# First Order Model Checking of $\omega$-Automata using Multiway Decision Graphs

Fang Wang

A Thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

Apirl 2005

this page is replaced by the signature page

## Abstract


**First Order Model Checking of $\omega$-Automata using Multiway Decision Graphs**



Fang Wang, Ph.D.
Concordia University, 2005



As the complexity of hardware digital systems increases, their correctness becomes a major concern. Traditional verification by simulation is infeasible to exhaustively test and guarantee correctness. More than a decade ago, however, formal verification has been introduced as complement technique to simulation. Formal methods establish that a design implementation satisfies its specification by mathematical reasoning. Among several techniques, model checking is one of the most successful technology, which is based on the exploration of the design state space. In this thesis, we propose a new model checking method based on the theory of $\omega$-automata and multiway decision graphs (MDGs). Unlike reduced ordered binary decision diagrams (ROBDDs), MDGs allow system models to be described using abstract state machines (ASMs) through abstract data sorts and uninterpreted function symbols, hence enabling the verification of larger designs independent of the datapath width.

Given an ASM and a first-order linear time temporal logic property, the model checking problem proposed in this thesis is reduced to a language emptiness checking of an $\omega$-automaton that accepts all $\omega$-words produced by the system violating the property formula. The checking method comprises four steps: (1) transforming the first-order property into a propositional formula by constructing ASMs for the

atomic formulas of the property; (2) generating an $\omega$-automaton from the negation of the transformed propositional formula; (3) computing the product of the generated automaton, the system model ASM and the constructed ASMs; and (4) applying a language emptiness checking algorithm on the product automaton. Three different checking algorithms have been developed, implemented, and proved correct in this thesis.

To evaluate the performance of the proposed model checking method and implemented tool, we conducted several experimentations and case studies. We also compared the efficiency of our tool with an existing MDG regular model checking application, as well as with popular ROBDD-based automata model checking tools such as VIS. Our model checker was found to be outperforming the above tools.

# Acknowledgments

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

I am especially grateful to thank Dr. Otmane Ait Mohamed for many discussions and helpful suggestions, which are invaluable to this thesis. I would like also to thank my PhD committee members: Dr. J. Chen, Dr. Xiaoyu Song, Dr. J. Paquet, Dr. V. Ramachandran, and Dr. O. Ait Mohamed for reviewing my thesis and giving me invaluable feedbacks.

Furthermore, I would like to thank my friends and fellow graduate students in the hardware verification group (HVG) for their help and discussions.

Last, but not least, I would like to thank all my family for their support and encouragement for my studies.

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| ASM | Abstract State Machine |
| ATM | Asynchronous Transfer Mode |
| BTTL | Branching Time Temporal Logic |
| CTL | Computation Tree Logic |
| DNF | Disjunctive Normal Form |
| FCD | Fair Cycle Detection |
| FO | First Order |
| FOBTL | First Order Branching time Temporal Logic |
| FOTL | First Order Temporal Logic |
| FOLTL | First Order Linear time Temporal Logic |
| FSM | Finite State Machine |
| GSH | Generic SCC-Hull |
| GBA | Generalized Büchi Automata |
| ITC | Island Tunnel Controller |
| LC | Language Containment |
| LEC | Language Emptiness Checking |
| LHS | Left Hand Side |
| MC | Model Checking |
| MDG | Multiway Decision Graph |
| PLTL | Propositional Linear time Temporal Logic |
| PTL | Propositional Temporal Logic |
| ReAn | Reachability Analysis |
| RHS | Right Hand Side |
| ROBDD | Reduced Ordered Binary Decision Diagrams |
| RTL | Register Transfer Level |
| SCC | Strongly Connected Component |
| TL | Temporal Logic |

# Chapter 1

# Introduction

During the last decades, technological advances in microelectronics have greatly increased the complexity of digital hardware designs. Their correctness thus becomes a major concern, especially in critical applications where failure is unacceptable. Traditionally, simulation is the only tool to validate a design. Using simulation, the designer needs to create a set of test vectors that represents the possible inputs to the system. The outputs for each of these test vectors are compared with the expected responses. This method is very costly and incomplete because of the large number of input sequences. In almost all practical situations it is infeasible to exhaustively simulate a design to guarantee its correctness.

As a complement to simulation, formal verification methods intend to establish an implementation satisfies a specification by mathematical reasoning [42]. The implementation refers to the hardware design to be verified and the specification refers to the property with respect to which correctness is to be determined. Formal verification conducts an exhaustive exploration of all the possible behaviors. Thus, when a design is pronounced correct by a formal verification method, it implies that all behaviors relative to the property have been explored [19].

Logic is a formalism to represent specification. The hierarchy of logics is arranged according to the generality of their data types and operators. In propositional logic, only propositional variables, Boolean operators and their derivations are allowed. First-order logic is much more general than propositional logic in that it allows variables over one or more other types. It also allows constant operators, functions and predicates over added types. Higher order logic is more general than first-order logic in that it allows variables and operators over functions and predicates. Therefore, functions and predicates can be defined and manipulated as objects by themselves.

Formal verification is a very broad and well studied research topic and numerous achievements have been contributed to the literature. This thesis focuses on the particular techniques of model checking based on the $\omega$-automata theory [79]. In this introduction, we first describe the background of this thesis. We then survey the literature and present related work. Finally, we summarize the scope of this thesis in Section 1.3 and give an outline of the thesis in Section 1.4.

## 1.1 Background

Formal verification methods can be classified in two main categories: interactive verification with a theorem prover and automated Finite State Machine (FSM) verification based on state enumeration [42].

Interactive verification with a theorem prover uses a powerful formalism such as higher order logic that allows the verification problem to be stated at many levels of abstraction. This approach has achieved significant successes in verifying microprocessor designs. However, interactive verification has the drawback that the user is responsible for coming up with the proof of correctness and feeding it to the theorem prover, which requires great expertise [37].

Automated FSM verification based on state enumeration techniques provides automation for behavior comparison and model checking. Model checking works on a finite-state model of the system to be verified and the logical specification of the desired behavior of the system model. It algorithmically checks if the finite state machine is a model of the specification formula. Since model checking can be completely automatic and has been used successfully to verify complex sequential circuit designs and communication protocols, it is emerging as an industrial standard tool for hardware design [58].

Model checking algorithms are based on exploring the reachable state space of design models. The state space of a finite state model for a concurrent system, however, grows exponentially as the number of components of the system increases. This is known as the state explosion problem in automatic verification, which is the main challenge of model checking.

The most promising approach to tackle the state explosion problem has been the application of ROBDDs (Reduced Ordered Binary Decision Diagrams) to the representation of state graphs [10]. ROBDDs encode the set of states as well as transition and output relations and perform an implicit enumeration of the state space, making it possible to verify finite state machines with a larger number of states [5]. ROBDDs have been proved to be a powerful tool for automated hardware verification. However, because they require a Boolean representation of the circuit, ROBDD-based verification cannot be directly applied to circuits with complex and large datapath [20].

