduction			
		State Vars	Nodes	Time (Sec)	Mem (MB)	State Vars	Nodes	Time (Sec)	Mem (MB)
P_1	2&8	20	1554	2.42	2.05	11	1249	2.41	1.72
	2&16	36	4083	4.73	3.94	19	3131	5.60	3.24
	2&20	-	-	-	-	23	5635	10.26	4.77
	2&28	-	-	-	-	31	9390	19.56	8.54
	12&28	-	-	-	-	31	42247	3863.6	517.8
P_2	2&8	20	1199	1.77	1.90	4	506	1.86	1.36
	2&16	36	3057	3.66	2.46	4	866	3.22	2.30
	2&20	-	-	-	-	4	1092	5.27	3.19
	2&28	-	-	-	-	4	1468	11.51	5.81
	12&28	-	-	-	-	4	23447	4006.6	507.3

P_1 illustrates one behavior of $r0$, which only refers to the boolean signals of $r0$, but does not refer to the other registers. Our algorithm automatically eliminates all the Boolean signals of the other registers. We can also see that for the models with 28 bit registers, no matter how many registers are added, there are always only 31 state variables used to verify P_1. These include $r0[0]$,..., $r0[27]$, and three

additional state variables in the auxiliary circuit for the property. P_2 only refers to $r0[0]$, and our algorithm automatically eliminates the other bits of $r0$ and all the other registers. Table 1 shows that only 4 state variables are used to verify P_2 no matter how many and how large the registers in the model are.

It should be pointed out that MDG compilation process constructs MDG graphs for each signal and each component which consumes a lot of memory and time. When the sizes of the circuits increase, memory and time usage increase significantly.

For instance, we let the model have two registers $r0$ and $r1$ with 28 bits, each bit being represented by one Boolean signal. The model is written in MDG-HDL which does not have a means to describe arrays. This makes the description quite long and thus it is not included here. We use P_2 to illustrate how the reduction algorithm works. P_2 is expressed by the following L_{MDG} formula:

$$\mathbf{AG}\ ((s = 0\ \&\ rs = 0\ \&\ r0\_0 = 0) \text{ -> } (\mathbf{X}\ (r0\_0 = 1)));$$

The additional ASM extracted from P_2 is shown in Figure 26, in which *flag*, *addsig*1 and *addsig*2 are state variables. The initial values of *flag*, *addsig*1 and *addsig*2 are 1, which guarantees that *flag* = 1 during the first 2 clock cycles. The verification of P_2 on the original model is transferred to the verification of P_2': **AG**(*flag* = 1) on the composite ASM consisting of M and M_{P2}.

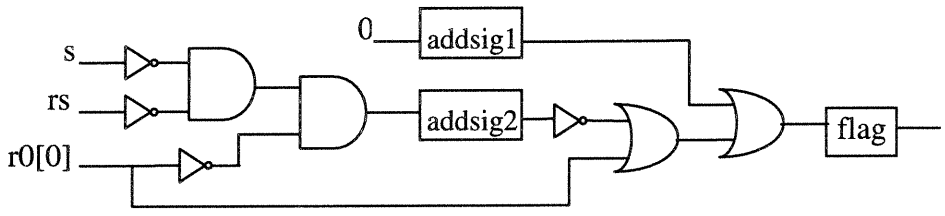


Figure 26. Additional ASM M_{P2} for P_2

The property dependency graph of P_2' is shown in Figure 27. Node s is a primary input and the other nodes are the state variables. At the first iteration of the

reduction algorithm, the set of the state variables {*flag*, *addsig*1, *addsig*2, *r*0[0]} is used to construct the reduced model, and the simplified property **AG** (*flag* = 1) is verified. Thus only 4 state variables are used to verify P_2 as shown in Table 1. In the worst case, if the design has some errors and P_2 should fail, i.e., the initial reduced model constructed using {*flag*, *addsig*1, *addsig*2, *r*0[0]} cannot satisfy the property **AG** (*flag* = 1), then the set {*rs*, *r*1[0]} is added to construct the model for the second iteration. Now all the state variables in the property dependency graph are used, and the verification is final. To verify P_2, only 6 state variables are needed in the worst case.

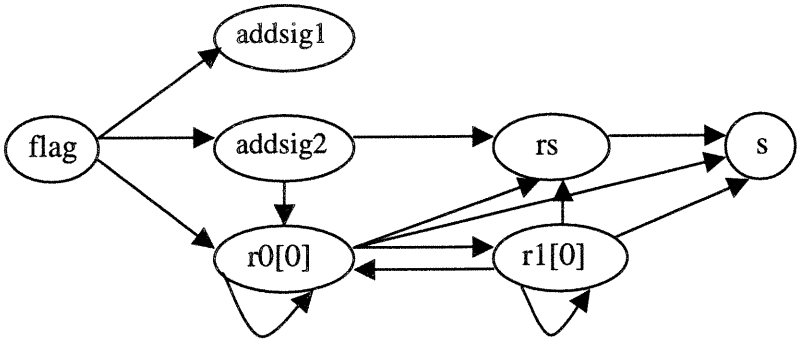


Figure 27. The property dependency graph of P_2'

We now will use FormalCheck and SMV to verify the models of the circuit shown in Figure 25. We again use the model with 2 registers of 28 bits. The model written in synthesizable Verilog for FormalCheck is as follows:

```
module main(s, RST, CLK, dout);
   input s, RST, CLK;
   output [27:0] dout;
   reg [27:0] r0, r1;
   reg rs;
   wire [27:0] dnew, dout;
   assign dout = (rs == 1'b0)? r0 : r1;
   assign dnew = dout + {{6{4'b0000}}, 4'b0001};
```

```
    always @(posedge CLK)
    begin
       if (RST == 1'b0)
       begin {r0,r1} <= {8{7'b0000000}}; end
       else  begin
          rs <= s;
          case (s)
             1'b0: r0 <= dnew;
             1'b1: r1 <= dnew;
          endcase
       end
    end
endmodule
```

Properties P_1, P_2 expressed in FormalCheck become:

```
P1:  After: (CLK==rising && RST!=0 && s==0 && rs==0 && r0==0)
       Always: r0==1
       Unless after: CLK==rising.

P2:  After: (CLK==rising && RST!=0 && s==0 && rs==0 && r0[0]==0)
       Always: r0[0]==1
       Unless after: CLK==rising.
```

In SMV the above model and the properties P_1 and P_2 were rewritten in Synchronous Verilog (SV), which requires some modifications to the original Verilog code. P_1 and P_2 expressed in SV are as follows:

```
P1:   always
        begin
            if (RST==1 & s==0 & rs==0 & r0==0)
            begin  wait(1);  assert r0_update: r0==1;  end
        end
```

```
P2:   always
        begin
            if (RST==1 & s==0 & rs==0 & r0[0]==0)
            begin  wait(1);  assert r0_0 update: r0[0]==1;  end
        end
```

The experimental results are shown in Table 2.

Table 2: Experimental results with FormalCheck, SMV and MDG (reduction on)

Property	Register No.& Width	FormalCheck			SMV			MDG		
		State Vars	Time (Sec)	Mem (MB)	State Vars	Time (Sec)	Mem (MB)	State Vars	Time (Sec)	Mem (MB)
P_1	2 & 16	26	26	3.39	34	0.29	8.40	19	5.6	3.24
	2 & 20	30	26	3.48	42	0.38	8.42	23	10.26	4.77
	2 & 28	38	37	3.73	58	0.63	8.42	31	19.56	8.54
	4 & 28	39	67	5.04	115	5.79	8.57	31	72.04	29.85
	8 & 28	40	131	9.24	228	7.28	9.52	31	759.8	165.7
	10 & 28	41	213	12.84	285	10.87	10	31	1820	315.3
	12 & 28	41	301	16.30	-	-	-	31	3863	517.8
P_2	2 & 16	38	1772	68.12	4	0.06	8.24	4	3.22	2.30
	2 & 20	49	32760	961.2	4	0.08	8.25	4	5.27	3.19
	2 & 28	-	-	-	4	0.08	8.25	4	11.51	5.81
	4 & 28	-	-	-	7	0.18	8.37	4	66.32	26.04
	8 & 28	-	-	-	12	0.49	9.03	4	780.1	158.8
	10 & 28	-	-	-	15	0.81	9.42	4	1886	306.6
	12 & 28	-	-	-	17	1.09	9.84	4	4006	507.3

For SMV, the column Time indicates user time, while for FormalCheck and MDG, it is real time including loading the Verilog or MDG-HDL description files, compilation and model checking. The reduction algorithm selected in

FormalCheck is Iterated with Empty reduction seed, and the run option is Symbolic (BDD). The run option of SMV uses heuristic variable ordering, computes the number of reachable states and restricts model checking to reachable states. The iterative reduction algorithm is selected in MDG.

Many different factors influence the experimental results, e.g., these tools use different variable ordering (no automatic variable ordering in MDG yet), different partitioning of the transition relations and different reduction methods. MDG graphs require more memory than the other model checkers in the case of concrete representations of signals, because the MDG structure and algorithms are more complicated than those of ROBDD to take into account abstract sorts (even though they were not used in this experiment). However, the columns indicating the number of state variables illustrate that our reduction algorithm can reduce the models appropriately according to the properties. For Property P_1 our reduction algorithm eliminated all the registers other than $r0$. From Table 2 we can see that for the models with 2, 4, 8, 10, 12 registers of 28 bits, there are always 31 state variables, that is $r0[0]$, ..., $r0[27]$ and 3 state variables in the additional ASM representing P_1. However, SMV used all registers to verify P_1, and when the model was enlarged to have 12 registers of 28 bits, SMV could not complete the verification. FormalCheck also used more state variables when the number of registers in the model was increased. After verifying P_1 on the model with 2 registers of 28 bits, we opened the reduction manager window to see that $r0[0]$, ..., $r0[27]$, $r1[0]$, ..., $r1[3]$ were used, but $r1[0]$, ..., $r1[3]$ could have been eliminated. For Property P_2, our reduction algorithm used only 4 state variables ($r0[0]$, *flag*, *addsig*1, *addsig*2) and automatically reduced all the other bits of $r0$ and all the other registers no matter how many and how large the registers were. SMV used the least significant bit variables of all the registers to verify P_2, e.g., for the model with eight registers of 28 bits, $r0[0]$, ..., $r7[0]$ are used by checking the "cone" window in SMV. Table 2 shows that the number of state variables used in SMV is growing when more registers are added in the models. FormalCheck could not reduce anything when verifying P_2. We opened the reduction manager window

and could see that all the state variables were used. When the models became even larger, FormalCheck could not complete the verification of P_2.

9.1.2 Property Checking on the Abstract Model

For the circuit shown in Figure 25, we are only concerned about the data in a selected register being correctly updated and stored in the appropriate register. We can define the registers as words of size n using abstract sorts, e.g., an abstract sort *wordn*. This makes the description generic, and the verification is thus applicable to registers of any word size. For the data processing unit, here an incrementer, we can use an uninterpreted function symbol *finc*. The symbols *wordn* and *finc* are defined to be of abstract sort in the algebraic file of the MDG model checker:

```
abs_sort(wordn).
function(finc, [wordn],wordn).
```

The abstract model *M* in MDG-HDL is as follows:

```
% Variables definition
signal(s, bool).
signal(d, wordn).
signal(n_r0, wordn).
signal(r0, wordn).
signal(n_r1, wordn).
signal(r1, wordn).
signal(n_rs, bool).
signal(rs, bool).
signal(dnew, wordn).
signal(dout, wordn).
% Pairs of a state variable and its next state variable
st_nxst(r0, n_r0).
st_nxst(r1, n_r1).
st_nxst(rs, n_rs).
% Circuit definition
```

```
component(fork_s, fork(input(s), output(n_rs))).
component(mux1,mux(sel(s),inputs([(0,dnew),(1,r0)]),output(n_r0))).
component(mux2,mux(sel(s),inputs([(1,dnew),(0,r1)]),output(n_r1))).
component(mux3,mux(sel(rs),inputs([(0,r0),(1,r1)]),output(dout))).
component(r0,reg(input(n_r0),output(r0))).
component(r1,reg(input(n_r1),output(r1))).
component(rs,reg(input(n_rs),output(rs))).
component(finc,transform(inputs(dout),function(finc),output(dnew))).
outputs([]).
output_partition([]).
next_state_partition([[[n_r0]],[[n_r1]],[[rs]]]).
par_strategy(auto,auto).
```

The property to be verified on the abstract model specifies that if *s* is 0 and *rs* is 0, and the value of *r*0 is *v* in the current clock cycle, then *r*0 will be the value of *finc*(*v*) in the next clock cycle. The property expressed in L_{MDG} is as follows:

P_3: **AG** (**LET** (*v* = *r*0) **IN** ((*s* = 0 & *rs* = 0) -> (**X** (*r*0 = *finc*(*v*)))));

The circuit in Figure 28 represents the additional ASM for P_3. The verification of P_3 on the original model is transferred to the verification of P_3': **AG** (*flag* = 1) on the composite ASM consisting of *M* and M_{P3}.

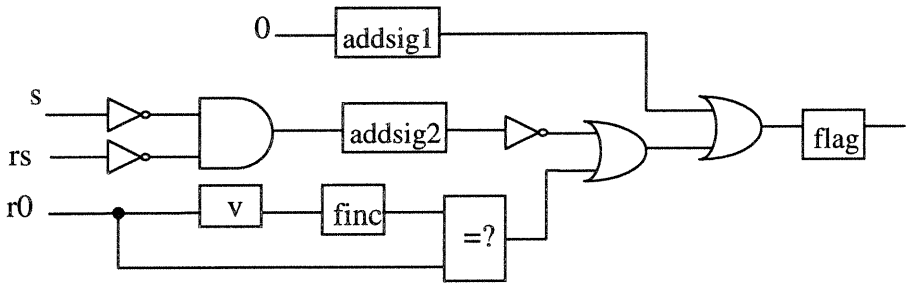


Figure 28. The additional ASM M_{P3} for P_3

The individual transition relations of the composite machine are represented by the following Directed Formulas that translate immediately to the MDG graph representation.

T_{flag}: $((addsig1 = 0) \wedge (addsig2 = 1) \wedge (abscomp(finc(v), r0) = 0) \wedge (flag' = 0)) \vee$
$(((addsig1 = 1) \vee (addsig2 = 0) \vee (abscomp(finc(v), r0) = 1)) \wedge (flag' = 1))$

$T_{addsig1}$: $(addsig1' = 0)$

$T_{addsig2}$: $(((rs = 1) \vee (s = 1)) \wedge (addsig2' = 0)) \vee ((rs = 0) \wedge (s = 0) \wedge (addsig2'=1))$

T_v: $(v' = r0)$

T_{rs}: $(rs' = s)$

T_{r0}: $((s = 1) \wedge (r0' = r0)) \vee ((s = 0) \wedge (rs = 0) \wedge (r0' = finc(r0))) \vee$
$((s = 0) \wedge (rs = 1) \wedge (r0' = finc(r1)))$

T_{r1}: $((s = 0) \wedge (r1' = r1)) \vee ((s = 1) \wedge (rs = 1) \wedge (r1' = finc(r1))) \vee$
$((s = 1) \wedge (rs = 0) \wedge (r1' = finc(r0)))$

The property is now verified in 0.7 second using 1.04M of memory and 201 MDG nodes. The state variables (rs, $r1$) are automatically converted to primary inputs, and only the state variables ($flag$, $addsig1$, $addsig2$, v, $r0$) are used. However, without this reduction, P_3 is successfully verified in 1.5 seconds using all state variables, 1.17M of memory, and 230 MDG nodes. Data abstraction makes the verification much faster. As it can be seen, by combining abstract data representation with efficient model reductions the state space explosion problem can be considerably diminished.

9.2 The Island Tunnel Controller

The Island Tunnel Controller (ITC) benchmark shown in Figure 29 was originally introduced by Fisler and Johnson [FJ95]. There is one lane tunnel connecting the mainland to an island. There are one traffic light and two sensors at both ends of the tunnel. On the island side, sensor *ie* detects the presence of vehicles at the tunnel entrance, and sensor *ix* detects the presence of vehicles at the tunnel exit.

Similarly, on the mainland side, sensor *me* is at the tunnel entrance and *mx* is at the tunnel exit. There is a constraint imposed on the maximum number of cars that may stay on the island. This introduces one counter *ic* on the island side and one counter *tc* in the tunnel to keep track of the number of cars currently on the island and in the tunnel respectively. We assume that all cars are finite in length, no car gets stuck in the tunnel, cars do not exit the tunnel before entering the tunnel, cars do not leave the tunnel entrance without traveling through the tunnel, and there is a sufficient distance between two cars such that the sensors can distinguish the cars. We will use MDG, SMV and FormalCheck to verify some properties on the ITC.