To overcome this limitation, Multiway Decision Graphs (MDGs) [12] have been proposed as a new class of decision graphs, of which ROBDDs are a special case. MDGs efficiently represent a class of well formed formulas of a many-sorted first-order logic with a distinction of abstract and concrete sorts [21]. In an MDG, a data signal is represented by a single variable of abstract sort rather than by a vector of Boolean variables, and a data operation is represented by an uninterpreted function symbol. Therefore, MDGs are much more compact than ROBDDs for circuits having complex and large datapaths.

Many ROBDD based (propositional) verification methods have been generalized into a first order level and developed using MDGs. The MDG based verification tools verify designs at the Register Transfer Level (RTL) using their models of ASM (Abstract State Machine), where MDGs encode the sets of (abstract) states and transition/output relations. The current MDG verification tools include an MDG package, a reachability analysis procedure, and applications for formal hardware verification. The MDG package implements manipulation algorithms of MDGs, the reachability algorithm checks that an invariant holds on the reachable states of an ASM using abstract implicit enumeration techniques. The existing MDG tool set provides the following verification applications: combinational verification, sequential equivalence checking, invariant checking and a first-order model checking [1, 12, 21, 81].

To complete the MDG tools, this thesis proposes to lift the $\omega$-automata based model checking method to a First-order Temporal Logic (FTL) model checking using MDGs. $\omega$-automata based model checking is a well studied and widely used model checking technique. Many research tools, such as VIS [9], SMV [62], SPIN [44] etc., have this application and the most successful commercial model checking tool FormalCheck [58] was built on this idea. Model checking of $\omega$-automata accepts a FSM modeling system design and a propositional formula as the property. Both of them are transformed into $\omega-$automata [57], and the model checking reduces to checking if the language (behaviors) of the system automaton $\mathcal{L}(M)$ is contained in the language (behaviors) of the property automaton $\mathcal{L}( \mathcal{B}_{\phi})$ [57]. By constructing an $\omega$-automaton $\mathcal{B}_{\neg\phi}$ accepting all the language (behaviors) violating the specification, the model checking is converted into a Language Emptiness Checking (LEC) on the product automaton $M \times \mathcal{B}_{\neg\phi}$. If the language is not empty, this means there is a behavior produced by the design not accepted by the specification. In this case, the specification fails on the design. Otherwise, all the behaviors produced by the system are contained in the behaviors of the property, and the property is verified on the design [7, 57, 79].

In this thesis, we present a model checking based on the $\omega-$automata theory using MDGs. Our approach checks a first-order temporal logic specification on an abstract of the state machine model, where the data signal is described as a single variable of abstract type rather than by a vector of Boolean variables, and a data operation is represented by a function symbol rather than by vectors of Boolean variables.

## 1.2 Related Work

The application of symbolic techniques to the model checking using $\omega$-automata theory has been a hot topic since 1980s, and thus numerous achievements have been reported in the literature. However, lifting this method to the first-order logic is a pretty new research direction. In this section we survey relevant first-order model checking techniques available in the literature.

The most related work to our thesis is the MDG based regular model checking technique (MDG MC) developed by Xu *et. al.* [81]. They defined a first-order branching time temporal logical with accepting existential quantifications and abstract variables, called $Abstract\_CTL^*$. They then obtained a subset, called $\mathcal{L}_{MDG}$ [82], by restricting $Abstract\_CTL^*$ to the universal quantification that can only appear at the beginning of the formula, and using a limited nesting of temporal formulas. This limited nesting of $\mathcal{L}_{MDG}$ is defined by the following templates: $\mathsf{A}(P)$, $\mathsf{AG}(P)$, $\mathsf{AF}(P)$, $\mathsf{A}(P)\mathsf{U}(Q)$, $\mathsf{AG}(P \Rightarrow (\mathsf{F}(Q))$, and $\mathsf{AG}((P) \Rightarrow ((Q)\mathsf{U}(R))$, where $P$, $Q$, and $R$ are next-let-formulas. A next-let-formula is composed by well-typed equations, Boolean connectives and temporal operator $X$ (nexttime). To check an $\mathcal{L}_{MDG}$ formula $\phi$ on an ASM $M$, for each next-let-formula $p$, MDG MC first constructs an abstract state machine. It then composes $M$ and all the constructed ASMs to produce a composed machine $P$. It finally checks a simplified property on the composed machine. Both MDG MC and our proposed MDG LEC, which will be described in Chapter 4, model the system design using ASM and construct circuit descriptions for the formula to be verified; however, there are significant differences between them: First, the MDG LEC method is based on the theory of $\omega$-automata, and the approach uses an Language Containment (LC) checking algorithm to check the satisfaction of any property, while MDG MC uses a different checking algorithm for each property template. Second, MDG LEC uses a broader FTL, which we will call ${\mathcal{L}_{MDG}}^*$, to describe the property. ${\mathcal{L}_{MDG}}^*$ breaks the template limitation of $\mathcal{L}_{MDG}$ and allows arbitrary temporal nesting. For example, an ${\mathcal{L}_{MDG}}^*$ formula $\mathsf{G}(a = 1 \rightarrow \mathsf{F}(b = 1) \wedge \mathsf{F}(c = 1))$ is not allowed by $\mathcal{L}_{MDG}$.

Bohn *et. al* [6, 50] presented an algorithm for checking a First Order (FO) ACTL (a subset of computation tree logic restricted to universal quantification) specification on FO Kripke structures. The FO Kripke structure is an extension of the "ordinary" Kripke structures by transitions represented with conditional assignments. The FO ACTL algorithm models data values and operations by means of FO predicates. The algorithm separates the control parts of a system under verification from its data parts and refines the resulting model into an intermediate description that contains sufficient information about the property to be checked. The enrichment of the control flow model introduces *verification conditions*, which can be generated automatically. If a property does not contain predicates on data, then ROBDD model checking is applicable. Otherwise, all FO predicates are substituted by *true*. If the model checking procedure reports a failure, this failure will be part of the control flow and occurs on the concrete model as well. If no failure can be found, correctness is not guaranteed in the original model since failure in the abstracted data cannot be detected. Therefore, the generated verification conditions have to be proven using a theorem prover [6, 50]. Compared to this work, our logic is less expressive since ${\mathcal{L}_{MDG}}^*$ cannot accept existential quantification. However, in our approach the property is checked on the whole model automatically, while in [6, 50] a theorem prover is needed to validate the first-order verification conditions. Besides, our method can be applied to any finite state models, while their application was limited to designs with a clear separation between data and control part and terminating data loops, where the control parts only allow a bounded number of computation on data.

Namjoshi and Kurshan [66] presented an algorithm to verify an FO model by a syntactic abstraction. The FO model is defined on a set of variables, and each variable has an associated domain of values. The initial states and transition relation of the model are defined as quantifier free first-order predicates. The property is described as a formula composed of $ACTL^*$ (a subset of the branching time temporal logic restricted to universal quantification) operator and first-order predicates. To verify the FO model, they first transform it into a finite Kripke structure by the syntactic abstraction, and then proceed the model checking with a propositional model checker. The syntactic abstraction starts with the set of predicates from the property formula, iteratively computes the predicates required for the abstraction relating to that property, and represents these predicates by Boolean variables in the finite model. Compared to this work, we model the system design as an ASM, where abstract variables and uninterpreted function symbols can be used. Although our property language cannot express quantifications, it is defined on atomic formulas within temporal operator $X$. Furthermore, our model checking implements an on-the-fly method, namely, abstraction and model checking are proceeded in parallel.

There exists other work on abstract model checking [46, 17] which combine abstraction interpretation and model checking to improve the automatic verification of infinite systems. Abstraction interpretation transforms the infinite system model $M$ into an finite model $M^+$. Usually $M^+$ is an over-approximation of the original model $M$, which means that given a state $s$ of $M$ and a trace $t$ produced by $M$, it is possible to find a state $s^+$ of $M^+$ and a trace $t^+$ produced by $M^+$ representing $s$ and $t$, respectively. However this transformation cannot preserve the negative correctness, that is, the failure of a property in the finite model does not imply that the property will fail in the original model. Two techniques have been successfully developed to construct $M^+$. The *predicate abstraction* [17] approach consists of substituting some selected model expressions with Boolean variables, which leads to important simplification. In contrast, the *data abstraction* [46] method reduces the type of certain data by transforming its original domain into an approximate and simpler domain [17]. Compared to these work, our method generates an approximation model which ensures both negative preservation and positive preservation. Moreover, most of these approaches are done manually, but ours is totally automatic.

## 1.3 Scope of the Thesis

This thesis explores a model checking based on $\omega$-automata using MDGs. While traditional ROBDD based method accepts Finite State Machine (FSM) as the system model, our method is based on Abstract State Machine(ASM), which uses abstract variables to model the data signals and function symbols to model the data operations. Due to the appearance of abstract variables and function symbols, we cannot directly implant ROBDD based methods into the MDG tool set.

To develop this new application, we define the syntax and semantics of the first-order specification language $\mathcal{L}_{MDG}{}^*$. We also propose an algorithm to transform a $\mathcal{L}_{MDG}{}^*$ formula into a Propositional Linear time Temporal Logic (PLTL) formula by constructing the ASMs. The algorithm is implemented by building circuit descriptions in MDG-HDL for the constructed ASMs. The constructed ASMs are further composed with the system model ASM to produce a composed ASM. Finally, we check the satisfaction of the PLTL on the composed ASM with an existing automaton constructing procedure and three new developed language emptiness checking algorithms.

The contributions of the thesis can be summarized as follows:

- The definition of a first-order specification language ${\mathcal{L}_{MDG}}^*$.
- The development of an algorithm to translate a ${\mathcal{L}_{MDG}}^*$ formula into a PLTL formula by constructing ASMs and proof of the correctness.
- The definition of rules for building the circuit descriptions for ASMs.
- The development of three language emptiness checking algorithms using MDG operators.
- Implementation of the new MDG LEC application into the MDG tool set.
- Performing case studies and experimental work on a set of benchmarks such as an ATM (Asynchronous Transfer Mode) switch fabric and Island Tunnel Controller (ITC).

## 1.4 Outline of the Thesis

The rest of the thesis is organized as follows:

Chapter 2 introduces MDGs and the ASM system modeling technique. We also present the existing ASM-based model checking method.

Chapter 3 reviews model checking techniques including Kripke structure models, $\omega$-automata theory, property specification languages and $\omega$-automata model checking techniques.

Chapter 4 proposes the MDG LEC structure. We describe the first-order temporal specification language ${\mathcal{L}_{MDG}}^*$. We also present the transformation algorithm which constructs ASMs and builds the circuit descriptions in MDG-HDL language.

Chapter 5 presents three language emptiness checking algorithms and proves their soundness.

In Chapter 6, several case studies are presented and some experimental results are provided.

Finally, conclusions and future directions of research are stated in Chapter 7.

# Chapter 2

# Multiway Decision Graphs

Multiway Decision Graphs (MDGs) are a data structure introduced to symbolically encode Abstract State Machines (ASMs) to model hardware designs at the Register Transfer Level (RTL). MDG was first proposed by Corella, *et. al* [20, 21]. Using MDGs, a data value can be represented by a single variable of abstract type, rather than by a vector of Boolean variables, and a data operation is represented by an uninterpreted function symbol, rather than vectors of Boolean variables.

In this chapter, we review the basic concepts of MDGs and MDG-related verification techniques. Section 2.1 describes the underlying formal logic of MDG. Section 2.2 introduces the *directed formulas*. In Section 2.3, we reviews the MDG data structure and describes a set of logic operators on MDGs. Section 2.4 gives the definition of an ASM and its implicit state enumeration technique. This chapter is concluded with the presentation of the MDG tool set.

## 2.1 Formal Logic

### 2.1.1 Syntax

The formal logic underlying MDG is a many-sorted first-order logic, augmented with the distinction between abstract sorts and concrete sorts [20]. This distinction is motivated by the natural division of datapath and control circuitry in RTL designs.

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

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

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

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

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

### 2.1.2 Semantics

An interpretation is a mapping $\psi$ that assigns a denotation to each sort, constant and function symbol such that [21]:

1. The denotation $\psi(\alpha)$ of an abstract sort $\alpha$ is a non-empty set.
2. If $\alpha$ is a concrete sort with enumeration $\{a_1, \cdots, a_n\}$, then $\psi(\alpha) = \{\psi(a_1), \cdots, \psi(a_n)\}$, and $\psi(a_i) \neq \psi(a_j)$ for $1 \leq i \leq j \leq n$.
3. If $c$ is a generic constant of sort $\alpha$, then $\psi(c) \in \psi(\alpha)$.
4. If $f$ is a function symbol of type $\alpha_1 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$, then $\psi(f)$ is a function mapping from $\psi(\alpha_1) \times \cdots \times \psi(\alpha_n)$ into the set $\psi(\alpha_{n+1})$.
5. Let $X$ be a set of variables, a variable assignment with a $\psi-$compatible interpretation is a function $\phi$ that maps every variable $x \in X$ of sort $\alpha$ to an element of $\phi(x)$ of $\psi(\alpha)$.

The truth of a formula $P$ under an interpretation $\psi$ and $\psi-$compatible variable assignment $\phi$ whose domain contains the variables that occur free in $P$, written $\psi, \phi \models P$ is defined by induction:

- $\psi, \phi \models A_1 = A_2$ iff $A_1$ and $A_2$ have the same denotation

- $\psi, \phi \models \neg P$ iff it is not the case $\psi, \phi \models P$
- $\psi, \phi \models P \wedge Q$ iff $\psi, \phi \models P$ and $\psi, \phi \models Q$
- $\psi, \phi \models P \vee Q$ iff $\psi, \phi \models P$ or $\psi, \phi \models Q$
- $\psi, \phi \models (\exists x)P$ iff $\psi, \phi' \models P$ for some $\phi'$ that assigns an arbitrary value to $x$ and otherwise coincides $x$ with $\phi$.

We write $\Phi_X^{\psi}$ for the set of $\psi-$compatible assignments to the variables in $X$. Formula $\psi \models P$ when $\psi, \phi \models P$ for every $\psi-$compatible assignment $\phi$ to the variables that occur freely in $P$, and $\models P$ when $\psi \models P$ for all $\psi$. Two formulas $P$ and $Q$ are logically equivalent iff $\models P \Leftrightarrow Q$. A formula $P$ logically implies a formula $Q$ iff $\models P \Rightarrow Q$.