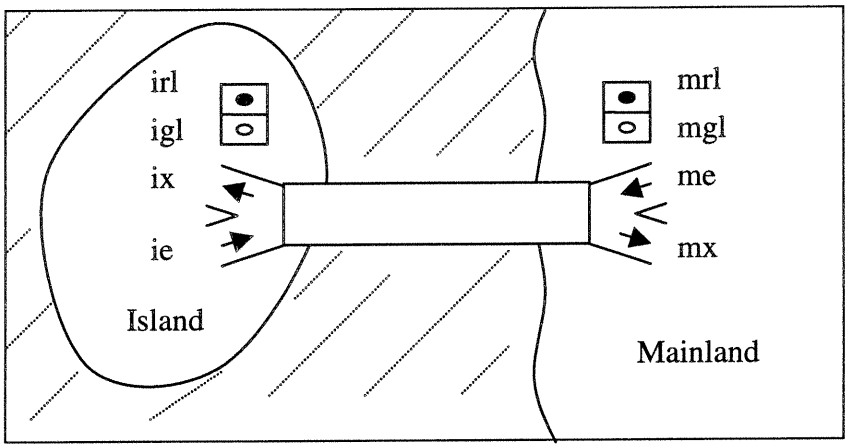


Figure 29. The island tunnel controller

9.2.1 ITC Specification

The specification of the ITC proposed by Fisler and Johnson [FJ95] uses three controllers and two counters shown in Figure 30. Their state transition diagrams are shown in Figure 31.

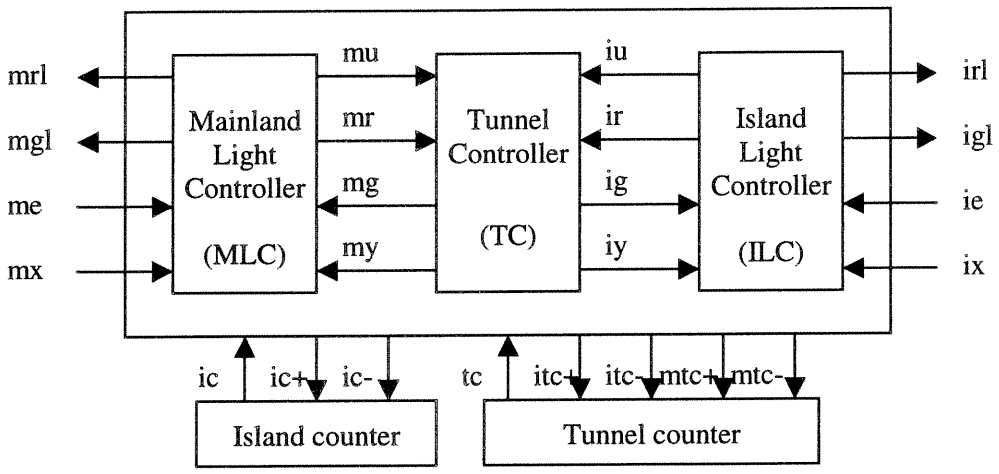
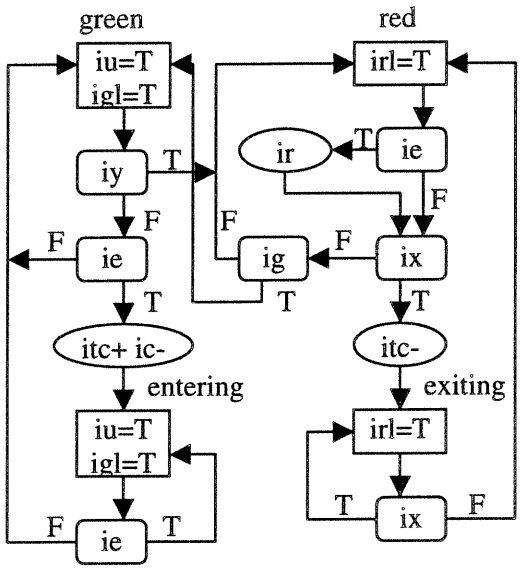
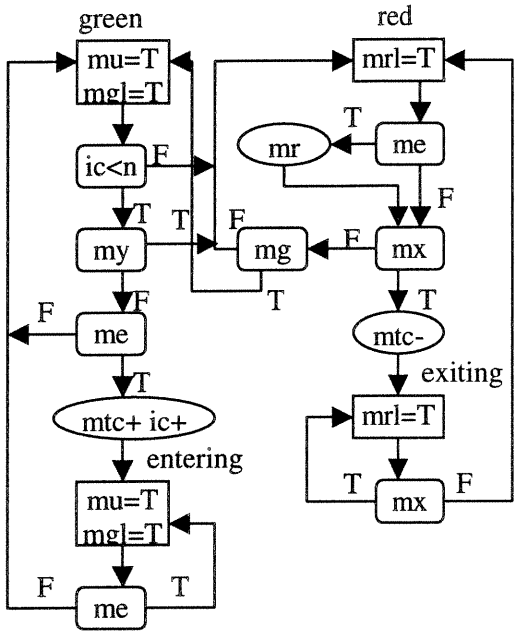


Figure 30. The specification of the Island Tunnel Controller

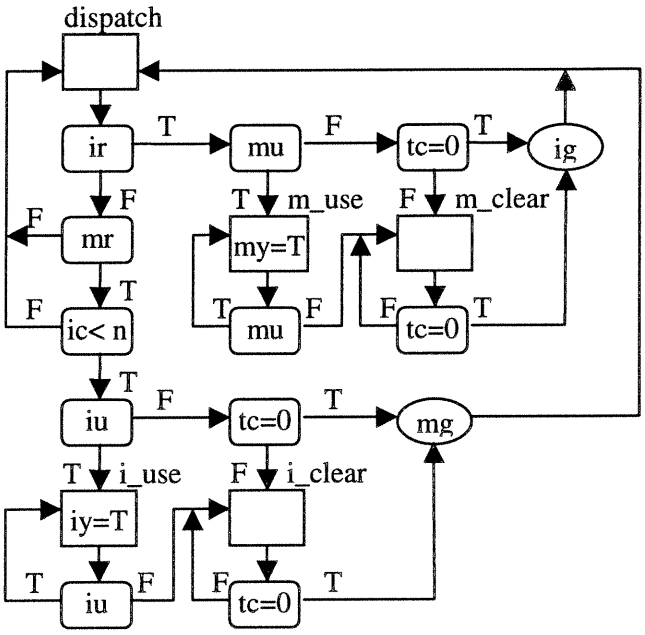
The island light controller (ILC) has four states: *green*, *entering*, *red* and *exiting*. The outputs *igl* and *irl* control the green and red lights on the island side, respectively; *iu* indicates that the cars from the island side are currently occupying the tunnel, and *ir* indicates that ILC is requesting the tunnel. The input *iy* requests the ILC to release control of the tunnel, and *ig* grants control of the tunnel from the island side. 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. For the tunnel counter, at each clock cycle, the counter *tc* is increased by 1 depending on *itc+* and *mtc+*, or decremented by 1 depending on *itc-* and *mtc-* unless it is already 0. The island counter operates in a similar way, except that the increment and decrement signals are *ic+* and *ic-*, respectively.



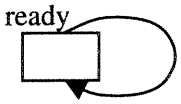
(a) Island light controller



(b) Mainland light controller

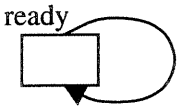


(c) Tunnel controller



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

(d) Island counter



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

(e) Tunnel counter

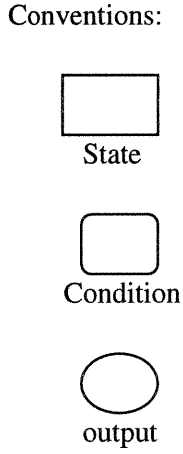


Figure 31. State transition diagrams of the Island Tunnel Controller

9.2.2 Property Checking on the ITC

We created the MDG-HDL models of the Island Tunnel Controller that include the modules representing ILC, MLC, TC and the counters. First, we defined all the signals as concrete variables. The three properties and their CTL formulas that we verified are as follows:

P_1: If the incremental signal of the island counter is valid and the island counter is 3 in the current clock cycle, then the island counter will be 4 in the next clock cycle.

$$\mathbf{AG}\ ((ic+ = 1)\ \&\ (ic = 3)) \rightarrow \mathbf{X}(ic = 4));$$

P_2: The tunnel counter is never ordered to increment simultaneously by ILC and MLC.

$$\mathbf{AG}\ (!\ ((itc+ = 1)\ \&\ (mtc+ = 1)));$$

P_3: The island counter is never ordered to increment and to decrement simultaneously.

$$\mathbf{AG}\ (!\ ((ic+ = 1)\ \&\ (ic- = 1)));$$

For example, with a 5-bit island counter, P_1 expressed in L_{MDG} is as follows:

```
AG (((ic_4=0 & ic_3=0 & ic_2=0 & ic_1=1 & ic_0=1) & (ic_plus=1))
-> X(ic_4=0 & ic_3=0 & ic_2=1 & ic_1=0 & ic_0=0));
```

Table 3 shows the experimental results of verifying P_1, P_2 and P_3 using the MDG model checker. We can see that without our reduction algorithm, MDG can only verify P_1, P_2 and P_3 on the ITC models having counters with less than 8 bits, while using our reduction algorithm MDG can verify the same properties on a model with 11-bit counters.

Table 3. Verifying P_1, P_2 and P_3 of ITC using MDG

Property	Counter Width (Bits)	No Reduction				Reduction			
		State Vars	Nodes	Time (Sec)	Mem (MB)	State Vars	Nodes	Time (Sec)	Mem (MB)
P_1	5	16	84849	240	29.5	11	8287	13.6	11.7
	6	18	284285	2369.5	93.2	12	16263	24.4	19.5
	7	20	953920	23662.9	306.3	13	34780	54	32.8
	8	-	-	-	-	14	71557	168.3	55
	9	-	-	-	-	15	152139	550	94.4
	10	-	-	-	-	16	328121	2868.7	168.4
	11	-	-	-	-	17	710457	14391.3	312.7
P_2	5	14	84486	230.1	29.2	9	7841	11.78	11.5
	6	16	283775	2601.8	92	10	15755	21.15	19.2
	7	18	930896	22507.7	304.8	11	33925	53.1	32.4
	8	-	-	-	-	12	70365	157	54.3
	9	-	-	-	-	13	150348	562.2	93.3
	10	-	-	-	-	14	325202	2717.6	166.3
	11	-	-	-	-	15	705391	14747.2	308.5
P_3	5	14	84486	234.4	29.6	9	7841	11	11.4
	6	16	283775	3010.4	91.7	10	15755	20.1	19.2
	7	18	953169	25318	301.6	11	33925	55.8	32.3
	8	-	-	-	-	12	70365	265.6	54
	9	-	-	-	-	13	150348	594.9	92.8
	10	-	-	-	-	14	325202	2945.1	165.3
	11	-	-	-	-	15	705391	14252.6	306.6

For example, to verify P_3 on the model with 5-bit counters, there are 84486 MDG nodes if model reduction is not used, while there are only 7841 nodes if reduction is used, which reduces the memory usage from 29.6MB to 11MB! From the columns labeled *Nodes*, *Time* and *Mem*, we can see that without reduction the

nodes, time and memory usage increase much faster than with reduction. When the two counters increase to 8 bits, the verification cannot finish. When reduction is used, for verifying P_1, P_2 and P_3 the tunnel counter is eliminated.

To illustrate the reductions obtained, we verify P_1 on the model with 3-bit counters. P_1 is transferred to an additional circuit composed with the original design. The property dependency graph of the composite machine is as follows:

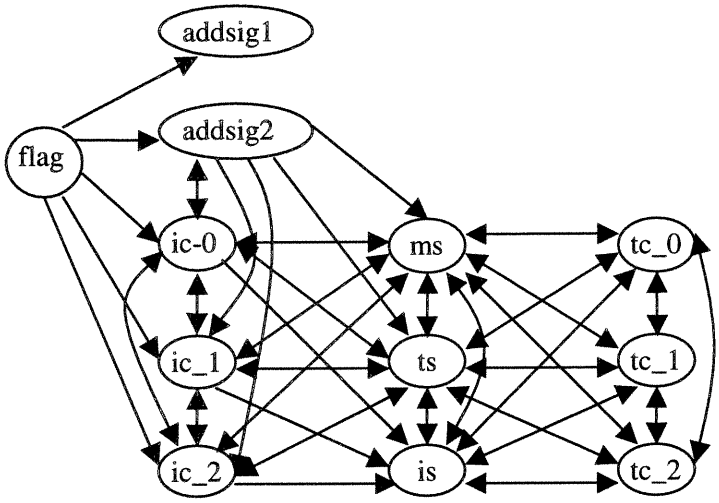


Figure 32. The property dependency graph

At the first iteration, (*flag*, *addsig*1, *addsig*2, *ic*_0, *ic*_1, *ic*_2) which appear in the additional ASM are used to construct the reduced model, and the property fails. Then, the direct determining state variables of the set are computed and the variables (*ms*, *ts*, *is*) are added, i.e., the set (*flag*, *addsig*1, *addsig*2, *ic*_0, *ic*_1, *ic*_2, *ms*, *ts*, *is*) is used to construct the reduced model. Property checking succeeds in this case at the second iteration. The tunnel counter *tc* was eliminated. The situation is similar when verifying P_2 and P_3. In the following, we will use FormalCheck and SMV to verify the same properties. The experimental results show that these tools do not eliminate *tc*.

We construct the Verilog and the SMV models of the ITC for FormalCheck and SMV respectively. P_1 expressed in FormalCheck is as follows:

```
After: main.clk==rising && main.rst==0 && main.ic_plus==1 && main.ic==3
Always: main.ic==4
Unless after: main.clk==rising
```

Property P_1 expressed in SMV is as follows:

```
SPEC AG (((ICplus = 1) & (IC = 3)) -> AX(IC = 4))
```

Properties P_2 and P_3 are expressed in a similar way as P_1 in FormalCheck and SMV. The experimental results of verifying P_1, P_2 and P_3 using FormalCheck, SMV and MDG are shown in Table 4. The reduction algorithm selected in FormalCheck is Iterated with the Empty reduction seed, and the run option is Symbolic (BDD). The run option of SMV uses heuristic variable ordering, computes the number of reachable states and restricts model checking to reachable states. In MDG our reduction algorithm (depth-first search) is selected.

From the columns labeled by *State Vars*, we can see that MDG uses less state variables than FormalCheck and SMV. To verify P_1, P_2 and P_3, MDG automatically reduces the tunnel counter, while SMV uses all the state variables. FormalCheck uses all the state variables for verifying P_1. For P_2 and P_3, FormalCheck uses *tc*[1] and *tc*[0], and reduces the higher order bits of the tunnel counter on models with counters less than 8 bits. (We can see from the "Reduction Manager" window that *tc*[1], *tc*[0] and other state variables are active, and the higher order bits of *tc* are started as inputs.) When the models have counters with 8 bits or more, FormalCheck uses all state variables.

Table 4. Verifying P_1, P_2, P_3 of ITC using FormalCheck, SMV and MDG

Property	Counter width	FormalCheck			SMV			MDG		
		State Vars	Time (Sec)	Mem (MB)	State Vars	Time (Sec)	Mem (MB)	State Vars	Time (Sec)	Mem (MB)
P_1	6	28	30	5.5	19	2.5	5.5	12	24.4	19.5
	7	30	33	5.7	21	6.7	6.5	13	54	32.8
	8	32	36	6.6	23	20.1	9.4	14	168.3	55
	9	34	50	8	25	211.4	76	15	550	94.4
	10	36	79	10.7	27	932.1	284	16	2868.7	168.4
	11	38	138	16.2	-	-	-	17	14391	312.7
P_2	6	24	37	5.4	19	2.4	5.6	10	21.15	19.2
	7	25	38	5.5	21	6.3	6.5	11	53.1	32.4
	8	32	65	6.6	23	53	23	12	157	54.3
	9	34	80	8	25	211.6	76	13	562.2	93.3
	10	36	112	10.7	27	916.7	284	14	2717.6	166.3
	11	38	167	16.3	-	-	-	15	14747	308.5
P_3	6	24	28	5.4	19	2.5	5.4	10	20.1	19.2
	7	25	30	5.5	21	6.2	6.3	11	55.8	32.3
	8	32	50	6.6	23	53.4	23	12	265.6	54
	9	34	62	1.5	25	212.2	76	13	594.9	92.8
	10	36	94	10.7	27	916.4	284	14	2945.1	165.3
	11	38	151	16.3	-	-	-	15	14252	306.6

When we manually reduce the tunnel counter by making all bits of *tc* as inputs in the "Reduction Manager", P_1, P_2 and P_3 are all verified by FormalCheck. We also manually reduce *tc* by setting *tc* free in the "Abstraction" window of SMV, P_1, P_2 and P_3 are successfully verified. Even though FormalCheck uses less time and memory than SMV and MDG in this example, since many different factors may affect the experimental results as mentioned before, it does not eliminate state variables as our algorithm in the MDG tool. From the table we can see that the memory usage of SMV is exponentially increasing with the increase in width of

the counter, while the memory usage of MDG is increased linearly by a factor about 1.7 with the increase of width. When the counters have 11 bits, SMV cannot automatically verify P_1, P_2 and P_3 in the available memory. But when we manually reduce *tc* in SMV, we can verify P_1 on the model with 11-bit counters in 165 seconds using 51MB memory, P_2 in 164.7 seconds using 48MB, and P_3 in 168.3 seconds using 48MB.

Summary

In this chapter we carried out property checking on two examples: a data processing circuit and the Island Tunnel Controller. We did a comparison between MDG, FormalCheck and SMV on these designs. From the results we can see that our reduction algorithms have enlarged the useful domain of the MDG tool, and make it practicable for large circuits. Also, these algorithms achieve better model reduction than the other tools in these two cases. The first example is a common circuit structure in telecommunication and data processing circuits. Since our methods can do efficient reduction on this example, it means that our methods can work on many real circuits having a similar structure. We can thus safely say that our methods provide better performance than other tools for a large class of circuits. Moreover, our methods are completely automatic without user-guided information, and do not use preimage computation that makes them useful for MDG. Our methods can also be used in other tools to improve their performance.

Chapter 10 Conclusions and Future Work

10.1 Conclusions

Although model checking can verify circuit designs automatically and produce state sequences as counterexamples when verification fails, the state explosion problem limits its use. In order to increase the efficiency of automatic formal verification, it is necessary to develop model reduction methods.

The MDG model checker is a formal verification tool developed in our university. Due to the state explosion problem, MDG could only be applied to small circuits in the past. Due to the occurrence of abstract state variables and uninterpreted function symbols in MDG, there is no preimage operation in MDG. Thus all the reduction algorithms based on preimage computation cannot be used in MDG. Our objective was to develop reduction algorithms that can operate under this restriction.

In this thesis, four reduction algorithms based on circuit topology or based on functional dependency were developed. All of the reduction algorithms do not use preimage operation and are particularly useful for the MDG model checker. Certainly, these methods can be also used in other tools to improve their performance. Experimental results have shown that our reduction algorithms are efficient. The original contributions of the thesis can be summarized as follows:

1. Two reduction algorithms based on topology of the circuits were developed. One such algorithm automatically searches the circuit and finds the *sufficient* part of the circuit that contains all signals and components connected to the

flags whose values are checked in the property. The *sufficient* part of circuit strongly preserves the property. The other one is an iterative reduction algorithm, which considers the influence of input signals of multiple fanin gates. It partitions the fanin signals of gates to sets of signals, each signal in one set has some common predecessor signals with another signal in the same set. Each time it constructs a reduced model by selecting one set and eliminating the other state variables by changing them into primary inputs. The reduced model weakly preserves the property. If the reduced model satisfies the property then verification terminates. If all the reduced models do not satisfy the property then the *sufficient* model is used. These two algorithms are fully automatic and the verification results are safe.

2. Property dependent state variables DV_P for a given property P were defined, and we proved that the reduced model constructed by using all the variables in DV_P is the least model regardless the initial states that strongly preserves property P stated in CTL*, and that the reduced model constructed by using a subset of DV_P that contains the state variables in P weakly preserves P stated in ACTL*. This is the theoretical basis of the reduction algorithms based on functional dependency. These theorems tell us where we can begin to reduce the model and when we can stop. When all the state variables in DV_P are used to construct the reduced model, the verification is final.

3. Two iterative reduction algorithms based on functional dependency were developed. Both of these algorithms start from the reduced model constructed by using all the state variables appearing in the property. If the verification fails on this reduced model, then a more detailed model that uses more state variables is constructed. At each iteration, more state variables from DV_P are used, until all DV_P is used. The critical thing is how to add a subset of DV_P at each iteration. We defined the *property dependency graph* (PDG) that reflects the functional dependency of the property, and *noncorrelated sets* that are functionally independent. Each time a noncorrelated set is selected and used to

construct the more detailed model. The two iterative reduction algorithms adopt two different search strategies: a Depth-first search and a Breadth-first search of the partitioned property dependency graph. These two algorithms are fully automatic and efficient.

4. The above reduction algorithms were implemented in the MDG model checker by using Quintus PROLOG. This makes the MDG model checker useful and practicable for real circuits. The automatic execution of reductions made this tool user friendly.

5. Experiments of property verification on a Data Processing Circuit and the Island Tunnel Controller benchmark were carried out using the MDG model checker, FormalCheck and SMV. The experimental results showed that our reduction algorithms can achieve better model reductions than the other tools on the classes of circuits represented by these benchmarks.

10.2 Future Work

The four reduction algorithms presented in this thesis largely alleviate the state explosion problem by reducing the original model to a smaller one. The implementation of these reduction algorithms in our MDG model checker has significantly improved the behavior of this tool and made it capable to verify large and complex circuits. In addition, other techniques can also be combined with our reduction algorithms to further improve the behavior of the MDG tool.

1. Develop heuristic algorithms to find a good variable order.

   In the MDG model checker, sets of states and transition relations are represented by MDG graphs. Like ROBDD, different node ordering may produce different sizes of MDG graphs. It is possible to lift some ROBDD node ordering techniques that have been successful at the Boolean level to MDGs

[CZJ92][BBF93][FFM93][FMK91][FOH93][Min96][PS95][RG97][Som96][THY93].

When using the MDG package, a custom symbol order must be given before verifying a circuit. In addition, MDGs have concrete variables, abstract variables and cross-terms in nodes that participate in the node ordering, and the ordering must satisfy a number of conditions for a well-formed MDG. Thus variable ordering is more difficult in MDGs than in ROBDDs. Since to find the optimum order is an NP-complete problem [THY93], we could study some heuristic algorithms to find good orders. We could start from four aspects. First, we can analyze the structure of MDGs, deduce the influence of the order of concrete variables, abstract variables and cross-terms on the topology of the MDGs as a guide. Second, we can analyze the topology of the circuits and try to get a good order. Third, we can analyze the functional dependency of the variables to get a good order. Fourth, since a fixed static custom symbol order may result in very large intermediate MDGs, we need to introduce dynamic reordering. The sifting and variable exchanging algorithms [Rud93][ISY91] could possibly be applied to MDG.

2. Develop algorithms to find good partition and ordering of transition relations.

In addition to a good variable order, finding a good partition of the transition relations of a sequential circuit or input-output relations into blocks and finding a good order of the blocks can also improve the efficiency of MDG computation. Burch, Clarke and Long [BCL91] proposed using partitioned transition relations. In this method, instead of using one ROBDD representation of the transition relation, the transition relations of different latches are kept as separate ROBDDs. Since ROBDDs representing the individual latch transition relations are much smaller than the combined one, this method can result in substantial memory savings. During image computation, the state variables that are not in the support of other transition relations can be quantified early, which also saves on memory usage. In [BCL91] and [GB94], the conjunction of the

transition relations is computed iteratively one by one. In [ZSCCL95], Zhou, Song and Cerny et al. developed a partitioned transition relation product algorithm in MDG. In this method, the relational product algorithm was extended to an n-ary operation and the partitioned transition relations were divided into blocks. This method also can be improved by finding a good way to partition the individual transition relations into blocks, and a good way to order the blocks.

3. Generate error trace

In addition to the above areas that improve the behavior of the MDG tool, there are other aspects of the MDG model checker that need to be improved. One advantage of model checking is that when the verification fails, an error trace from the initial state to the failure state can be generated. This helps designers to find bugs. Several ROBDD based model checkers possess this feature. Right now the MDG model checker does not. A counter example facility could be added in MDG to store the trace from a set of initial states to sets of states in which the property is not satisfied.

4. Use error traces to guide reduction.

Beside the reduction approaches presented in this thesis, there are some other approaches [GD00] [CGJLV00][WHLKZMD00] that use error traces as a guide to select the variables. When a property is violated on a reduced model, its error trace is used to find out what information is lost in the reduction process. Then hints can be given to guide the reduction process. The existing methods that use preimage computation cannot be applied to the MDG model checker. The error trace generator and new reduction algorithms of utilizing the error traces for MDG could be another research in the future.

5. Develop interface translating Verilog or VHDL to MDG-HDL

The MDG model checker accepts circuit descriptions in the MDG-HDL language, while some other tools (FormalCheck, VIS, SMV) accept circuit

descriptions in simplified Verilog or VHDL, the most popular hardware description languages. This makes those tools more easily applicable to real designs. Since there is no array type in MDG-HDL, it is very hard for a user to manually translate a circuit in Verilog/VHDL to MDG-HDL. An automatic translation system from Verilog or VHDL to MDG-HDL is needed.

6. Combine theorem proving with MDG model checking

The MDG model checking tool can be linked with the theorem prover HOL [PTCMS00]. Since theorem proving is built on higher order logic, hierarchical verification is possible where the module of design can be divided into several submodules. There are several hybrid methods that combine theorem proving with model checking [RSS95][JS93]. In the combined system, model checking is used to verify the submodules and pass the results to the theorem prover that completes the verification of the whole system. Theorem proving can verify large circuits, but it is not automatic, while model checking is automatic, but it cannot handle large design. Using the combined system, we can verify larger designs partially automatically. A combined system MDG-HOL is currently under study.

7. Solve the nontermination problem in MDG

Due to abstract sorts in MDG, reachability analysis may not terminate in circuits with a cyclic behavior. Some early research proposed two methods to solve the non-termination problem in some situations. One approach is to modify the circuit description file in such a way that the generic constant initial value is generalized by an abstract variable and the necessary rewriting rules are added to avoid non-termination problem [ZSTCCL96]. The other one [MSC97] provided an idea of generalizing the initial value by using a ρ-term [CH95] to finitely represent the infinite sets of states generated during reachability analysis. A more general method is expected that can automatically analyze the description files and infer the two generalization methods.

Bibliography

[ABCCLS96] K. D. Anon, N. Boulerice, E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, Y. Xu, Z. Zhou. MDG Tools for the Verification of RTL Designs. In *Proceedings of CAV '96*, pp. 372-382, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[ABHQR97] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, S. K. Rajamani. Partial-Order Reduction in Symbolic State Space Exploration. In *Proceedings of CAV'97*, pp. 340-351, Haifa, Israel, June 22-25, 1997.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[ASB94] A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Equivalences for fair Kripke structures. In *Proc. 21st Int. Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, 1994.

[Bai96] Christel Baier. Polynomial Time Algorithms for Testing Probabilistic Bisimulation and Simulation. In *Proceedings of CAV '96*, pp. 50-61, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[BBFS93] T. Besson, H. Bouzouzou, I. Floricica, G. Saucier. Input Order for ROBDDs Based on Kernal Analysis. *The European Conference on Design Automation with the European Event in (ASIC) Design 1993*, pp. 266-272, Paris, France, February 1993.

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property Preserving Simulations. In *Proceedings of the Forth Workshop on Computer Aided Verification (CAV'92),* pp. 260-273, Montreal, Canada, July 1992.

[BBR90] K. Brace, R. Bryant and R. Rudell. Efficient Implementation of a BDD Package. In *Proceedings 27th Design Automation Conference*, pp. 40-45, June 1990.

[BC94] R. E. Bryant, and Y. A. Chen, Verification of Arithmetic Functions with Binary Moment Diagrams. *Technical Report* CMU-CS-94-160, May 31, 1994.

[BCDM86] M. C. Browne, E. M. Clarke, D.L. Dill and B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. In *IEEE Transactions on Computer*s, pp. 1035-1044, December 1986.

[BCL91] J. R. Brurch, E. M. Clarke, D .E. Long. Symbolic Model Checking with Partitioned Transition Relations. In *Proceedings of VLSI 91*, pp. 49-58, Edinburgh, Scotland, 1991.

[BCL91-2] J. R. Brurch, E. M. Clarke, D .E. Long. Representing Circuits More Efficiently in Symbolic Model Checking. *28th ACM/IEEE Design Automation Conference*, pp. 403-407, San Francisco, USA, June 1991.

[BCLMD94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification. In *IEEE Transactions on Computer-Aided Desig*n, 13(4), pp. 401-424, April 1994.

[BCM90] C. Berthet, O. Coudert and J. C. Madre. New Ideas on Symbolic Manipulations of Finite State Machines. In *IEEE International Conference on Computer Design*, pp. 224-227, Sept.17-19, 1990.

[BCMDH92] J. R. Burch, E. M. Clarke, K. L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Information and Computation* pp. 142-170, Vol. 98, No. 2, June 1992.

[BCMD90] J. 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 Conferenc*e, pp. 46-51. IEEE Computer Society Press, Los Alamitos, CA, June 1990.

[BD94] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *6th International Conference on Computer Aided Verification.,*1994.

[Bell98] Bell Labs Design Automation. FormalCheck User's Guide. Lucent Technologies, May 1998.

[Ben01] Bob Bentley. Validating the Intel Pentium 4 Microprocessor. *Proceedings of the 38th Design Automation Conference*, pp. 244-248, Las Vegas Convention Center, Las Vegas, NV, June 18-22,2001.

[BF90] S. Bose and A. L. Fisher. Automatic Verification of Synchronous Circuits using Symbolic Simulation and Temporal Logic. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Desig*n, Leuven, Belgium, 1989, L.J.M. Claesen, (ed), pp. 759-764. North-Holland, Amsterdam, 1990.

[BFH91] A. Bouajjani, J. Fernandez, N. Halbwachs. Minimal Model Generation. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol 3, 1991.

[BGV99] R. E. Bryant, S. German, M. N. Velev. Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic. *Technical Report CMU-CS-99-115*, May, 1999.

[BHSSAC96] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, T. Villa. VIS: A System for Verification and Synthesis. In *Proceedings of CAV '96*, pp. 372-382, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[BLPV95] J. Bormann, J. Lohse, M. Payer, G. Venzl. Model Checking in Industrial Hardware Design. In *Proceedings of the 32th Design Automation Conference (DAC'95)*, pp. 298-303, Chiba, Japan, June 1995.

[BLS96] S. Bensalem, Y. Lakhnech, H. Saïdi. Powerful Techniques for the Automatic Generation of Invariants. In *Proceedings of CAV '96*, pp. 323-335, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[BM97] G. Barrett, A. McIsaac. Model Checking in a Microprocessor Design Project. In *Proceedings of CAV'97*, pp. 214-225, Haifa, Israel, June 22-25, 1997.

[BM98] R.S. Boyer and J.S.Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1998.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *proceedings of 27th Design Automation Conference*, pp. 40-45, June, 1990.

[Bry95] R. E. Bryant. Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. *International Conference on Computer-Aided Design ICCAD '95*, pp. 236-243, Las Vegas, U.S.A., November, 1995.

[Bry94] R. E. Bryant, and C.-J. H. Seger. Digital Circuit Verification using Partially-Ordered State Models. Invited paper, *International Symposium on Multi-Valued Logic*, pp. 2-7, May, 1994.

[Bry92] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, pp. 293-318, Vol. 24, No. 3, September, 1992.

[Bry91] R. E. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. *IEEE Transactions on Computers*, pp. 205-213, Vol. 40, No. 2, February, 1991.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, pp. 677-691, Vol. C-35, No. 8, August, 1986.

[Bry85] R.E.Bryant. Symbolic Manipulation of Booleann Functions Using a Graphical Representation. *22nd Design Automation Conference*, pp. 688-694, June, 1985.

[BS91] R. E. Bryant and C. J. H. Seger. Formal Verification of Digital Circuits using Symbolic Ternary System Models. In *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)*, E. M. Clarke and R. P. Kurshan (eds.). volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Spring-Verlag, pp. 121-146, New York, NY, 1991.

[BT] BlackTie web page. http://www.verplex.com/Release_bt_final.html

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of Synchronous Sequential Machines using Symbolic Execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, volume 407 of Lecture Notes in Computer Science. pp. 365-373. Spring-Verlag, New York, 1989.