## 2.2 Directed Formulas

MDG provide efficient representation to a class of well-formed first-order formulas defined on well-typed 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 of type $U \to V$ is a formula in Disjunctive Normal Form (DNF) such that:

1. Each disjunct is a conjunction of equations of the form
   - $A = a$, where $A$ is a term of concrete sort $\alpha$ of the form $f(B_1, \cdots, B_n)$ ($f$ is thus a cross-operator) that contains no variables other than elements of $U$, and $a$ is an individual constant in the enumeration of $\alpha$, or
   - $w = a$, where $w \in (U \cup V)$ is a variable of concrete sort $\alpha$, and $a$ is an individual constant in the enumeration of $\alpha$, or
   - $v = A$, where $v \in V$ is a variable of abstract sort $\alpha$ and $A$ is a term of type $\alpha$ containing no variables other than elements of $U$.
2. In each disjunct, LHSs of the equations are pairwise distinct.

3. Every abstract variable $v \in V$ appears as the LHS of an equation $v = A$ in each of the disjuncts. Note that there need not be an equation $v = a$ for every concrete variable $v \in V$.

Intuitively, in a *directed formula* of type $U \to V$, the $U$ variables play the role of independent variables, the $V$ variables play the role of dependent variables, and the disjuncts enumerate possible cases. In each disjunct, the equations of the form $u = a$ and $A = a$ specify a case in terms of the $U$ variables while other equations specify the values of (some of the) $V$ variables in that case. The cases need not be mutually exclusive, nor exhaustive.

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

## 2.3 Multiway Decision Graphs

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

**Definition 2.3.1** *A multiway decision graph is a finite directed acyclic graph $G$ where the leaf nodes are labeled by formulas, the internal nodes are labeled by terms, and the edges issuing from an internal node $N$ are labeled by terms of the same sort as the label of $N$. Such a graph represents a formula defined inductively as follows:*

- *if $G$ consists of a single leaf node labeled by a formula $P$, then $G$ represents $P$;*
- *if $G$ has a root node labeled $A$ with edges labeled $B_1, \ldots, B_n$ leading to subgraphs $G_1, \cdots, G_n$, and if each $G_i$ represents a formula $P_i$, then $G$ represents*

*the formula* $\bigvee_{1 \leq i \leq n}((A = B_i) \wedge P_i)$.

To illustrate the above definitions, we give an example.

**Example 2.3.1** *Figure 2.1 shows the ALU of a microprocessor. The variables* $x_1, x_2$ *and* $y$ *representing the data inputs and the output are of an abstract sort, while the variable* $x_0$ *representing the control input is of concrete sort with the enumeration* $\{0, 1, 2, 3\}$. *Depending on the value of* $x_0$*, the ALU can add, subtract, increment, or produce zero. The operations are represented by function symbols add, sub, inc. The symbol zero is a generic constant.*

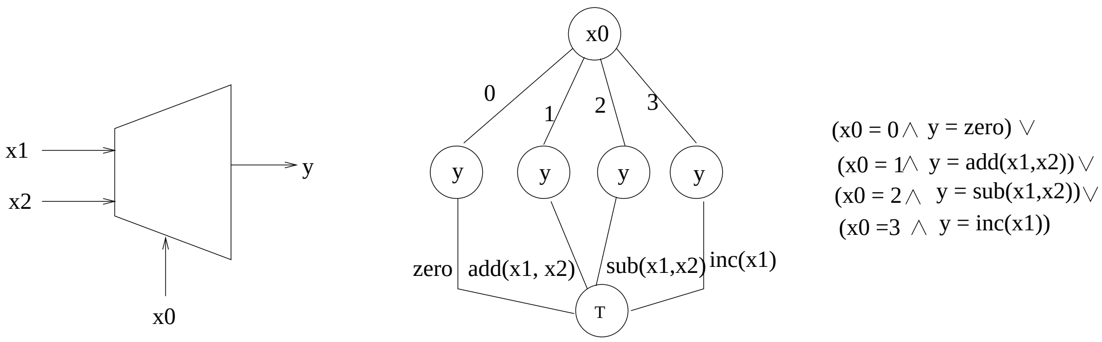



Figure 2.1: An MDG example

### 2.3.1 Basic MDG Algorithms

The following basic MDG algorithms were implemented in the MDG package [85]. To simplify the description of the algorithms we identify an MDG with the *directed formula* that it represents.

**Disjunction**: The disjunction algorithm is $n$-ary. It takes as inputs a set of *directed formulas* $P_i$, $1 \leq i \leq n$, of type $U_i \rightarrow V$, and produces a *directed formula* $R = \mathsf{Disj}(\{P_i\}_{1 \leq i \leq n})$ of type $(\cup_{1 \leq i \leq n} U_i) \rightarrow V$ such that

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

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

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

$$((\cup_{1 \leq i \leq n} U_i) \setminus (\cup_{1 \leq i \leq n} V_i)) \longrightarrow (\cup_{1 \leq i \leq n} V_i).$$

such that

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

Note that for $1 \leq i < j \leq n$, $V_i$ and $V_j$ need not have any abstract variables in common, otherwise the conjunction cannot be computed.

**Relational product**: This algorithm takes as inputs a set of *directed formulas* $P_i, 1 \leq i \leq n$, of types $U_i \rightarrow V_i$, a set of variables $E$ to be existentially quantified, and a renaming substitution $\eta$, and produces a *directed formula* $R = Relp(\{P_i\}_{1 \leq i \leq n}, E, \eta)$ such that

$$\models R \Leftrightarrow (((\exists E)(\wedge_{1 \leq i \leq n} P_i)) \cdot \eta).$$

The algorithm computes the conjunction of the $P_i$, existentially quantifies the variables in $E$, and applies the renaming substitution $\eta$. For $1 \leq i < j \leq n$, $V_i$ and $V_j$ must not have any abstract variables in common. The result of only computing the conjunction is a *directed formula* of type

$$((\cup_{1 \leq i \leq n} U_i) \setminus (\cup_{1 \leq i \leq n} V_i)) \rightarrow (\cup_{1 \leq i \leq n} V_i).$$

The set $E$ of variables to be existentially quantified must be a subset of $(\cup_{1 \leq i \leq n} V_i)$. The result of only computing conjunction and existential quantification would be a *directed formula* of type

$$((\cup_{1 \leq i \leq n} U_i) \setminus (\cup_{1 \leq i \leq n} V_i)) \rightarrow ((\cup_{1 \leq i \leq n} V_i) \setminus E).$$

The domain $\eta$ must be a subset of $((\cup_{1\leq i\leq n}V_i) \setminus E)$. The type of the result $R$ is then

$$(\cup_{1\leq i\leq n}U_i) \setminus (\cup_{1\leq i\leq n}V_i)) \rightarrow (((\cup_{1\leq i\leq n}V_i) \setminus E) \cdot \eta).$$

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

$$\models R \vee (\exists U)Q \Leftrightarrow P \vee (\exists U)Q. \tag{2.1}$$

**Remark 2.3.1** *The name of the algorithm comes from the fact that the disjuncts that are removed from $P$ are subsumed by $Q$.*

Since $R$ is derived from $P$ by pruning, after the formulas represented by $R$ and $P$ have been converted to Disjunctive Normal Form (DNF), the disjuncts in the DNF of $R$ are a subset of those in the DNF of $P$. Therefore, we obtain $\models R \Leftrightarrow P$. And from (2.1), it follows tautologically that

$$\models P \wedge \neg(\exists U)Q \Rightarrow R, \tag{2.2}$$

from which, we have

$$\models (P \wedge \neg(\exists U)Q \Rightarrow R) \wedge (R \Rightarrow P). \tag{2.3}$$

Hence, we can view $R$ as an approximation to the logical difference of $P$ and $(\exists U)Q$. In general, there is no *directed formula* logically equivalent to $P \wedge \neg(\exists U)Q$. If $R$ is $\mathsf{F}$, then it follows tautologically from (2.1) that $\models P \Rightarrow (\exists U)Q$.