[CCLQ97] G. Cabodi, P. Camurati, L. Lavagno, and S. Quer. Disjunctive Partitioning and Partial Iterative Squaring. In *Design Automation Conference*, 1997.

[CDFGNO96] C. Capellmann, R. Demant, F. Fatahi-Vanani, R. Galvez-Estrada, U. Nitsche, P. Ochsenschläger. Verification by Behaviour Abstraction - A Case Study of Service Interaction Detection in Intelligent Telephone Networks. In *Proceedings of CAV '96*, pp. 372-382, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[CDK90] E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A Unified Approach for Showing Language Containment and Equivalence Between Various Types of ω–Automata. In *Proceeding of the 15th Colloquium on Trees in Algebra and Programming*, volume 407 of Lecture Notes in Computer Science. Springer-Verlag, 1990.

[CDM00] U. S. Costa, D. Deharbe, A. M. Moreira. Variable Ordering of BDD with Parallel Genetic Algorithms. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA 2000.

[Cer80] Eduard Cerny. Characteristic function in multivalued logic systems. *Digital Processes*, Vol. 6, pp. 167-174, 1980.

[Cer77] Eduard Cerny. Unique and Identity Solutions of Boolean Equations. *Digital Processes*, Vol. 6, pp. 331-337, 1977.

[CM77] Eduard Cerny, M. A. Marin. An Approach to Unified Methodology of Combinational Switching Circuits. *IEEE Trans. Computers*, C-26(8), pp. 745-756, Aug., 1977.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proceedings of the Workshop on Logic of Programs*. Volume 131 of Lecture Notes in Computer Science,1981.

[CE86] E. M. Clarke and E. A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logic of Programs*: Workshop, Yorktown Heights, NY, May 1981, vol. 131 of LNCS, Springer-Verlag, 1981.

[CES86] 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*, 8(2): pp. 244-263, April 1986.

[CFJ93] E. M. Clarke, T. Filkorn, S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In *CAV 1993*, pp. 450-462, Elounda, Greece, June, 1993.

[CFJ96] E. M. Clarke, T. Filkorn, S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In *Formal Methods in System Design*, Vol 9, 1996.

[CGJLV00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. E.A.Emerson and A.P.Sistla (Eds.), *CAV 2000*, LNCS 1855, pp. 154-169, Chicago, USA, July 2000.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, January 1992.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Model Checking and Abstraction. In *ACM-TOPLAS*, Vol 16, No.5, September 1994.

[CGL94-2] E. Clarke, O. Grumberg, and D. Long. Verification tools for Finite-State Concurrent Systems. In *Lecture Notes in Computer Science* 803, 1994.

[CGL96] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking. In *Springer-Verilag Nato ASI Series F*, Volume 152, 1996.

[CGM87] A. J. Camilleri, M. J. C. Gordon, and T. F. Melham. Hardware Verification Using Higher-order Logic. In *From HDL Descriptions to Guaranteed Correct Circuit Design*s, pp. 43-67. D. Borrione(ed.). North-Holland, Amserdam, 1987.

[CGMZ95] E. M. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient Generation of Counter Examples and Witness in Symbolic Model Checking. In *DAC 95*, pp. 427-432, San Francisco, U.S.A.

[CH95] Hong Chen, and Jieh Hsiang. Recurrence Domains: Their Unification and Application to Logic Programming. In *Information and Computation 122*, pp. 45-69, 1995.

[CLCZS95] F. Corella, M. Langevin, E. Cerny, Z. Zhou, 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)*, Frankfurt, Germany, October 1995.

[CM90] O.Coudert and J. C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proceedings of International Conference on Computer-Aided Design (ICCAD'90)*, pp. 126-129, Santa Clara, U.S.A, November1990.

[CMB90] O. Coudert, J. C. Madre, and C. Berthet. Verifying Temporal Properties of Sequential Machines without Building Their Date Diagrams. In *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)*, E.M. Clarke and R.P.Kurshan (eds.). volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, Spring-Verlag, pp. 23-32, New York, NY, 1991.

[CMCG96] E. M. Clarke, K. L. McMillan, S. Campos, V. Hartonas-Garmhausen. Symbolic Model Checking. In *Proceedings of CAV '96*, pp372-382, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[Cur94] Paul Curzon. The formal verification of the Fairisle ATM switching element. *Technical Report No.328, No.329*, University of Cambridge Computer Laboratory, 1994.

[CYB97] Y. A. Chen, B. Yang, and R. E. Bryant. Breadth-First with Depth-First BDD Construction: A Hybrid Approach. *Technical Report* CMU-CS-97-120, March,1997.

[CYF94] B. Chen, M. Yamazaki, M. Fujita. Bug Identification of a Real Chip Design by Symbolic Model Checking. In *Proceedings of International Symposium on Circuits and Systems (ISCAS'94),* 1994.

[CZJYT92] Ney Calazans, Q. Zhang, R. Jacobi, B. Yernaux and A. M. Trullemans. Advanced Ordering and Manipulation Techniques for Binary Decision Diagrams. In *Proceedings of the 29th Design Automation Conference (DAC'29),* June 1992.

[CZSLC97] F. Corella, Z.Zhou, X. Song, M. Langevin, E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. *Formal Methods in System Design.* 10(1): pp. 7-46, February 1997.

[Dam96] Dennis. R. Dams. Abstract Interpretation and Partition Refinement for Model Checking. *Ph.D thesis*, Eindhoven University of Technology, 1996.

[DDG98] Rolf Drechsler, Nicole Drechsler, W. Günther. Fast Exact Minimization of BDDs. In *Proceedings of the 34th Design Automation Conference (DAC'98),* pp. 200-205. San Francisco, California. June 1998.

[DG98] Rolf Drechsler, Wolfgang Günther. Linear Transformations and Exact Minimization of BDDs. In *Great Lakes Symposium on VLSI (GLSV'98*), pp. 325-330. Lafayette, 1998.

[DGG93] Dennis Dams, Orna Grumberg, Rob Gerth. Generation of Reduced models for Checking Fragments of CTL. In *Proceedings of CAV93*, pp. 479-490, Elouda, Greece, June 1993.

[Dil96] David L. Dill. The Mur*phi* Verification System. In *Proceedings of CAV '96*, pp. 372-382, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[Emer90] E. Allen Emerson. Temporal and Modal Logic, 16th chapter of *HANDBOOK OF THEORETICAL COMPUTER SCIENC*E, edited by J.van Leeuwen. Elsevier Science Publishers B.V., 1990.

[EN96] E. A. Emerson, K. S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems (Extended Abstract). In *Proceedings of CAV '96*, pp87-98, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[ES93] E.A. Emerson, A. P. Sistla. Symmmetry and Model Checking. In *Proceedings of CAV93*, pp. 450-462, Elounda, Greece, June 1993.

[FC] FormalCheck web page.
http://www.bell-labs.com/org/blda/product_formal.html

[FC97] FormalCheck User's Guide. Bell labs Design Automation,
Lucent Technologies, V1.1, 1997.

[Fer90] J. Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 1990.

[FFM93] M. Fujita, H. Fujisawa, Y. Matsunaga. Variable Ordering Algorithms for Ordered Binary Decision Diagrams and their Evaluation. In *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 6-12, January 1993.

[FJ95] K. Fisler and S. Johnson. Integrating design and Verification Environments Through A Logic Supporting Hardware Diagrams. In *Proceedings of IFIP Conference on Hardware Description Languages and their Applications (CHDL'95).* August 1995, Chiba, Japan.

[FMK91] M. Fujita, Y. Matsunaga, and T. Kakuda. On Variable Ordering of Binary Decision Diagrams for the Application of Multi-level Synthesis. In *Proceedings European Design Automation Conference*, pp. 50-54, Amsterdam, February 1991

[FOH93] Hiroshige Fujii, Goichi Ootomo, and Chikahiro 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, U.S.A, June 1993.

[FV98] K. Fisler, Moshe Y. Vardi. Bisimulation Minimization in an Automata-Theoretic Verification Framework. In *Proceedings of FMCAD'98*. Palo Alto, CA, USA, Nov. 1998.

[GB94] D. Geist, I. Beer. Efficient Model Checking by Automated Ordering of Transition Relation Partitions. In *Proceedings of Computer Aided Verification*, D.L.Dill Ed. LNCS 818, Springer-Verlag, pp. 299-310, June 1994.

[GD00] S. Govindaraju and D. Dill. Counterexample-guided Choice of Projections in Approximate Symbolic Model Checking. In *Proceedings of ICCAD 2000*, pp. 115-119, San Jose, CA, Nov. 2000.

[GH86] M. J. C. Gordon and J. Herbert. Formal hardware verification methodology and its application to a network interface chip. *IEE Proceedings,* 133, Part E(5), pp. 255-270, September 1986.

[GL91] O. Grumberg, and David E.Long. Model Checking and Modular Verification. J.c.m.Baeton, J.F.Groote (Eds), Proceedings of CONCUR'91 2nd International Conference on Concurrency Theory. Vol 527 of LNCS, pp. 250-265. Springer-Verlag, Aug. 1991.

[GL94] O. Grumberg, and D. E. Long. Model Checking and Modular Verification. In *ACM Transactions on Programming Languiages and Systems*, pp. 843-871, Vol 16, No.3, May 1994.

[GM93] M. J. C. Gordon, T. F. Melham. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. *Cambridge University Press*, Cambridge, UK,1993.

[Gor93] M. J. C. Gordon, et al. The HOL System Description, Cambridge Research Center, SRI International, Suite 23, Miller's Yard, Cambridge CB2 1RQ, England, 1993.

[Gri96] E. P. Gribomont. Atomicity Refinement and Trace Reduction Theorems. In *Proceedings of CAV '96*, pp. 311-322, New Brunswick, NJ, USA, July 31-August 3, 1996.

[GS97] V. Gyuris, A. P. Sistla. On-the-Fly Model Checking Under Fairness That Exploits Symmetry. In *Proceedings of CAV'97*, pp. 232-243, Haifa, Israel, June 22-25, 1997.

[GS97-2] S. Graf, H. Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV'97*, pp. 72-83, Haifa, Israel, June 22-25, 1997.

[Gup92] A. Gupta. Formal Hardware Verification Methods: A Survey. In Journal *Formal Methods in System Desig*n, vol 1, pp. 151-238 (1992).

[HB95] R. Hojati, R. K. Brayton. Automatic datapath Abstraction In Hardware Systems. In *Proceedings of Conference on Hardware Description Language (CHDL'95),* Tokyo, Japan, August 1995.

[HC00] Jin Hou and Eduard Cerny. Model Reductions Based on MDG Model Checking. In *Proceedings of 13th Annual International ASIC/SOC Conference*. Washington D.C., USA, Sept. 13-16, 2000.

[HC00-2] Jin Hou and Eduard Cerny. Model Reductions and a Case Study. In *Proceeding of FMCAD 2000*, pp. 299-315, Austin, TX, USA, Nov. 1-3, 2000. Lecture Notes in Computer Science 1954.

[HD93] A. J. Hu and David L. Dill. Reducing BDD Size by Exploiting Functional Dependencies. *30th ACM/IEEE Design Automation Conference*, pp. 266-271, Dallas, June 1993.

[HDB97] R. Hojati, D. L. Dill, R. K. Brayton. Verifying linear temporal properties of data insensitive controllers using finite instantiations. In *Proceedings of IFIP Conference on Hardware Description Languages and their Applications (CHDL'97).* Spain, April 1997.

[HGD95] H. Hungar, O. Grumberg, and W. Damm. What if Model Checking Must Be Truly Symbolic. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95*), Aarhus, Denmark, May 1995.

[HIKB96] R. Hojati, A. Isles, D. Kirkpatrick, R. K. Brayton. Verification Using Uninterpreted Functions and Finite Instantiations. In *Proceding of Formal Methods in Computer-Aided Design* (FMCAD), November 1996.

[HM96] H. Higuchi, Y. Matsunaga. A Fast State Reduction Algorithm for Incompletely Specified Finite State Machines. In *33rd Design Automation Conference*, pp. 436-466, Las Vegas, June 1996.

[Hunt89] W. A. Hunt, Jr. Microprocessor Design Verification. *Journal of Automated Reasoning,* 5(4), pp429-460 (1989).

[ID96] C. Norris Ip, David Dill. Better Verification Through Symmetry. In *Formal Methods in System Designs*, Vol 9, Auguest 1996.

[ID96-2] C. Norris Ip, David Dill. State Reduction Using Reversible Rules. In *33rd Design Automation Conference*, pp. 564-567, Las Vegas, June 1996.

[ID96-3] C. Norris Ip, David L. Dill. Verifying Systems with Replicated Components in Mur*phi*. In *Proceedings of CAV '96*, pp. 147-158, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[Ip96] C. N. Ip. State Reduction Methods for Automatic Formal Verification. *Ph.D Thesis*, Stanford University, U.S.A. 1996.

[ISY91] Nagisa Ishiura, Hiroshi Sawada and Shuzo Yajima. Minimization of Binary Decision Diagrams Based on Exchanges of Variables. In *Proceedings International Conference on Computer-Aided Design*, pp. 472-475, Santa Clara, U.S.A, November 1991.

[JQKP97] Jae-Y. Jang, Shaz Qadeer, Matt Kaufmann, Carl Pixley. Formal Verification of FIRE: A Case Study. In *Proceedings of the 34th Design Automation Conference (DAC'97)*. Anaheim, CA, June 1997.

[JS93] J. Joyce and C. Seger. Linking BDD-based Symbolic Evaluation to Interactive Theorem Proving. In *Proceedings of the 30th Design Automation Conference*, Association for Computing Machinery, pp. 469-474, Dallas, U.S.A, 1993.

[Kai93] Roope Kaivola. Compositional Model Checking for Linear-Time Temporal Logic. In *LNCS*, pp. 248-259, Vol. 663, 1993.

[Kla97] N. Klarlund. An *n* log *n* Algorithm for Online BDD Refinement. In *Proceedings of CAV'97*, pp. 107-118, Haifa, Israel, June 22-25, 1997.

[KPV97] I. Kokkarinen, D. Peled, A. Valmari: Relaxed Visibility Enhances Partial Order Reduction. In *Proceedings of CAV'97*, pp. 328-339, Haifa, Israel, June 22-25, 1997.

[KS83] P. Kanellakis and S. Smolka. Ccs Expressions, Finite State Processes and Three Problems of Equivalence. In *Proceedings ACM Symp*. On Principles of Distributed Computing, 1983.

[Kur87] R. P. Kurshan. Reducibility in Analysis of Coordination. In *Lecture Notes in Control and Information Sciences*. Vol 103, pp. 19-39, Springer-Verlag, 1987.

[Kur90] R. P. Kurshan. Analysis of Discrete Event Coordination. In *Lecture Notes in Compute Science*. Vol 430, 1990.

[Kur92] R. P. Kurshan. Automata-Theoretic Verification of Coordinating Processes. *UC Berkeley notes*, 1992.

[Kur97] R. P. Kurshan. Formal Verification in a Commercial Setting. In *Proceedings of the 34th Design Automation Conference (DAC'97)*. Anaheim, California. July 1997.

[KV96] O. Kupferman, M. Y. Vardi. Module Checking. In *Proceedings of CAV '96*, pp. 75-86, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[KV96] O. Kupferman, M. Y. Vardi. Verification of Fair Transisiton Systems. In *Proceedings of CAV '96*, pp. 372-382, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[KV97] O. Kupferman and M. Y. Vardi. Model Checking Revisited. In *Proceedings of CAV'97*, pp. 36-47, Haifa, Israel, June 22-25, 1997.

[Lon93] D. E.Long. Model Checking, Abstraction, and Compositional Verification. *Ph. D Thesis,* Carnegie Mellon Univerisity, 1993.

[LPJHS96] W. Lee, A. Pardo, J. Y. Jang, G. Hachtel, F. Somenzi. Tearing Based Automatic Abstraction for CTL Model Checking. *ICCAD96*, pp. 76-81, San Jose CA USA, Nov.10-14, 1996.

[LS90] S. S. Lam, A. U. Shankar. Refinement and Projection of Relational Specifications. *Lecture Notes in Computer Science*, pp. 414-453, Vol 430, 1990.

[LY92] David Lee, Mihalis Yannakakis. Online Minimization of Transition Systems. *24th Annual ACM STOC* 1992.

[Mar97] F. E. Marshner. Practical Challenges for Industrial Formal Verification Tools. In *Proceedings of CAV'97*, Haifa, Israel, June 22-25, 1997.

[MB99] C. B. McDonald, and R. E. Bryant. Symbolic Functional and Timing Verification of Transistor-Level Circuits. *International Conference on Computer-Aided Design (ICCAD '99)*, November, 1999.

[McF93] 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.

[McM92] K. L. McMillan. Symbolic Model Checking, An Approach to the State Explosion Problem.. *Ph.D. Thesis*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.

[McM96] K. L. McMillan. A Conjunctively Decomposed Boolean Representation for Symbolic Model Checking. In *Proceedings of CAV '96*, pp. 13-25, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[McM97] K. L. McMillan. A Compositional Rule for Hardware Design Refinement. In *Proceedings of CAV'97*, pp. 24-35, Haifa, Israel, June 22-25, 1997.

[McM98] K. L. McMillan. SMV documentation. http://www-cad. eecs.berkeley.edu/~kenmcmill

[Mil80] R. Milner. A Calculus of Communicating Systems, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin 1980.

[Min96] Shin-ichi Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, 1996.

[MK96] H. Miller, S. Katz. Saving Space by Fully Exploiting Invisible Transitions. In *Proceedings of CAV '96*, pp. 336-347, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[MS91] K. L. McMillan and J. Schwalbe. Formal Verification of the Encore Gigamax Cache Consistency Protocol. In *Proceedings of the International Symposium on Shared* Memory Multiprocessing, pp. 242-251, 1991.

[MSC97] O. Ait-Mohamed, X. Song, E. Cerny. On the non-termination of MDGs-based Abstract State Enumeration. *IFIP TC10 WG10.5 International Conference on Correct Hardware and Verification Methods*, October 1997.

[MWBS88] Sharad Malik, A.Wang, R.Brayton, and A.Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis

Environment. In *Proceedings International Conference on Computer-aided Design*, November 1988.

[NB99] M. N. Velev, and R. E. Bryant. Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions. *Correct Hardware Design and Verification Methods CHARME '99*, pp. 37-53, L. Pierre, and T. Kropf, eds., LNCS 1703, Springer-Verlag, September, 1999.

[NIJBV97] A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned-ROBDDs. 1997.

[NK00] K. S. Namjoshi and R. P. Kurshan. Syntactic Program Transformations for Automatic Abstraction. E. A. Emerson and A. P. Sistla (Eds.), *CAV 2000, LNCS 1855*, pp. 139-153, Chicago, USA, July 2000.

[NT00] K. S. Namjoshi and R. J. Trefler. On the Completeness of Compositional Reasoning. E.A.Emerson and A.P.Sistla (Eds.), *CAV 2000, LNCS 1855*, pp. 139-153, Chicago, USA, July 2000.

[OSC97] 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.

[PB99] M. Pandey, and R. E. Bryant. Exploiting Symmetry When Verifying Transistor-level Circuits by Symbolic Trajectory Evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 918-935, Vol. 18, No. 7, July, 1999.

[PB97] M. Pandey, R. E. Bryant. Exploiting Symmetry When Verifying Transitor-Level Circuits by Symbolic Trajectory Evaluation. In *Proceedings of CAV'97*, pp. 244-255, Haifa, Israel, June 22-25, 1997.

[PH97] A. Pardo and G. D. Hachtel. Automatic Abstraction Techniques for Propositional μ-calculus Model Checking. In *Proceedings of CAV'97*, pp. 12-23, Haifa, Israel, June 22-25, 1997.

[PH98] A. Pardo and G. D. Hachtel. Incremental CTL Model Checking using BDD Subsetting. In *35th Design Automation Conference*, pp. 457-462, San Francisco, CA USA, June 1998.

[Pnue86] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In *Current Trends in Concurrency*, J.W. de Bakker, W.-P.de Roever, and G. Rozenberg (eds.), volume 224 of *Lecture Notes in Computer Science*, pp. 510-584. Springer-Verlag, New York, 1986.

[PS95] S. Panda, F. Somenzi. Who are the Variables in Your Neighbourhood? In *Proceedings of International Conference on Computer-Aided Design (ICCAD'95)*, 1995.

[PS96] M. Pistore, D. Sangiorgi. A Partition Refinement Algorithm for the *pi*-Calculus (Extended Abstract). In *Proceedings of CAV '96*, pp. 38-49, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[PT87] R. Paige, R. E. Tarjan. Three Partition Refinement Algorithms. *SIAM J. Comput.*, Vol. 16, No.6, December 1987.

[PTCMS00] V. K. Pisini, S. Tahar, P. Curzon, O. Ait-Mohamed and X. Song: Formal Hardware Verification by Integrating HOL and MDG; *Proc. ACM 10th Great Lakes Symposium on VLSI (GLS-VLSI'00)*, pp. 23-28, Chicago, Illinois, USA, March 2000, ACM Publications.

[QS81] J. P. Quielle and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th International Symposium in Programming*, 1981.

[RG97] Rajeev K. Ranjan, W. Gosti . Speeding up Variable Reordering of OBDDs. In *Proceedings of International Conference on Computer Design (ICCD'97)*, Austin,TX, USA 1997.

[RMSS98] K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi. Approximation and Decomposition of Binary Decision Diagrams. In *35th Design Automation Conference*. San Francisco, CA USA, June 1998.

[Row97] M. Rowe. Formal Verification - Applications & Case Studies. In *Proceedings of CAV'97*, pp. 11-12, Haifa, Israel, June 22-25, 1997.

[RSS95] S. Rajan, N. Shankar and M. Srivas. An Integration of Model checking with Automated Proof Checking. In *Computer Aided Verification*. P. Wolper (Ed.) LNCS pp. 84-97, Vol.939, 1995.

[Rud93] Richard Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. *IWLS 93 Workshop Notes.*

[Rus96] John M. Rushby. Automated Deduction and Formal Methods. In *Proceedings of CAV '96*, pp. 169-183, New Brunswick, NJ, USA, July 31 - August 3, 1996.

[SB93] C. Seger and R. E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectorie. *UBC Department of Computer Science Technical Report 93-8*, April 1993.

[SB95] M. Starkey, and R. E. Bryant. Using Ordered Binary-Decision Diagrams for Compressing Images and Image Sequences. *Technical Report* CMU-CS-95-105, January, 1995. [SC85] A. P. Sistla and E.M. Clarke. *Complexity of propositional linear temporal logic. Journal of the ACM,* 32(3): pp. 733-749 July 1985.

[SD97] U. Stern, D. L. Dill. Parallelizing the Mur*phi* Verifier. In *Proceedings of CAV'97*, pp. 256-278, Haifa, Israel, June 22-25, 1997.

[Seg98] C. Seger. Combining Theorem Proving and Model Checking: How Much Theorem Proving Is Needed? Invited talk. In *Proceedings of Conference on Computer-Aided Verification (CAV'98)*. Vancouver, BC, Canada, July 1998.

[Sis97] A. P. Sistla. Parametrized Verification of Linear Networks Using Automata as Invariants. In *Proceedings of CAV'97*, pp. 412-423, Haifa, Israel, June 22-25, 1997.

[SLH98] M. Sela, F. Limor, I. Haifa. Input Elimination and Abstraction in Model Checking. *FMCAD98*, Palo Alto, CA, USA, Nov. 1998.

[SMV] CMU - School of Computer Science Formal Methods - Model Checking web page. http://www.cs.cmu.edu/~modelcheck/

[SS99] H. Saidi, N. Shankar. Abstract and Model Checking While You Prove. N. Halbwachs, D.Peled (Eds). *CAV'99*, pp. 443-454, Trento, Italy, July 1999.

[SUM96] H. Sipma, T. E. Uribe, Z. Manna. Deductive Model Checking. In *Proceedings of CAV '96*, pp. 208-219, New Brunswick, NJ, USA, July 31-August 3, 1996.

[Tar55] A. Tarski. A Lattice-theoretical Fixpoint Theorem and Its Applications. *Pacific J. Math*, pp. 285-309, vol 5, 1955.

[TB93] G. Thuau, B. Berkane. A Unified Framework for Describing and Verifying Hardware Synchronous Sequential Systems. In *Formal Methods in System Design*, pp. 259-276, vol 2, 1993.

[TBK95] H. J. Touati, R. K. Brayton, R. Kurshan. Testing Language Containment for w-Automata Using BDDs. In *Information and Computation* 118, pp. 101-109, Vol. 118, No.1, 1995.

[TB97] S. Tasiran, R. K. Brayton. STARI: A Case Study in Compositional and Hierarchical Timing Verification. In *Proceedings of CAV'97*, pp. 191-201, Haifa, Israel, June 22-25, 1997.

[THY93] 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, University of Tokyo,1993.

[TSLBV90] 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, U.S.A., 1990.

[TY96] S. Tripakis, S. Yovine. Analysis of Timed Systems Based on Time-Abstracting Bisimulation. In *Proceedings of CAV '96*, pp. 232-243, New Brunswick, NJ, USA, July 31-August 3, 1996.

[VB98] M. N. Velev, and R. E. Bryant. Bit-level Abstraction in the Verification of Pipelined Microprocessors by Correspondence Checking. In *Formal Methods in Computer-Aided Design FMCAD '98*, G. Gopalakrishnan and P. Windley, eds., LNCS 1522, Springer-Verlag, pp. 18-35, November, 1998.

[VIS] The VIS Group web page. VIS: Verification Interacting with Synthesis, 1995. http://www-cad.eecs.berkeley.edu/Respep/Research/vis.

[WHLKZMD00] Dong Wang, Pei-Hsin Ho, Jiang Long, James Kukula, Yunshan Zhu, Tony Ma, Robert Damiano. Formal Property Verification by Abstraction Refinement with Formal, Simulation and Hybrid Engines. In *Proceedings of*

the 38th Design Automation Conference (DAC38). pp. 35-40, Las Vegas, NV, June 18-22, 2001.

[XCSCLP99] Y.Xu, E.Cerny, A.Silburt, A.Coady, Y.Liu,P.Pownall. Pratical Application of Formal Verification Techniques on a Frame MuxDemux Chip from Nortel Semiconductors. *Charme 1999,* Bad Herrenalb, Germany, pp. 110-124, September 1999.

[XCSCM98] Y. Xu, E. Cerny, X. Song, F. Corella, 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, BC, Canada, July 1998.

[XCSH97] Y. Xu, E. Cerny, A. Silburt, and R. B. Hughes. Property Verification Using Theorem Proving and Model Checking. In *Integrated System Desig*n, November 1997.

[Xu99] Y. Xu. Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs. *Ph.D Thesis*, University of Montreal, 1999.

[YBHBC98] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi, A Performance Study of BDD-Based Model Checking, *Formal Methods in Computer-Aided Design FMCAD '98*, G. Gopalakrishnan and P. Windley, eds., LNCS 1522, Springer-Verlag, pp. 255-289, November, 1998.

[YCBH98] B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. O'Hallaron. Space- and Time-Efficient BDD Construction by Working Set Control. *Asian-Pacific Design Automation Conference ASPDAC '98*, pp. 423-432, Feb., 1998.

[YSB99] B. Yang, R. Simmons, R. E. Bryant, and D. R. O'Hallaron. Optimizing Symbolic Model Checking for Constraint-Rich Models. In *Proceedings of Computer-Aided Verification CAV '99*, 1999. N. Halbwachs, and D. Peled eds., LNCS 1633, Springer-Verlag, pp. 328-340, July, 1999.

[ZB96] Z. Zhou and N. Boulerice. MDG Tools (V1.0) User's Manual. D'IRO, University of Montreal, June 1996.

[Zhou97] Z. Zhou. Multiway Decision Graphs and Their Applications in Automatic Formal Verification of RTL Designs. *PhD Thesis*, D'IRO, University of Montreal, 1997.

[ZSCCL94] Z. Zhou, X. Song, F. Corella, E. Cerny, M. Langevin. Description and Verification of RTL Design Using Multiway Decision Graphs. *Technical Report RC19822*, IBM T. J. Watson Research Center, November 1994.

[ZSCC95] Z. Zhou, X. Song, F. Corella, E. Cerny, M. Langevin. Partitioning Transition Relations Efficiently and Automatically. In *IEEE Proceedings of fifth Great Lakes Symposium on VLSI*, March 1995.

[ZSCCL95] 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.

[ZSTCCL96] 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, CA, USA, 1996.