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

## 2.4 Abstract State Machine (ASM)

An ASM is a finite state machine given by an abstract description in terms of *directed formula* [20].

### 2.4.1 Representing Sets using MDGs

Let $P$ be an MDG of type $U \rightarrow V$. Then, for a given interpretation $\psi$, $P$ can be used to represent the set of vectors $Set_V^{\psi}(P) = \{\phi \in \Phi_V^{\psi} | \psi, \phi \models (\exists U)P\}$. In the next section, *directed formulas* will thus be used in this fashion to represent sets of states and sets of output vectors. We shall also see how MDG can be used to represent relations.

### 2.4.2 Describing ASM with MDGs

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

1. $X$, $Y$ and $Z$ are pairwise disjoint sets of input symbols, state symbols and output symbols, respectively. Let $\eta$ be a one-to-one function that maps each state variable $y$ to a distinct variable $\eta(y)$ obtained, for example, by adorning $y$ with a prime. The variables in $Y' = \eta(Y)$ are used to denote the next-state variables. $X, Y$ and $Z$ must be disjoint from $Y'$.

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

   A variable in $X \cup Y \cup Z$ is called an $ASM\_variable$ [82].

2. $F_I$ is a *directed formula* representing the set of initial states, of type $U \rightarrow Y$, where $U$ is a set of abstract variables disjoint from $X \cup Y \cup Y' \cup Z$. Typically, $F_I$ is a one-disjunct *directed formula* representing the set of initial states.

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

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

3. $F_T$ is a *directed formula* of type $(X \cup Y) \rightarrow Y'$, representing the transition relation.

   Given an interpretation $\psi$, an input vector $\phi \in \Phi_X^\psi$ and a state $\phi' \in \Phi_Y^\psi$, a state $\phi'' \in \Phi_Y^\psi$ is a possible next state iff $\psi, \phi \cup \phi' \cup (\phi'' \circ \eta) \models F_T$. Thus the transition relation of the state machine $M$ represented by $D$ is given by:

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

4. $F_O$ is a *directed formula* of type $(X \cup Y) \rightarrow Z$, representing the output relation.

   Given an interpretation $\psi$, the output relation of the state machine $M$ represented by $D$ is

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

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

To illustrate the above definitions, we give an example.

**Example 2.4.1** *(MinMax) The ASM shown as Figure 2.2 models a simple machine to pick up the minimal and maximal numbers from a set of natural numbers. The circles in the figure correspond to the control states, and the arrows correspond to the control transitions of the machine. The transition labels specify the conditions under which each transition is taken and an assignment of values to the abstract next state variables. The machine has* 2 *input variables* $r, x$ *and* 3 *state variables* $c, m, M$*, where* $r$ *describes the reset signal,* $x$ *represents the input number,* $c$ *stands* *for the states of machine, and $m, M$ stores the minimal or maximal number obtained so far. $r$ and $c$ are defined as variables of concrete sort and $x, m, M$ are defined as variables of abstract sorts wordn. There are no output variables in the machine. A function symbol, leq, is an operator to compare two variables $a$ and $b$ of sort wordn. $leq(a, b) = 1$ if and only if $a$ is less than or equal to $b$. The initial states are $F_I : c = 1 \wedge m = max \wedge M = min$. The transition relation $F_T$ is*

$$(c = 1 \wedge r = 1 \wedge m' = max \wedge M' = min \wedge c' = 1) \bigvee$$
$$(c = 1 \wedge r = 0 \wedge m' = x \wedge M' = x \wedge c' = 0) \bigvee$$
$$(c = 0 \wedge r = 1 \wedge m' = max \wedge M' = min \wedge c' = 1) \bigvee$$
$$(c = 0 \wedge r = 0 \wedge leq(x, m) = 1 \wedge leq(x, M) = 1 \wedge m' = x \wedge M' = M \wedge c' = 0) \bigvee$$
$$(c = 0 \wedge r = 0 \wedge leq(x, m) = 0 \wedge leq(x, M) = 1 \wedge m' = m \wedge M' = M \wedge c' = 0) \bigvee$$
$$(c = 0 \wedge r = 0 \wedge leq(x, m) = 1 \wedge leq(x, M) = 0 \wedge m' = x \wedge M' = x \wedge c' = 0) \bigvee$$
$$(c = 0 \wedge r = 0 \wedge leq(x, m) = 0 \wedge leq(x, M) = 0 \wedge m' = m \wedge M' = x \wedge c' = 0)$$

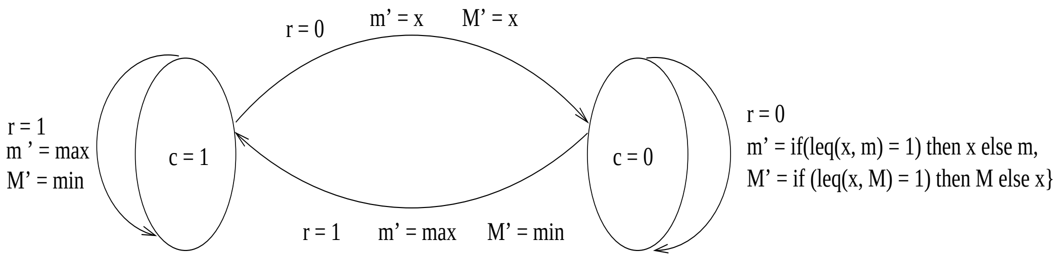



Figure 2.2: The ASM of the Minmax

### 2.4.3 State Exploration and Invariant Checking

Given an abstract state machine description $D = (X, Y, Z, F_I, F_T, F_O)$, we can compute the set of the reachable states of a state machine $M = (\Phi_X^\psi, \Phi_Y^\psi, \Phi_Z^\psi, S_I, R_T, R_O)$ represented by $D$, for any interpretation $\psi$ and $\psi$-compatible assignments $\Phi$, using the MDG algorithms mentioned above. During the computation, the algorithm checks if an invariant condition holds on all the reachable states. The invariant is represented by an MDG $C$ of type $W \to Z$, where $W$ is a set of abstract variables disjoint from $X, Y, Y, Z$ and $U$. For a given interpretation $y$, an output vector is deemed to satisfy the invariant iff $\psi, \phi \models (\exists W)C$; therefore, $Set_Z^{\psi}(C)$ is the set of output vectors that satisfy the invariant.

The Reachability Analysis (ReAn) algorithm [21] can be described by the following pseudo-code:

**Algorithm 2.4.1** *(Reachability Analysis Algorithm)*

1 *ReAn(D,C)*
2 *R:=* $F_I$*; Q =* $F_I$*; K := 0;*
3 *loop*
4 *K:= K+1;*
5 *I :=* Fresh*(X, K);*
6 *O :=* RelP*(*$\{I,\ Q,\ F_O\}, X \cup Y, \emptyset$*);*
7 *P :=* PbyS*(O, C);*
8 *if P* $\neq$ F *then return failure;*
9 *N :=* RelP*(*$\{I,\ Q,\ F_T\}, X \cup Y,\ Y' \to Y$*);*
10 *Q :=* PbyS*(N, R);*
11 *if Q =* F *then return success;*
12 *R :=* PbyS*(R, Q);*
13 *R :=* Disj*(R, Q);*
14 *end loop;*
15 *end ReAn;*

The variables $I, N, P, Q$ and $R$ represent sets of states, and $O$ represents a set of output vectors. Before each iteration, $R$ contains the states reached so far, while $Q$ is the frontier set, i.e., a subset of $\Phi_Y^{\psi}(R)$ containing at least all those states that entered $\Phi_Y^{\psi}(R)$ for the first time in the previous iteration. In line 5, $Fresh(X, K)$ constructs a one-disjunct *directed formula* representing a conjunction of equation $x = u$, one for each abstract input variable $x \in X$, where $u$ is a fresh variable from the set of auxiliary abstract variables $U$. The value of the loop counter $K$ is used to generate the fresh variables. This one-disjunct *directed formula* is assigned to $I$, which represents the set of input vectors. In line 6, the relation product (RelP) operation is used to compute the *directed formula* representing the set of output vectors produced by the states in the frontier set. The resulting *directed formula* is assigned to $O$, Then, in line 7, the pruning-by-subsumption (PbyS) operation is used to remove from $O$ those disjuncts that represent output vectors which satisfy the invariable $C$. The resulting *directed formula* is assigned to $P$. In line 8, if $P$ is not F, then the procedure stops and reports failure. If $P$ is F, then every output vector produced by a state in the frontier set satisfies the invariant and the verification procedure continues. In line 9, the relational product operation is used again; this time compute the *directed formula* representing the set of states that can be reached in one step from the frontier set of states. Note that the *directed formula* $Q$ represents the frontier set. Lines 10 and 11 check whether $\Phi_Y^\psi(N) \subseteq \Phi_Y^\psi(R)$ by the same method used in lines 7 and 8, respectively. If it is the case, then every state reachable from the frontier set was already in $\Phi_Y^\psi(R)$. The fixpoint has been reached and $R$ represents all the reachable states. Therefore, the procedure terminates and reports success. Otherwise, the *directed formula* assigned to $Q$ in line 10 represents the new frontier set. Line 12 simplifies $R$ by removing it any disjuncts that are subsumed by $Q$, using PbyS. There may be such disjuncts because $Q$ was not computed earlier as an exact difference. Line 13 then computes the new value of $R$ by taking the disjunction of $R$ and $Q$ which represents the set of states $\Phi_Y^\psi(R) \cup \Phi_Y^\psi(Q)$ and assigns it to $R$.

## 2.5 MDG Verification Applications

At present, there exists several MDG-based hardware verification applications, which constitute an MDG tool set. Figure 2.3 depicts the components. The MDG tools accept a Prolog-styled Hardware Description Language (HDL), called MDG-HDL [85] which allows the use of abstract variables and uninterpreted function symbols. MDG-HDL supports structural descriptions, behavioral descriptions, or the mixture of structural descriptions and behavioral descriptions. A structural description is usually a netlist of components (predefined in MDG-HDL) connected by signals. A behavioral description is given by a tabular representation of the transition/output relation or truth table [83].

Besides circuit descriptions, a variety of information, such as sort and function type definitions, symbol ordering and invariant specification, etc., has to be provided in order to use the applications. All of these are organized into four kinds of input files: that algebraic file, the symbol order file, the circuit description file, and the invariant specification file. All these files are compiled into internal MDG data structures by the MDG-HDL compiler. The MDG package supplies the MDG operation algorithms described above. The reachability procedure implements the ReAn algorithm and printing facility provides various printing options. On top of them are built a set of verification application including combinational equivalence checking, safety property invariant checking, sequential equivalence checking, and model checking.

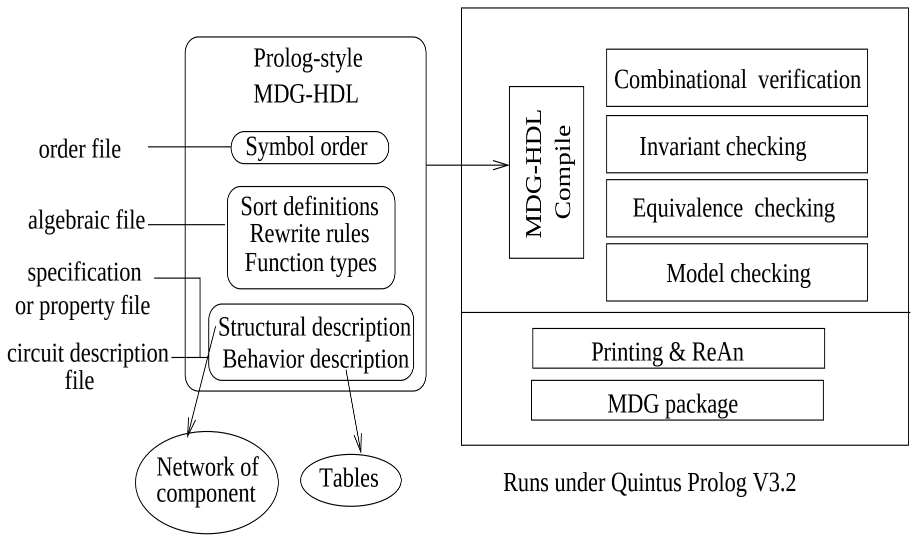



Figure 2.3: The MDG tool set

**Combinational equivalence checking**. Given two combinational circuits, an MDG is computed for each of them to represent its input-output relation by combining the MDGs of the components of the circuit using the relational product operations. Because of the canonicity of MDG, comparing the functionality of two combinational circuits reduces to computing the MDGs representing their input/output relations. If the two circuits have the same functionality, the two MDGs must represent logically equivalent formulas, and hence they must be isomorphic.

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

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

**Model checking**. Model checking algorithms for a subset of Abstract-CTL* called $\mathcal{L}_{MDG}$ was developed by Xu [81]. It can verify both safety and liveness properties. To check a property $p$ in $\mathcal{L}_{MDG}$ on an ASM $M$, additional ASMs $M_j$ are first built for basic sub-formulas of $p$ in which only the temporal operator $X$ is allowed (called Next_let_formulas), and then these additional ASMs are composed with $M$. Finally, appropriate algorithms are applied to verify the transformed simplified properties on the composite machine. We will describe the details on $\mathcal{L}_{MDG}$ in Chapter 3.