Appendix 1 MDG-HDL Code for the Circuit in Figure 25

A Concrete model having 2 registers of 28 bits :

```
:- multifile signal/2.
:- multifile component/2.
:- multifile outputs/1.
:- multifile st_nxst/2.
:- multifile init_val/2.
:- multifile init_var/2.
:- multifile par_strategy/2.
:- multifile next_state_partition/1.
:- multifile output_partition/1.
:- dynamic signal/2.
:- dynamic component/2.
:- dynamic outputs/1.
:- dynamic st_nxst/2.
:- dynamic init_val/2.
:- dynamic init_var/2.
:- dynamic next_state_partition/1.

% input var
signal(s,bool).
% connection vars
signal(a0,bool).
signal(a1,bool).
signal(a2,bool).
signal(a3,bool).
signal(a4,bool).
signal(a5,bool).
signal(a6,bool).
signal(a7,bool).
signal(a8,bool).
signal(a9,bool).
signal(a10,bool).
signal(a11,bool).
signal(a12,bool).
signal(a13,bool).
signal(a14,bool).
signal(a15,bool).
signal(a16,bool).
signal(a17,bool).
signal(a18,bool).
signal(a19,bool).
signal(a20,bool).
signal(a21,bool).
```

```
signal(a22,bool).
signal(a23,bool).
signal(a24,bool).
signal(a25,bool).
signal(a26,bool).
signal(a27,bool).

signal(c0,bool).
signal(c1,bool).
signal(c2,bool).
signal(c3,bool).
signal(c4,bool).
signal(c5,bool).
signal(c6,bool).
signal(c7,bool).
signal(c8,bool).
signal(c9,bool).
signal(c10,bool).
signal(c11,bool).
signal(c12,bool).
signal(c13,bool).
signal(c14,bool).
signal(c15,bool).
signal(c16,bool).
signal(c17,bool).
signal(c18,bool).
signal(c19,bool).
signal(c20,bool).
signal(c21,bool).
signal(c22,bool).
signal(c23,bool).
signal(c24,bool).
signal(c25,bool).
signal(c26,bool).
signal(c27,bool).
signal(c28,bool).

signal(sum0,bool).
signal(sum1,bool).
signal(sum2,bool).
signal(sum3,bool).
signal(sum4,bool).
signal(sum5,bool).
signal(sum6,bool).
signal(sum7,bool).
signal(sum8,bool).
signal(sum9,bool).
signal(sum10,bool).
signal(sum11,bool).
signal(sum12,bool).
signal(sum13,bool).
signal(sum14,bool).
signal(sum15,bool).
signal(sum16,bool).
signal(sum17,bool).
signal(sum18,bool).
```

```
signal(sum19,bool).
signal(sum20,bool).
signal(sum21,bool).
signal(sum22,bool).
signal(sum23,bool).
signal(sum24,bool).
signal(sum25,bool).
signal(sum26,bool).
signal(sum27,bool).


% state variables in the machine
signal(n_r,bool).
signal(r,bool).

signal(r0_0,bool).
signal(r0_1,bool).
signal(r0_2,bool).
signal(r0_3,bool).
signal(r0_4,bool).
signal(r0_5,bool).
signal(r0_6,bool).
signal(r0_7,bool).
signal(r0_8,bool).
signal(r0_9,bool).
signal(r0_10,bool).
signal(r0_11,bool).
signal(r0_12,bool).
signal(r0_13,bool).
signal(r0_14,bool).
signal(r0_15,bool).
signal(r0_16,bool).
signal(r0_17,bool).
signal(r0_18,bool).
signal(r0_19,bool).
signal(r0_20,bool).
signal(r0_21,bool).
signal(r0_22,bool).
signal(r0_23,bool).
signal(r0_24,bool).
signal(r0_25,bool).
signal(r0_26,bool).
signal(r0_27,bool).

signal(n_r0_0,bool).
signal(n_r0_1,bool).
signal(n_r0_2,bool).
signal(n_r0_3,bool).
signal(n_r0_4,bool).
signal(n_r0_5,bool).
signal(n_r0_6,bool).
signal(n_r0_7,bool).
signal(n_r0_8,bool).
signal(n_r0_9,bool).
signal(n_r0_10,bool).
signal(n_r0_11,bool).
```

```
signal(n_r0_12,bool).
signal(n_r0_13,bool).
signal(n_r0_14,bool).
signal(n_r0_15,bool).
signal(n_r0_16,bool).
signal(n_r0_17,bool).
signal(n_r0_18,bool).
signal(n_r0_19,bool).
signal(n_r0_20,bool).
signal(n_r0_21,bool).
signal(n_r0_22,bool).
signal(n_r0_23,bool).
signal(n_r0_24,bool).
signal(n_r0_25,bool).
signal(n_r0_26,bool).
signal(n_r0_27,bool).

signal(r1_0,bool).
signal(r1_1,bool).
signal(r1_2,bool).
signal(r1_3,bool).
signal(r1_4,bool).
signal(r1_5,bool).
signal(r1_6,bool).
signal(r1_7,bool).
signal(r1_8,bool).
signal(r1_9,bool).
signal(r1_10,bool).
signal(r1_11,bool).
signal(r1_12,bool).
signal(r1_13,bool).
signal(r1_14,bool).
signal(r1_15,bool).
signal(r1_16,bool).
signal(r1_17,bool).
signal(r1_18,bool).
signal(r1_19,bool).
signal(r1_20,bool).
signal(r1_21,bool).
signal(r1_22,bool).
signal(r1_23,bool).
signal(r1_24,bool).
signal(r1_25,bool).
signal(r1_26,bool).
signal(r1_27,bool).

signal(n_r1_0,bool).
signal(n_r1_1,bool).
signal(n_r1_2,bool).
signal(n_r1_3,bool).
signal(n_r1_4,bool).
signal(n_r1_5,bool).
signal(n_r1_6,bool).
signal(n_r1_7,bool).
signal(n_r1_8,bool).
signal(n_r1_9,bool).
```

```
signal(n_r1_10,bool).
signal(n_r1_11,bool).
signal(n_r1_12,bool).
signal(n_r1_13,bool).
signal(n_r1_14,bool).
signal(n_r1_15,bool).
signal(n_r1_16,bool).
signal(n_r1_17,bool).
signal(n_r1_18,bool).
signal(n_r1_19,bool).
signal(n_r1_20,bool).
signal(n_r1_21,bool).
signal(n_r1_22,bool).
signal(n_r1_23,bool).
signal(n_r1_24,bool).
signal(n_r1_25,bool).
signal(n_r1_26,bool).
signal(n_r1_27,bool).

% next state
st_nxst(r,n_r).
st_nxst(r0_0,n_r0_0).
st_nxst(r0_1,n_r0_1).
st_nxst(r0_2,n_r0_2).
st_nxst(r0_3,n_r0_3).
st_nxst(r0_4,n_r0_4).
st_nxst(r0_5,n_r0_5).
st_nxst(r0_6,n_r0_6).
st_nxst(r0_7,n_r0_7).
st_nxst(r0_8,n_r0_8).
st_nxst(r0_9,n_r0_9).
st_nxst(r0_10,n_r0_10).
st_nxst(r0_11,n_r0_11).
st_nxst(r0_12,n_r0_12).
st_nxst(r0_13,n_r0_13).
st_nxst(r0_14,n_r0_14).
st_nxst(r0_15,n_r0_15).
st_nxst(r0_16,n_r0_16).
st_nxst(r0_17,n_r0_17).
st_nxst(r0_18,n_r0_18).
st_nxst(r0_19,n_r0_19).
st_nxst(r0_20,n_r0_20).
st_nxst(r0_21,n_r0_21).
st_nxst(r0_22,n_r0_22).
st_nxst(r0_23,n_r0_23).
st_nxst(r0_24,n_r0_24).
st_nxst(r0_25,n_r0_25).
st_nxst(r0_26,n_r0_26).
st_nxst(r0_27,n_r0_27).

st_nxst(r1_0,n_r1_0).
st_nxst(r1_1,n_r1_1).
st_nxst(r1_2,n_r1_2).
st_nxst(r1_3,n_r1_3).
st_nxst(r1_4,n_r1_4).
st_nxst(r1_5,n_r1_5).
```

```
st_nxst(r1_6,n_r1_6).
st_nxst(r1_7,n_r1_7).
st_nxst(r1_8,n_r1_8).
st_nxst(r1_9,n_r1_9).
st_nxst(r1_10,n_r1_10).
st_nxst(r1_11,n_r1_11).
st_nxst(r1_12,n_r1_12).
st_nxst(r1_13,n_r1_13).
st_nxst(r1_14,n_r1_14).
st_nxst(r1_15,n_r1_15).
st_nxst(r1_16,n_r1_16).
st_nxst(r1_17,n_r1_17).
st_nxst(r1_18,n_r1_18).
st_nxst(r1_19,n_r1_19).
st_nxst(r1_20,n_r1_20).
st_nxst(r1_21,n_r1_21).
st_nxst(r1_22,n_r1_22).
st_nxst(r1_23,n_r1_23).
st_nxst(r1_24,n_r1_24).
st_nxst(r1_25,n_r1_25).
st_nxst(r1_26,n_r1_26).
st_nxst(r1_27,n_r1_27).


% transition relation
component(fork_s,fork(input(s),output(n_r))).
component(comp_r,reg(input(n_r),output(r))).
component(const1,constant_signal(value(1),signal(c0))).
% For bit0
component(mux0_0,table([[s,n_r0_0],[0,sum0]|r0_0])).
component(mux1_0,table([[s,n_r1_0],[1,sum0]|r1_0])).

component(mux_r_0,table([[r,a0],[0,r0_0],[1,r1_0]])).

component(comp_sum0,xor(input(c0,a0),output(sum0))).
component(comp_carrier0,and(input(c0,a0),output(c1))).

% For bit1
component(mux0_1,table([[s,n_r0_1],[0,sum1]|r0_1])).
component(mux1_1,table([[s,n_r1_1],[1,sum1]|r1_1])).

component(mux_r_1,table([[r,a1],[0,r0_1],[1,r1_1]])).

component(comp_sum1,xor(input(c1,a1),output(sum1))).
component(comp_carrier1,and(input(c1,a1),output(c2))).

% For bit2
component(mux0_2,table([[s,n_r0_2],[0,sum2]|r0_2])).
component(mux1_2,table([[s,n_r1_2],[1,sum2]|r1_2])).

component(mux_r_2,table([[r,a2],[0,r0_2],[1,r1_2]])).

component(comp_sum2,xor(input(c2,a2),output(sum2))).
component(comp_carrier2,and(input(c2,a2),output(c3))).

% For bit3
```

```
component(mux0_3,table([[s,n_r0_3],[0,sum3]|r0_3])).
component(mux1_3,table([[s,n_r1_3],[1,sum3]|r1_3])).

component(mux_r_3,table([[r,a3],[0,r0_3],[1,r1_3]])).

component(comp_sum3,xor(input(c3,a3),output(sum3))).
component(comp_carrier3,and(input(c3,a3),output(c4))).

% For bit4
component(mux0_4,table([[s,n_r0_4],[0,sum4]|r0_4])).
component(mux1_4,table([[s,n_r1_4],[1,sum4]|r1_4])).

component(mux_r_4,table([[r,a4],[0,r0_4],[1,r1_4]])).

component(comp_sum4,xor(input(c4,a4),output(sum4))).
component(comp_carrier4,and(input(c4,a4),output(c5))).

% For bit5
component(mux0_5,table([[s,n_r0_5],[0,sum5]|r0_5])).
component(mux1_5,table([[s,n_r1_5],[1,sum5]|r1_5])).

component(mux_r_5,table([[r,a5],[0,r0_5],[1,r1_5]])).

component(comp_sum5,xor(input(c5,a5),output(sum5))).
component(comp_carrier5,and(input(c5,a5),output(c6))).

% For bit6
component(mux0_6,table([[s,n_r0_6],[0,sum6]|r0_6])).
component(mux1_6,table([[s,n_r1_6],[1,sum6]|r1_6])).

component(mux_r_6,table([[r,a6],[0,r0_6],[1,r1_6]])).

component(comp_sum6,xor(input(c6,a6),output(sum6))).
component(comp_carrier6,and(input(c6,a6),output(c7))).

% For bit7
component(mux0_7,table([[s,n_r0_7],[0,sum7]|r0_7])).
component(mux1_7,table([[s,n_r1_7],[1,sum7]|r1_7])).

component(mux_r_7,table([[r,a7],[0,r0_7],[1,r1_7]])).

component(comp_sum7,xor(input(c7,a7),output(sum7))).
component(comp_carrier7,and(input(c7,a7),output(c8))).

% For bit8
component(mux0_8,table([[s,n_r0_8],[0,sum8]|r0_8])).
component(mux1_8,table([[s,n_r1_8],[1,sum8]|r1_8])).

component(mux_r_8,table([[r,a8],[0,r0_8],[1,r1_8]])).

component(comp_sum8,xor(input(c8,a8),output(sum8))).
component(comp_carrier8,and(input(c8,a8),output(c9))).

% For bit9
component(mux0_9,table([[s,n_r0_9],[0,sum9]|r0_9])).
component(mux1_9,table([[s,n_r1_9],[1,sum9]|r1_9])).
```

```
component(mux_r_9,table([[r,a9],[0,r0_9],[1,r1_9]])).

component(comp_sum9,xor(input(c9,a9),output(sum9))).
component(comp_carrier9,and(input(c9,a9),output(c10))).

% For bit10
component(mux0_10,table([[s,n_r0_10],[0,sum10]|r0_10])).
component(mux1_10,table([[s,n_r1_10],[1,sum10]|r1_10])).

component(mux_r_10,table([[r,a10],[0,r0_10],[1,r1_10]])).

component(comp_sum10,xor(input(c10,a10),output(sum10))).
component(comp_carrier10,and(input(c10,a10),output(c11))).

% For bit11
component(mux0_11,table([[s,n_r0_11],[0,sum11]|r0_11])).
component(mux1_11,table([[s,n_r1_11],[1,sum11]|r1_11])).

component(mux_r_11,table([[r,a11],[0,r0_11],[1,r1_11]])).

component(comp_sum11,xor(input(c11,a11),output(sum11))).
component(comp_carrier11,and(input(c11,a11),output(c12))).

% For bit12
component(mux0_12,table([[s,n_r0_12],[0,sum12]|r0_12])).
component(mux1_12,table([[s,n_r1_12],[1,sum12]|r1_12])).

component(mux_r_12,table([[r,a12],[0,r0_0],[1,r1_0]])).

component(comp_sum12,xor(input(c12,a12),output(sum12))).
component(comp_carrier12,and(input(c12,a12),output(c13))).

% For bit13
component(mux0_13,table([[s,n_r0_13],[0,sum13]|r0_13])).
component(mux1_13,table([[s,n_r1_13],[1,sum13]|r1_13])).

component(mux_r_13,table([[r,a13],[0,r0_13],[1,r1_13]])).

component(comp_sum13,xor(input(c13,a13),output(sum13))).
component(comp_carrier13,and(input(c13,a13),output(c14))).

% For bit14
component(mux0_14,table([[s,n_r0_14],[0,sum14]|r0_14])).
component(mux1_14,table([[s,n_r1_14],[1,sum14]|r1_14])).

component(mux_r_14,table([[r,a14],[0,r0_14],[1,r1_14]])).

component(comp_sum14,xor(input(c14,a14),output(sum14))).
component(comp_carrier14,and(input(c14,a14),output(c15))).

% For bit15
component(mux0_15,table([[s,n_r0_15],[0,sum15]|r0_15])).
component(mux1_15,table([[s,n_r1_15],[1,sum15]|r1_15])).

component(mux_r_15,table([[r,a15],[0,r0_15],[1,r1_15]])).
```

```
component(comp_sum15,xor(input(c15,a15),output(sum15))).
component(comp_carrier15,and(input(c15,a15),output(c16))).

% For bit16
component(mux0_16,table([[s,n_r0_16],[0,sum16]|r0_16])).
component(mux1_16,table([[s,n_r1_16],[1,sum16]|r1_16])).

component(mux_r_16,table([[r,a16],[0,r0_16],[1,r1_16]])).

component(comp_sum16,xor(input(c16,a16),output(sum16))).
component(comp_carrier16,and(input(c16,a16),output(c17))).

% For bit17
component(mux0_17,table([[s,n_r0_17],[0,sum17]|r0_17])).
component(mux1_17,table([[s,n_r1_17],[1,sum17]|r1_17])).

component(mux_r_17,table([[r,a17],[0,r0_17],[1,r1_17]])).

component(comp_sum17,xor(input(c17,a17),output(sum17))).
component(comp_carrier17,and(input(c17,a17),output(c18))).

% For bit18
component(mux0_18,table([[s,n_r0_18],[0,sum18]|r0_18])).
component(mux1_18,table([[s,n_r1_18],[1,sum18]|r1_18])).

component(mux_r_18,table([[r,a18],[0,r0_18],[1,r1_18]])).

component(comp_sum18,xor(input(c18,a18),output(sum18))).
component(comp_carrier18,and(input(c18,a18),output(c19))).

% For bit19
component(mux0_19,table([[s,n_r0_19],[0,sum19]|r0_19])).
component(mux1_19,table([[s,n_r1_19],[1,sum19]|r1_19])).

component(mux_r_19,table([[r,a19],[0,r0_19],[1,r1_19]])).

component(comp_sum19,xor(input(c19,a19),output(sum19))).
component(comp_carrier19,and(input(c19,a19),output(c20))).

% For bit20
component(mux0_20,table([[s,n_r0_20],[0,sum20]|r0_20])).
component(mux1_20,table([[s,n_r1_20],[1,sum20]|r1_20])).

component(mux_r_20,table([[r,a20],[0,r0_20],[1,r1_20]])).

component(comp_sum20,xor(input(c20,a20),output(sum20))).
component(comp_carrier20,and(input(c20,a20),output(c21))).

% For bit21
component(mux0_21,table([[s,n_r0_21],[0,sum21]|r0_21])).
component(mux1_21,table([[s,n_r1_21],[1,sum21]|r1_21])).

component(mux_r_21,table([[r,a21],[0,r0_21],[1,r1_21]])).

component(comp_sum21,xor(input(c21,a21),output(sum21))).
```

```
component(comp_carrier21,and(input(c21,a21),output(c22))).

% For bit22
component(mux0_22,table([[s,n_r0_22],[0,sum22]|r0_22])).
component(mux1_22,table([[s,n_r1_22],[1,sum22]|r1_22])).

component(mux_r_22,table([[r,a22],[0,r0_22],[1,r1_22]])).

component(comp_sum22,xor(input(c22,a22),output(sum22))).
component(comp_carrier22,and(input(c22,a22),output(c23))).

% For bit23
component(mux0_23,table([[s,n_r0_23],[0,sum23]|r0_23])).
component(mux1_23,table([[s,n_r1_23],[1,sum23]|r1_23])).

component(mux_r_23,table([[r,a23],[0,r0_23],[1,r1_23]])).

component(comp_sum23,xor(input(c23,a23),output(sum23))).
component(comp_carrier23,and(input(c23,a23),output(c24))).

% For bit24
component(mux0_24,table([[s,n_r0_24],[0,sum24]|r0_24])).
component(mux1_24,table([[s,n_r1_24],[1,sum24]|r1_24])).

component(mux_r_24,table([[r,a24],[0,r0_24],[1,r1_24]])).

component(comp_sum24,xor(input(c24,a24),output(sum24))).
component(comp_carrier24,and(input(c24,a24),output(c25))).

% For bit25
component(mux0_25,table([[s,n_r0_25],[0,sum25]|r0_25])).
component(mux1_25,table([[s,n_r1_25],[1,sum25]|r1_25])).

component(mux_r_25,table([[r,a25],[0,r0_25],[1,r1_25]])).

component(comp_sum25,xor(input(c25,a25),output(sum25))).
component(comp_carrier25,and(input(c25,a25),output(c26))).

% For bit26
component(mux0_26,table([[s,n_r0_26],[0,sum26]|r0_26])).
component(mux1_26,table([[s,n_r1_26],[1,sum26]|r1_26])).

component(mux_r_26,table([[r,a26],[0,r0_26],[1,r1_26]])).

component(comp_sum26,xor(input(c26,a26),output(sum26))).
component(comp_carrier26,and(input(c26,a26),output(c27))).

% For bit27
component(mux0_27,table([[s,n_r0_27],[0,sum27]|r0_27])).
component(mux1_27,table([[s,n_r1_27],[1,sum27]|r1_27])).

component(mux_r_27,table([[r,a27],[0,r0_27],[1,r1_27]])).

component(comp_sum27,xor(input(c27,a27),output(sum27))).
component(comp_carrier27,and(input(c27,a27),output(c28))).
```

```
outputs([flag]).

output_partition([]).

next_state_partition([
[[n_addedSignal2]],
[[n_addedSignal62]],
[[n_flag]],
[[n_r]],
[[n_r0_0]],
[[n_r0_1]],
[[n_r0_2]],
[[n_r0_3]],
[[n_r0_4]],
[[n_r0_5]],
[[n_r0_6]],
[[n_r0_7]],
[[n_r0_8]],
[[n_r0_9]],
[[n_r0_10]],
[[n_r0_11]],
[[n_r0_12]],
[[n_r0_13]],
[[n_r0_14]],
[[n_r0_15]],
[[n_r0_16]],
[[n_r0_17]],
[[n_r0_18]],
[[n_r0_19]],
[[n_r0_20]],
[[n_r0_21]],
[[n_r0_22]],
[[n_r0_23]],
[[n_r0_24]],
[[n_r0_25]],
[[n_r0_26]],
[[n_r0_27]],
[[n_r1_0]],
[[n_r1_1]],
[[n_r1_2]],
[[n_r1_3]],
[[n_r1_4]],
[[n_r1_5]],
[[n_r1_6]],
[[n_r1_7]],
[[n_r1_8]],
[[n_r1_9]],
[[n_r1_10]],
[[n_r1_11]],
[[n_r1_12]],
[[n_r1_13]],
[[n_r1_14]],
[[n_r1_15]],
[[n_r1_16]],
[[n_r1_17]],
[[n_r1_18]],
```

```
[[n_r1_19]],
[[n_r1_20]],
[[n_r1_21]],
[[n_r1_22]],
[[n_r1_23]],
[[n_r1_24]],
[[n_r1_25]],
[[n_r1_26]],
[[n_r1_27]]
]).

par_strategy(auto,auto).
```

Appendix 2 MDG-HDL Code for ITC

An ITC concrete model with 8 bit counters:

```
% Multifile declaration required by Prolog system.

:- multifile component/2.
:- multifile signal/2.
:- multifile next_state_partition/1.
:- multifile output_partition/1.
:- multifile init_val/2.
:- multifile init_var/2.
:- multifile st_nxst/2.
:- multifile outputs/1.
:- multifile par_strategy/2.
:- dynamic component/2.
:- dynamic signal/2.
:- dynamic next_state_partition/1.
:- dynamic output_partition/1.
:- dynamic st_nxst/2.
:- dynamic init_val/2.
:- dynamic init_var/2.
:- dynamic outputs/1.

%================= Inputs and Outputs ==================

%--- Input signals---

signal(ie,bool).
signal(ix,bool).

signal(me,bool).
signal(mx,bool).

%--- Outputs ---

signal(irl_A,bool).
signal(igl_A,bool).
signal(itc_plus_A,bool).
signal(itc_min_A,bool).
signal(ic_min_A,bool).

signal(mrl_A,bool).
signal(mgl_A,bool).
signal(mtc_plus_A,bool).
signal(mtc_min_A,bool).
signal(ic_plus_A,bool).
```

```
%=============== Island Light Controller ================

%--- Input signals---

signal(ig_A,bool).
signal(iy_A,bool).

%--- Outputs ---

signal(ir_A,bool).
signal(iu_A,bool).

%--- State variables---

signal(is_A,mi_sort).

%--- Behavioral description for the island light controller----


component(is_comp_A,table([[is_A,ig_A,iy_A,ie,ix,n_is_A],
                      [green,*,0,0,*,green],
                      [green,*,0,1,*,entering],
                      [green,*,1,*,*,red],
                      [entering,*,*,0,*,green],
                      [entering,*,*,1,*,entering],
                      [red,  0,*,*,0,red],
                      [red,  1,*,*,0,green],
                      [red,  *,*,*,1,exiting],
                      [exiting, *,*,*,0,red],
                      [exiting, *,*,*,1,exiting]])).

component(ir_comp_A,table([[is_A,ie,ir_A],
                      [red,1,1]|0])).

component(irl_comp_A,table([[is_A,irl_A],
                      [red,1],
                      [exiting,1]|0])).

component(igl_comp_A,table([[is_A,igl_A],
                      [green,1],
                      [entering,1]|0])).

component(iu_comp_A,table([[is_A,iu_A],
                      [green,1],
                      [entering,1]|0])).


component(itc_plus_comp_A,table([[is_A,iy_A,ie,itc_plus_A],
                      [green,0,1,1]|0])).

component(itc_minus_comp_A,table([[is_A,ix,itc_min_A],
                      [red,1,1]|0])).

component(ic_min_comp_A,table([[is_A,iy_A,ie,ic_min_A],
                      [green,0,1,1]|0])).
```

```
%=============== Mainland Light Controller ==================

%--- Input signals---

signal(mg_A,bool).
signal(my_A,bool).

%--- Outputs ---

signal(mr_A,bool).
signal(mu_A,bool).

%--- State variables---

signal(ms_A,mi_sort).

%--- Behavioral description for the mainland light controller---

component(ms_comp_A,table(
                    [[ms_A,mg_A,my_A,me,mx,lessn_ic_A,n_ms_A],
                     [green,*,*,*,*,0,red],
                     [green,*,0,0,*,1,green],
                     [green,*,0,1,*,1,entering],
                     [green,*,1,*,*,1,red],
                     [entering,*,*,0,*,*,green],
                     [entering,*,*,1,*,*,entering],
                     [red,  0,*,*,0,*,red],
                     [red,  1,*,*,0,*,green],
                     [red,  *,*,*,1,*,exiting],
                     [exiting, *,*,*,0,*,red],
                     [exiting, *,*,*,1,*,exiting]])).

component(mr_comp_A,table([[ms_A,me,mr_A],
                           [red,1,1]|0])).

component(mrl_comp_A,table([[ms_A,mrl_A],
                            [red,1],
                            [exiting,1]|0])).

component(mgl_comp_A,table([[ms_A,mgl_A],
                            [green,1],
                            [entering,1]|0])).

component(mu_comp_A,table ([[ms_A,mu_A],
                            [green,1],
                            [entering,1]|0])).


component(mtc_plus_comp_A,table(
                           [[ms_A,my_A,me,lessn_ic_A,mtc_plus_A],
                            [green,0,1,1,1]|0])).

component(mtc_minus_comp_A,table(
                           [[ms_A,mx,mtc_min_A],
                            [red,1,1]|0])).
```

```
component(ic_plus_comp_A,table(
                            [[ms_A,my_A,me,lessn_ic_A,ic_plus_A],
                             [green,0,1,1,1]|0])).

%================== Tunnel Controller ====================

%--- State variables---

signal(ts_A,ts_sort).

%---- Behavioral description for the tunnel controller----

component(ts_comp_A,table(
         [[ts_A,ir_A,mr_A,lessn_ic_A,equz_tc_A,iu_A,mu_A,n_ts_A],
          [dispatch,    0,0,*,*,*,*,      dispatch],
          [dispatch,    0,1,0,*,*,*,      dispatch],
          [dispatch,    0,1,1,*,1,*,      iuse],
          [dispatch,    0,1,1,0,0,*,      iclear],
          [dispatch,    0,1,1,1,0,*,      dispatch],
          [dispatch,    1,*,*,0,*,0,      mclear],
          [dispatch,    1,*,*,1,*,0,      dispatch],
          [dispatch,    1,*,*,*,*,1,      muse],
          [iuse,        *,*,*,*,0,*,      iclear],
          [iuse,        *,*,*,*,1,*,      iuse],
          [muse,        *,*,*,*,*,0,      mclear],
          [muse,        *,*,*,*,*,1,      muse],
          [iclear,      *,*,*,0,*,*,      iclear],
          [iclear,      *,*,*,1,*,*,      dispatch],
          [mclear,      *,*,*,0,*,*,      mclear],
          [mclear,      *,*,*,1,*,*,      dispatch]])).

component(ig_comp_A,table([[ts_A,ir_A,equz_tc_A,mu_A,ig_A],
                           [dispatch,1,1,0,1],
                           [mclear,  *,1,*,1]|0])).

component(iy_comp_A,table([[ts_A,iy_A],
                           [iuse,1]|0])).

component(mg_comp_A,table(
            [[ts_A,ir_A,mr_A,lessn_ic_A,equz_tc_A,iu_A,mu_A,mg_A],
             [iclear,    *,*,*,1,*,*,1],
             [dispatch,  0,1,1,1,0,*,1]|0])).

component(my_comp_A,table([[ts_A,my_A],
                           [muse,1]|0])).

%----- Behavioral description for Counters -------

signal(tc_A_0,bool).
signal(tc_A_1,bool).
signal(tc_A_2,bool).
signal(tc_A_3,bool).
signal(tc_A_4,bool).
signal(tc_A_5,bool).
signal(tc_A_6,bool).
```

```
signal(tc_A_7,bool).

signal(sum0,bool).
signal(sum1,bool).
signal(sum2,bool).
signal(sum3,bool).
signal(sum4,bool).
signal(sum5,bool).
signal(sum6,bool).
signal(sum7,bool).

signal(c0,bool).
signal(c1,bool).
signal(c2,bool).
signal(c3,bool).
signal(c4,bool).
signal(c5,bool).
signal(c6,bool).
signal(c7,bool).
signal(c8,bool).

signal(s1,bool).
signal(s2,bool).
signal(s3,bool).
signal(s4,bool).
signal(s5,bool).
signal(s6,bool).
signal(s7,bool).

signal(tc_0,bool).
signal(tc_1,bool).
signal(tc_2,bool).
signal(tc_3,bool).
signal(tc_4,bool).
signal(tc_5,bool).
signal(tc_6,bool).
signal(tc_7,bool).

signal(c_1,bool).
signal(c_2,bool).
signal(c_3,bool).
signal(c_4,bool).
signal(c_5,bool).
signal(c_6,bool).
signal(c_7,bool).
signal(c_8,bool).

signal(ic_A_0,bool).
signal(ic_A_1,bool).
signal(ic_A_2,bool).
signal(ic_A_3,bool).
signal(ic_A_4,bool).
signal(ic_A_5,bool).
signal(ic_A_6,bool).
signal(ic_A_7,bool).
```

```
signal(ic0,bool).
signal(ic1,bool).
signal(ic2,bool).
signal(ic3,bool).
signal(ic4,bool).
signal(ic5,bool).
signal(ic6,bool).
signal(ic7,bool).

signal(ca0,bool).
signal(ca1,bool).
signal(ca2,bool).
signal(ca3,bool).
signal(ca4,bool).
signal(ca5,bool).
signal(ca6,bool).
signal(ca7,bool).
signal(ca8,bool).

signal(ic_0,bool).
signal(ic_1,bool).
signal(ic_2,bool).
signal(ic_3,bool).
signal(ic_4,bool).
signal(ic_5,bool).
signal(ic_6,bool).
signal(ic_7,bool).

signal(ca_1,bool).
signal(ca_2,bool).
signal(ca_3,bool).
signal(ca_4,bool).
signal(ca_5,bool).
signal(ca_6,bool).
signal(ca_7,bool).
signal(ca_8,bool).

signal(s_1,bool).
signal(s_2,bool).
signal(s_3,bool).
signal(s_4,bool).
signal(s_5,bool).
signal(s_6,bool).
signal(s_7,bool).

signal(tc_plus_A, bool).
signal(tc_min_A,bool).

signal(equz_tc_A, bool).
signal(lessn_ic_A,bool).

component(tc_plus_A, or(input(itc_plus_A, mtc_plus_A),
          output(tc_plus_A))).
component(tc_minus_A,
          or(input(itc_min_A,mtc_min_A),output(tc_min_A))).
```

```
% For increasing 1 of the counter tc_A

component(const1,constant_signal(value(1),signal(c0))).
component(comp_sum0,xor(input(c0,tc_A_0),output(sum0))).
component(comp_carrier0,and(input(c0,tc_A_0),output(c1))).

component(comp_sum1,xor(input(c1,tc_A_1),output(sum1))).
component(comp_carrier1,and(input(c1,tc_A_1),output(c2))).

component(comp_sum2,xor(input(c2,tc_A_2),output(sum2))).
component(comp_carrier2,and(input(c2,tc_A_2),output(c3))).

component(comp_sum3,xor(input(c3,tc_A_3),output(sum3))).
component(comp_carrier3,and(input(c3,tc_A_3),output(c4))).

component(comp_sum4,xor(input(c4,tc_A_4),output(sum4))).
component(comp_carrier4,and(input(c4,tc_A_4),output(c5))).

component(comp_sum5,xor(input(c5,tc_A_5),output(sum5))).
component(comp_carrier5,and(input(c5,tc_A_5),output(c6))).

component(comp_sum6,xor(input(c6,tc_A_6),output(sum6))).
component(comp_carrier6,and(input(c6,tc_A_6),output(c7))).

component(comp_sum7,xor(input(c7,tc_A_7),output(sum7))).
component(comp_carrier7,and(input(c7,tc_A_7),output(c8))).

% For decreasing 1 of the counter tc_A

component(comp_tc_0,not(input(tc_A_0),output(tc_0))).
component(comp_fork,fork(input(tc_A_0),output(c_1))).

component(comp_s1,xor(input(c_1,tc_A_1),output(s1))).
component(comp_tc_1,not(input(s1),output(tc_1))).
component(comp_carri1,or(input(c_1,tc_A_1),output(c_2))).

component(comp_s2,xor(input(c_2,tc_A_2),output(s2))).
component(comp_tc_2,not(input(s2),output(tc_2))).
component(comp_carri2,or(input(c_2,tc_A_2),output(c_3))).

component(comp_s3,xor(input(c_3,tc_A_3),output(s3))).
component(comp_tc_3,not(input(s3),output(tc_3))).
component(comp_carri3,or(input(c_3,tc_A_3),output(c_4))).

component(comp_s4,xor(input(c_4,tc_A_4),output(s4))).
component(comp_tc_4,not(input(s4),output(tc_4))).
component(comp_carri4,or(input(c_4,tc_A_4),output(c_5))).

component(comp_s5,xor(input(c_5,tc_A_5),output(s5))).
component(comp_tc_5,not(input(s5),output(tc_5))).
component(comp_carri5,or(input(c_5,tc_A_5),output(c_6))).

component(comp_s6,xor(input(c_6,tc_A_6),output(s6))).
component(comp_tc_6,not(input(s6),output(tc_6))).
component(comp_carri6,or(input(c_6,tc_A_6),output(c_7))).
```

```
component(comp_s7,xor(input(c_7,tc_A_7),output(s7))).
component(comp_tc_7,not(input(s7),output(tc_7))).
component(comp_carri7,or(input(c_7,tc_A_7),output(c_8))).

% When tc_A reaches 255 and tc_plus_A=1, tc_A remains 255.
% When tc_A reaches 0 and tc_min_A=1, tc_A remains 0.

component(tc_A_0, table(
         [[tc_plus_A, tc_min_A, c8, c_8, sum0, tc_0, n_tc_A_0],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1],[0,1,*,0,*,*,0]
          |tc_A_0] )).

component(tc_A_1, table(
         [[tc_plus_A, tc_min_A, c8, c_8, sum1, tc_1, n_tc_A_1],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |tc_A_1] )).

component(tc_A_2, table(
         [[tc_plus_A, tc_min_A, c8, c_8, sum2, tc_2, n_tc_A_2],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |tc_A_2] )).

component(tc_A_3, table(
         [[tc_plus_A, tc_min_A, c8, c_8, sum3, tc_3, n_tc_A_3],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |tc_A_3] )).

component(tc_A_4, table(
         [[tc_plus_A, tc_min_A, c8, c_8, sum4, tc_4, n_tc_A_4],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |tc_A_4] )).

component(tc_A_5, table(
         [[tc_plus_A, tc_min_A, c8, c_8, sum5, tc_5, n_tc_A_5],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |tc_A_5] )).

component(tc_A_6, table(
         [[tc_plus_A, tc_min_A, c8, c_8, sum6, tc_6, n_tc_A_6],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |tc_A_6] )).

component(tc_A_7, table(
         [[tc_plus_A, tc_min_A, c8, c_8, sum7, tc_7, n_tc_A_7],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |tc_A_7] )).
```

```
% For increasing 1 of the counter ic_A

component(const2,constant_signal(value(1),signal(ca0))).
component(comp_ic0,xor(input(ca0,ic_A_0),output(ic0))).
component(comp_carr0,and(input(ca0,ic_A_0),output(ca1))).

component(comp_ic1,xor(input(ca1,ic_A_1),output(ic1))).
component(comp_carr1,and(input(ca1,ic_A_1),output(ca2))).

component(comp_ic2,xor(input(ca2,ic_A_2),output(ic2))).
component(comp_carr2,and(input(ca2,ic_A_2),output(ca3))).

component(comp_ic3,xor(input(ca3,ic_A_3),output(ic3))).
component(comp_carr3,and(input(ca3,ic_A_3),output(ca4))).

component(comp_ic4,xor(input(ca4,ic_A_4),output(ic4))).
component(comp_carr4,and(input(ca4,ic_A_4),output(ca5))).

component(comp_ic5,xor(input(ca5,ic_A_5),output(ic5))).
component(comp_carr5,and(input(ca5,ic_A_5),output(ca6))).

component(comp_ic6,xor(input(ca6,ic_A_6),output(ic6))).
component(comp_carr6,and(input(ca6,ic_A_6),output(ca7))).

component(comp_ic7,xor(input(ca7,ic_A_7),output(ic7))).
component(comp_carr7,and(input(ca7,ic_A_7),output(ca8))).

% For decreasing 1 of the counter ic_A

component(comp_ic_0,not(input(ic_A_0),output(ic_0))).
component(comp_fork2,fork(input(ic_A_0),output(ca_1))).

component(comp_s_1,xor(input(ca_1,ic_A_1),output(s_1))).
component(comp_ic_1,not(input(s_1),output(ic_1))).
component(comp_ca_1,or(input(ca_1,ic_A_1),output(ca_2))).

component(comp_s_2,xor(input(ca_2,ic_A_2),output(s_2))).
component(comp_ic_2,not(input(s_2),output(ic_2))).
component(comp_ca_2,or(input(ca_2,ic_A_2),output(ca_3))).

component(comp_s_3,xor(input(ca_3,ic_A_3),output(s_3))).
component(comp_ic_3,not(input(s_3),output(ic_3))).
component(comp_ca_3,or(input(ca_3,ic_A_3),output(ca_4))).

component(comp_s_4,xor(input(ca_4,ic_A_4),output(s_4))).
component(comp_ic_4,not(input(s_4),output(ic_4))).
component(comp_ca_4,or(input(ca_4,ic_A_4),output(ca_5))).

component(comp_s_5,xor(input(ca_5,ic_A_5),output(s_5))).
component(comp_ic_5,not(input(s_5),output(ic_5))).
component(comp_ca_5,or(input(ca_5,ic_A_5),output(ca_6))).

component(comp_s_6,xor(input(ca_6,ic_A_6),output(s_6))).
component(comp_ic_6,not(input(s_6),output(ic_6))).
component(comp_ca_6,or(input(ca_6,ic_A_6),output(ca_7))).
```

```
component(comp_s_7,xor(input(ca_7,ic_A_7),output(s_7))).
component(comp_ic_7,not(input(s_7),output(ic_7))).
component(comp_ca_7,or(input(ca_7,ic_A_7),output(ca_8))).

% When ic_A reaches 255 and ic_plus_A=1, ic_A remains 255.
% When ic_A reaches 0 and ic_min_A=1, ic_A remains 0.

component(ic_A_0, table(
         [[ic_plus_A, ic_min_A, ca8, ca_8, ic0, ic_0, n_ic_A_0],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |ic_A_0] )).

component(ic_A_1, table(
         [[ic_plus_A, ic_min_A, ca8, ca_8, ic1, ic_1, n_ic_A_1],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |ic_A_1] )).

component(ic_A_2, table(
         [[ic_plus_A, ic_min_A, ca8, ca_8, ic2, ic_2, n_ic_A_2],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |ic_A_2] )).

component(ic_A_3, table(
         [[ic_plus_A, ic_min_A, ca8, ca_8, ic3, ic_3, n_ic_A_3],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |ic_A_3] )).

component(ic_A_4, table(
         [[ic_plus_A, ic_min_A, ca8, ca_8, ic4, ic_4, n_ic_A_4],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |ic_A_4] )).

component(ic_A_5, table(
         [[ic_plus_A, ic_min_A, ca8, ca_8, ic5, ic_5, n_ic_A_5],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |ic_A_5] )).

component(ic_A_6, table(
         [[ic_plus_A, ic_min_A, ca8, ca_8, ic6, ic_6, n_ic_A_6],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |ic_A_6] )).

component(ic_A_7, table(
         [[ic_plus_A, ic_min_A, ca8, ca_8, ic7, ic_7, n_ic_A_7],
          [1,0,0,*,0,*,0], [1,0,0,*,1,*,1], [1,0,1,*,*,*,1],
          [0,1,*,1,*,0,0], [0,1,*,1,*,1,1], [0,1,*,0,*,*,0]
          |ic_A_7] )).
```

```
component(equz_tc_A,
table([[tc_A_7,tc_A_6,tc_A_5,tc_A_4,tc_A_3,tc_A_2,tc_A_1,tc_A_0,eq
uz_tc_A], [0,0,0,0,0,0,0,0, 1]|0])).

component(lessn_ic_A,
table([[ic_A_7,ic_A_6,ic_A_5,ic_A_4,ic_A_3,ic_A_2,ic_A_1,ic_A_0,
lessn_ic_A], [1,1,1,1,1,1,1,1, 0]|1])).


%--- Initial states ---

init_val(is_A,red).
init_val(ms_A,red).
init_val(ts_A,dispatch).

init_val(tc_A_0,0).
init_val(tc_A_1,0).
init_val(tc_A_2,0).
init_val(tc_A_3,0).
init_val(tc_A_4,0).
init_val(tc_A_5,0).
init_val(tc_A_6,0).
init_val(tc_A_7,0).

init_val(ic_A_0,0).
init_val(ic_A_1,0).
init_val(ic_A_2,0).
init_val(ic_A_3,0).
init_val(ic_A_4,0).
init_val(ic_A_5,0).
init_val(ic_A_6,0).
init_val(ic_A_7,0).

%--- Outputs ---

outputs([irl_A, igl_A, mrl_A, mgl_A, itc_plus_A, mtc_plus_A,
         ic_plus_A, ic_min_A]).

%--- Partitions ---

output_partition([[[irl_A]],[[igl_A]],[[mrl_A]],[[mgl_A]],
                  [[itc_plus_A]], [[mtc_plus_A]],
                  [[ic_plus_A]], [[ic_min_A]]]).

next_state_partition([
[[n_is_A]],
[[n_ms_A]],
[[n_ts_A]],
[[n_tc_A_0]],
[[n_tc_A_1]],
[[n_tc_A_2]],
[[n_tc_A_3]],
[[n_tc_A_4]],
[[n_tc_A_5]],
[[n_tc_A_6]],
```

```
[[n_tc_A_7]],

[[n_ic_A_0]],
[[n_ic_A_1]],
[[n_ic_A_2]],
[[n_ic_A_3]],
[[n_ic_A_4]],
[[n_ic_A_5]],
[[n_ic_A_6]],
[[n_ic_A_7]]
]).


%--- State variable, next state variable mapping---

st_nxst(is_A,n_is_A).
st_nxst(ms_A,n_ms_A).
st_nxst(ts_A,n_ts_A).

st_nxst(tc_A_0,n_tc_A_0).
st_nxst(tc_A_1,n_tc_A_1).
st_nxst(tc_A_2,n_tc_A_2).
st_nxst(tc_A_3,n_tc_A_3).
st_nxst(tc_A_4,n_tc_A_4).
st_nxst(tc_A_5,n_tc_A_5).
st_nxst(tc_A_6,n_tc_A_6).
st_nxst(tc_A_7,n_tc_A_7).

st_nxst(ic_A_0,n_ic_A_0).
st_nxst(ic_A_1,n_ic_A_1).
st_nxst(ic_A_2,n_ic_A_2).
st_nxst(ic_A_3,n_ic_A_3).
st_nxst(ic_A_4,n_ic_A_4).
st_nxst(ic_A_5,n_ic_A_5).
st_nxst(ic_A_6,n_ic_A_6).
st_nxst(ic_A_7,n_ic_A_7).

%--- Partition strategy---

par_strategy(auto, auto).
```