## 2.6 Conclusion

This chapter reviewed the basic concepts of Multiway Decision Graphs (MDGs), which underlying logic is a many-sorted first-order logic and described by the well-formed first-order formulas (*directed formulas*). We reviewed the MDG data structure and basic operators. We also described the implicit abstract state enumeration procedure and various existing MDG based verification techniques.

# Chapter 3

# Model Checking and $\omega$-Automata

Model checking is a technique to algorithmically check whether a design model satisfies its specification. The model is represented by a Kripke structure $M$, and its specification is described as a formula $p$ in some temporal logic. The satisfaction problem can be expressed mathematically as the decision of $M \models p$. The symbolic model checking sets a new milestone in the development process of model checking techniques. Symbolic model checking encodes the sets and the transition relations with Reduced Ordered Binary Decision Diagrams (ROBDDs), which are canonical representation of the Boolean characteristic function once an order on the variables has been established. The implicit enumeration technique greatly increases the state spaces that can be accepted by model checking [5].

In this chapter, we first introduce the notion of Kripke structure and its symbolic representation, and also give the definition of bisimulation relation over two Kripke structures. Then we give the different logics (propositional, first-order, and temporal logics) for describing properties. Finally, we focus on the $\omega$-automata based model checking method, which we intend to lift from propositional to first-order logic.

## 3.1 Kripke Structure

Model checking describes system design as a Kripke Structure. We will introduce the definition and give the bisimulation relation over Kripke Structures in this section.

### 3.1.1 ROBDD Representation

For model checking, the system design is as a Kripke structure [54], which can be viewed as a labeled Finite State Machine (FSM).

**Definition 3.1.1** *Let $AP$ be a set of atomic propositions. A Kripke structure $M$ over $AP$ is a four-tuple $M = (S, S_0, R, L)$, where*

- *$S$ is the set of states,*
- *$S_0 \subseteq S$ is the set of initial states,*
- *$R \subseteq S \times S$ is the transition relation, which must be total, i.e., for every state $s \in S$, there is a state $s' \in S$ such that $R(s, s')$, we write $s \to s'$ for clarity,*
- *$L : S \to \mathcal{P}(AP)$ is a function that labels each state with the set of atomic propositions true in that state.*

A *path* (computation) in $M$ starting from a state $s$ is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$, such that $s_0 = s$ and for every $i \geq 0$, $s_i \to s_{i+1}$. The suffix of $\pi$ from state $s_i$ is denoted $\pi^i$.

A symbolic representation of the Kripke structure is implemented by describing the state set and transition relation with Boolean characteristic functions. Let $AP = \{p_1, \cdots, p_n\}$. A state $s$ is represented by a vector of Boolean variables $X = \{x_1, \ldots, x_n\}$, where $x_i = p_i$ or $x_i =\sim p_i$ (the negation of $p_i$). A set of states can be represented by a Boolean characteristic function $Q = \{X | f(x_1, \ldots, x_n) = 1\}$. Similarly, the relation $R \subseteq S \times S$ can be represented by a Boolean characteristic function of two sets of variables $R = \{(X, X') | f(x_1, \ldots, x_n, x'_1, \ldots, x'_n) = 1\}$.

To illustrate the above definitions, we give an example.

**Example 3.1.1** *A modulo-4 counter counts* $0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, \cdots$. *Initially the counter is zero. Let $AP = \{Hi, Low\}$ be the set of two propositional variables to encode the four states. The Kripke structure graph and its corresponding symbolic representation are show in Figure 3.1.*

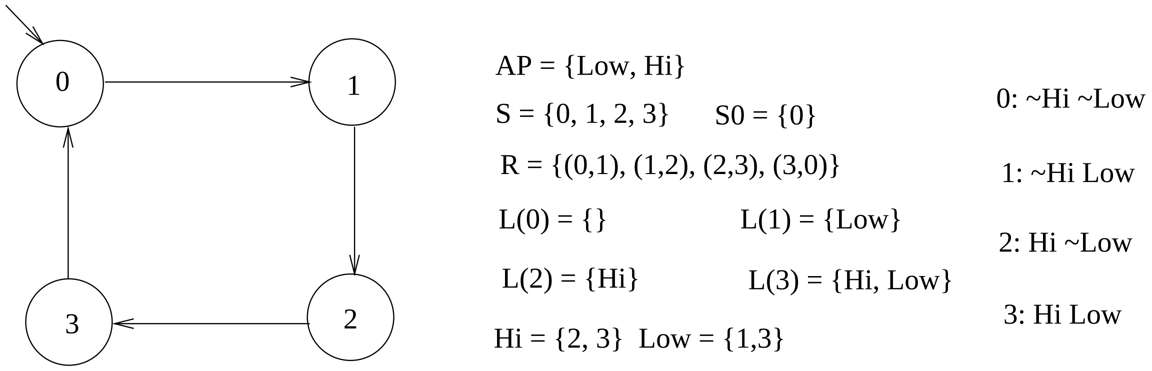



Figure 3.1: The Kripke structure of a modulo-4 counter

### 3.1.2 Bisimulation Relation

Let $AP$ be a set of atomic propositions and let $M_1 = (S_1, S_{01}, R_1, L_1)$ and $M_2 = (S_2, S_{02}, R_2, L_2)$ be two Kripke structures. The *bisimulation relation* defined in [68] is described as below:

**Definition 3.1.2** *(Bisimulation) A relation $H \subseteq S_1 \times S_2$ is said to be a bisimulation relation over $M_1$ and $M_2$ if the following conditions hold:*

1. *For every $s_1 \in S_{01}$, there is $s_2 \in S_{02}$ such that $H(s_1, s_2)$. Moreover, for every $s_2 \in S_{02}$, there is $s_1 \in S_{01}$ such that $H(s_1, s_2)$.*

2. *For every $(s_1, s_2) \in H$*

   - $L_1(s_1) = L_2(s_2)$ *and*
   - $\forall t_1\ [R_1(s_1, t_1) \rightarrow\ \exists t_2\ [R_2(s_2, t_2)\ \wedge\ H(t_1, t_2)]]$
   - $\forall t_2\ [R_2(s_2, t_2) \rightarrow\ \exists t_1\ [R_1(s_1, t_1)\ \wedge\ H(t_1, t_2)]]$.

   *We write $s_1 \equiv s_2$ for $H(s_1, s_2)$. We call $M_1$ and $M_2$ are bisimular (denote $M_1 \equiv M_2$) if there exists a bisimulation relation $H$ over $M_1$ and $M_2$.*

To illustrate this concept, we give an example shown in Figure 3.2, where the relation $H$ is shown in the bottom.

Bisimulation relation produces both correct positives and correct negatives, which is shown in the following theorem.