Appendix 3 Verilog Code for ITC

The ITC model with 8 bit counters:

```
`define green    0
`define entering 1
`define red      2
`define exiting  3
`define dispatch 0
`define iuse     1
`define muse     2
`define iclear   3
`define mclear   4

/*============== Main module ==============*/

module main(clk,rst,igl,irl,mgl,mrl);
input clk, rst;
output igl,irl,mgl,mrl;

wire ie,ix,me,mx,igl,irl,mgl,mrl;
wire ic_plus,ic_minus,itc_plus,itc_minus,mtc_plus,mtc_minus;
wire [7:0] tc,ic;

sensor sensor(clk,rst, ie,ix,me,mx);
counter counter(clk,rst,tc,ic,ic_plus,ic_minus,
              itc_plus,itc_minus,mtc_plus,mtc_minus);
island island(clk,rst,ie,ix,igl,irl,ic_minus,
              itc_plus,itc_minus,iu,ir,ig,iy);
mainland mainland(clk,rst,me,mx,mgl,mrl,ic,ic_plus,
              mtc_plus,mtc_minus,mu,mr,mg,my);
tunnel tunnel(clk,rst,iu,ir,ig,iy,mu,mr,mg,my,tc,ic);

endmodule

module sensor(clk,rst,ie,ix,me,mx);
input clk,rst;
output ie,ix,me,mx;

wire rand_choice1,rand_choice2,rand_choice3,rand_choice4;
reg ie,ix,me,mx;

always @(posedge clk)
begin
  if (rst==1'b1) // reset all flops
    begin
      ie = 0;
      ix = 0;
      me = 0;
      mx = 0;
```

```
     end
    else
    begin
     if (rand_choice1==0)
         ie = 0;
     else
       ie = 1;
     if (rand_choice2==0)
         ix = 0;
     else
       ix = 1;
     if (rand_choice3==0)
         me = 0;
     else
       me = 1;
     if (rand_choice4==0)
         mx = 0;
     else
       mx = 1;
    end
end
endmodule

/*=================== Counters ====================*/

module counter(clk,rst, tc,ic,

ic_plus,ic_minus,itc_plus,itc_minus,mtc_plus,mtc_minus);
input clk,rst;
input ic_plus,ic_minus,itc_plus,itc_minus,mtc_plus,mtc_minus;
output tc,ic;

reg [7:0] tc,ic;
wire ic_plus,ic_minus,itc_plus,itc_minus,mtc_plus,mtc_minus;

always @(posedge clk)
begin
  if (rst==1'b1) // reset all flops
    begin
     tc = 0;
     ic = 0;
    end
   else
   begin
     if ((ic_minus==1)&&(ic > 0)) ic = ic - 1;
     else if ((ic_plus==1)&&(ic<255)) ic = ic + 1;
     else ic = ic;

     if  ((itc_minus==1)&&(tc>0))  tc = tc - 1;
     else if ((itc_plus==1) &&(tc<255)) tc = tc + 1;
     else if ((mtc_minus==1)&&(tc>0))  tc = tc - 1;
     else if ((mtc_plus==1) &&(tc<255)) tc = tc + 1;
     else tc = tc;
   end
end
endmodule
```

```
/*============== Island Light Controller ==============*/

module island(clk,rst,ie,ix,igl,irl,ic_minus,
              itc_plus,itc_minus,iu,ir,ig,iy);

input clk,rst;
input ie,ix,ig,iy;
output igl,irl,ic_minus,itc_plus,itc_minus,iu,ir;

wire ie,ix,ig,iy,igl,irl,iu,ir;
wire ic_minus,itc_plus,itc_minus;

reg [1:0] is;

always @(posedge clk)
begin
  if (rst==1'b1) // reset all flops
    begin
      is = ``red;
    end
   else
    begin
      case (is)
      `green:    if ((iy==0)&&(ie==0)) is = `green;
            else if ((iy==0)&&(ie==1)) is = `entering;
            else is = `red;
      `entering: if (ie==0) is = `green;
            else is = `entering;
      `red:      if ((ix==0)&&(ig==0)) is = `red;
            else if ((ix==0)&&(ig==1)) is = `green;
            else is = `exiting;
      `exiting:  if (ix==0) is = `red;
            else is = `exiting;
      endcase
    end
end

assign ir = ((is==`red)&&(ie==1)) ? 1 : 0;

assign iu = ((is==`green)||(is==`entering)) ? 1 : 0;

assign irl = ((is==`red)||(is==`exiting)) ? 1 : 0;

assign igl = ((is==`green)||(is==`entering)) ? 1 : 0;

assign itc_plus = ((is==`green)&&(iy==0)&&(ie==1)) ? 1 : 0;

assign itc_minus = ((is==`red)&&(ix==1)) ? 1 : 0;

assign ic_minus = ((is==`green)&&(iy==0)&&(ie==1)) ? 1 : 0;

endmodule
```

```
/*============== Mainland Light Controller ===============*/

module mainland(clk,rst,me,mx,mgl,mrl,ic,ic_plus,
                mtc_plus,mtc_minus,mu,mr,mg,my);

input clk,rst;
input [7:0] ic;
input me,mx,mg,my;
output mgl,mrl,ic_plus,mtc_plus,mtc_minus,mu,mr;

wire [7:0] ic;
wire me,mx,mg,my;
wire mgl,mrl,ic_plus,mtc_plus,mtc_minus,mu,mr;

reg [1:0] ms;

always @(posedge clk)
begin
  if (rst==1'b1) // reset all flops
    begin
       ms = `red;
    end
   else
    begin
      case (ms)
      `green:    if (ic<255) ms = `red;
            else if ((my==0)&&(me==0)) ms = `green;
            else if ((my==0)&&(me==1)) ms = `entering;
            else ms = `red;
      `entering: if (me==0) ms = `green;
            else ms = `entering;
      `red:      if ((mx==0)&&(mg==0)) ms = `red;
            else if ((mx==0)&&(mg==1)) ms = `green;
            else ms = `exiting;
      `exiting: if (mx==0) ms = `red;
            else ms = `exiting;
      endcase
    end
end

assign mr = ((ms==`red)&&(me==1)) ? 1 : 0;

assign mu = ((ms==`green)||(ms==`entering)) ? 1 : 0;

assign mrl = ((ms==`red)||(ms==`exiting)) ? 1 : 0;

assign mgl = ((ms==`green)||(ms==`entering)) ? 1 : 0;

assign mtc_plus =((ms==`green)&&(my==0)&&(me==1)
                  &&(ic<255)) ? 1 : 0;

assign mtc_minus = ((ms==`red)&&(mx==1)) ? 1 : 0;

assign ic_plus = ((ms==`green)&&(my==0)&&(me==1)&&(ic<255))? 1:0;

endmodule
```

```
/*================= Tunnel Controller ==================*/

module tunnel(clk,rst,iu,ir,ig,iy,mu,mr,mg,my,tc,ic);

input clk,rst;
input [7:0] ic, tc;
input iu,ir,mu,mr;
output ig,iy,mg,my;

wire [7:0] ic, tc;
wire iu,ir,mu,mr;
wire ig,iy,mg,my;

reg [2:0] ts;

always @(posedge clk)
begin
  if (rst==1'b1) // reset all flops
    begin
      ts = `dispatch;
    end
   else
    begin
      case (ts)
      `dispatch: if ((ir==0)&&(mr==0))
                    ts = `dispatch;
            else if ((ir==0)&&(mr==1)&&(ic>=255))
                    ts = `dispatch;
            else if ((ir==0)&&(mr==1)&&(ic<255)&&(iu==1))
                    ts = `iuse;
            else if ((ir==0)&&(mr==1)&&(ic<255)&&(iu==0)&&(tc!=0))
                    ts=`iclear;
            else if ((ir==0)&&(mr==1)&&(ic<255)&&(iu==0)&&(tc==0))
                    ts=`dispatch;
            else if ((ir==1)&&(mu==1))
                    ts=`muse;
            else if ((ir==1)&&(mu==0)&&(tc!=0))
                    ts=`mclear;
            else    ts=`dispatch;
      `iuse:     if (iu==0) ts = `iclear;
            else ts = `iuse;
      `muse:     if (mu==0) ts = `mclear;
            else ts = `muse;
      `iclear:   if (tc!=0) ts = `iclear;
            else ts = `dispatch;
      `mclear:   if (tc!=0) ts = `mclear;
            else ts = `dispatch;
      endcase
    end
end

assign ig = (((ts==`dispatch)&&(ir==1)&&(tc==0)&&(mu==0)) ||
            ((ts==`mclear)&&(tc==0))) ? 1 : 0;
```

```
assign iy = (ts==`iuse) ? 1 : 0;

assign mg = (((ts==`dispatch)&&(ir==0)&&(mr==1)&&(ic<255)&&
            (tc==0)&&(iu==0)) || ((ts==`iclear)&&(tc==0)))? 1 : 0;

assign my = (ts==`muse) ? 1 : 0;

endmodule
```