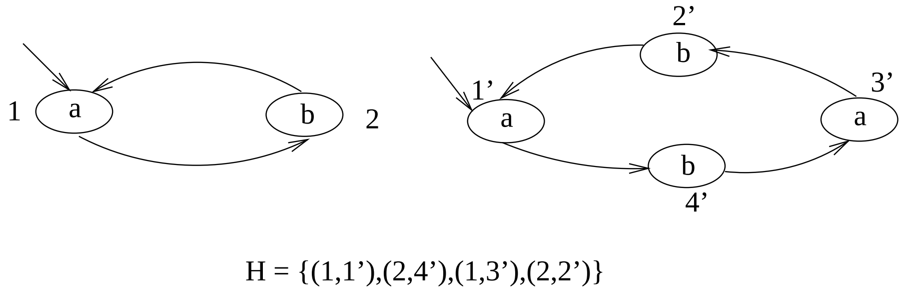



Figure 3.2: An example of the bisimulation relation

**Theorem 3.1.1** *(see [14]) Let $M_1$ and $M_2$ be two Kripke models. If $M_1 \equiv M_2$, then for every $CTL^*$ formula $f$ (with atomic propositions in $AP$), $M_2 \models f$ if and only if $M_1 \models f$.*

## 3.2 $\omega$-Automata Theory

Model checking using $\omega$-automata theory transfers both the system and property into $\omega$-automata, thus reduces the checking problem into a language containment problem. Automaton can be viewed as a transition system structure with an acceptance condition. $\omega$-automata deviate from the traditional finite automaton by interpretation of the acceptance conditions: there are no final states; instead, acceptance is determined with respect to the set of states that are visited infinitely often. Different types of acceptance conditions are studied [76]. In the thesis, we mainly focus on Büchi conditions.

**Definition 3.2.1** *A Büchi automaton $\mathcal{B} = (Q, I, \delta, F)$ over an alphabet $\Sigma$ is given by a finite set $Q$ of states, a non-empty set $I \subseteq Q$ of initial states, a transition relation $\delta \subseteq Q \times \Sigma \times Q$, and a set $F \subseteq Q$ of acceptance states. A run $r$ of $\mathcal{B}$ over an $\omega-$word $\omega = a_0 a_1 \ldots \in \Sigma^\omega$ is an infinite sequence $r = q_0 q_1 \ldots$ of states $q_i \in Q$ such that $q_0 \in I$ and $(q_i, a_i, q_{i+1}) \in \delta$ hold for all $i \geq 0$. The run $r$ is accepted iff there exists some $q \in F$ such that $q_i = q$ holds for infinitely many times.*

The language $\mathcal{L}(\mathcal{B}) \subseteq \Sigma^\omega$ is the set of $\omega$-words, for which there exists some accepted run $r$ of $\mathcal{B}$. A language $\mathcal{L} \subseteq \Sigma^\omega$ iff $\mathcal{L} = \mathcal{L}(\mathcal{B})$ for some Büchi automaton $\mathcal{B}$.

The following are some properties of Büchi automaton that establish the foundation of application in the $\omega$-automata based model checking [76, 79].

**Proposition 3.2.1** *For every LTL formula $\varphi$ of length $n$, there exists a Büchi automaton $\mathcal{B}_\varphi = (Q, I, \delta, F)$ with $2^{O(n)}$ states, which accepts precisely $\omega-$words that satisfy $\varphi$.*

**Proposition 3.2.2** *For a Büchi automaton $\mathcal{B}$ with $n$ states over alphabet $\sigma$, there is a Büchi automaton $\bar{\mathcal{B}}$ with $2^{O(nlogn)}$ states such that $\mathcal{L}(\overline{\mathcal{B}}) = \Sigma_\omega \setminus \mathcal{L}(\mathcal{B})$.*

**Proposition 3.2.3** *For a Büchi automaton $\mathcal{B}$ with $n$ states, it is decidable in time $O(n)$ whether $\mathcal{L}(\mathcal{B}) = \emptyset$ or not.*

The Büchi automaton type we considered is *generalized Büchi automaton* $\mathcal{B}$. The acceptance condition of $\mathcal{B}$ is defined by a finite set $\mathcal{F} = \{F_1, \ldots, F_n\}$ of sets of states. A run is accepted if some states from each $F_i$ are visited infinitely often. Note that in the special case $\mathcal{F} = \emptyset$ all infinite runs of the *generalized Büchi automaton* $\mathcal{B}$ are accepted.

**Proposition 3.2.4** *Let $\mathcal{A}$ be a generalized Büchi automaton $(Q, Q_0, \delta, \mathcal{F})$ over the alphabet set $\Sigma$, where $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$. We can construct a Büchi automaton $\mathcal{B}$ that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$.*

The construction is based on the sets in $\mathcal{F}$, which is given as follows:

- $\mathcal{F} = \emptyset$ : $\mathcal{B} = (Q, Q_0, \delta, Q)$,
- $\mathcal{F} = \{F_1\}$ : $\mathcal{B} = (Q, Q_0, \delta, F_1)$,
- $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$, where $n \geq 2$ : $\mathcal{B} = (Q', Q'_0, \delta', F'_1)$, where
  - $Q' = Q \times \{1, 2, \ldots, n\}$,
  - $Q'_0 = Q_0 \times \{1\}$,

– $\delta'$ is defined as follows: $(q', j) \in \delta((s, i), a)$ iff $s' \in \delta(s, a)$ and $((s \notin F_i$ and $j = i) \vee ((s \in F_i)$ and $j = (i \bmod n) + 1))$, and

– $F'_1 = F_1 \times \{1\}$.

Note that the automaton for the system design $\mathcal{B}_M$ should accept all paths produced by $M$, and a *generalized Büchi automaton* with $\mathcal{F} = \emptyset$ accepts all infinite runs of $\mathcal{B}$. In particular, we assume the fairness conditions to be expressed as part of the property formula. Thus, the automaton for the system design is a *generalized Büchi automaton* without any acceptance condition. For simplicity, we will use $M$ to represent $\mathcal{B}_M$ in the sequel.

## 3.3 Property Specification Language

Model checking checks the satisfaction of a Kripke structure with respect to properties given in some kinds of temporal logics, which we describe next.

### 3.3.1 Propositional Logic

Successful symbolic model checking is based on the propositional logic. A propositional logic [59] consists of a set of propositions $AP = \{p, q, \ldots\}$ and Boolean connectives $\wedge$, $\vee$, and $\neg$, representing *and, or* and *not*, respectively. The formulas are composed of $AP$ and Boolean connectives. The symbols $\rightarrow$ (implication) and $\leftrightarrow$ (equivalence) can be interpreted as follows: $p \rightarrow q$ abbreviates $\neg p \vee q$ and $p \leftrightarrow q$ abbreviates $p \rightarrow q$ *and* $q \rightarrow p$, respectively.

The semantics of the proposition formulas can be induced with following rules.

- $\neg p$ is *True* iff $p$ is *False*.
- $p \vee q$ is *True* iff $p$ is *True* or $q$ is *True*.
- $p \wedge q$ is *True* iff both $p$ and $q$ are *True*.
- $p \rightarrow q$ is *True* iff $p$ is *True* then $q$ is *True*.
- $p \leftrightarrow q$ is *True* iff both $p \rightarrow q$ is *True* and $q \rightarrow p$ is *True*.

### 3.3.2 First Order Logic

The developed model checking technique in this thesis is based on the first-order logic. A first-order logic [30] language $\mathcal{L}$ consists of a set of signature symbols (countable sets of symbols for constant, functions, predicates, and variables), a set of standard Boolean connectives and quantifiers.

**Symbols of $\mathcal{L}$** are composed of a set of function symbols, a set of predicate symbols and a set of individual variables. The 0-ary function symbols comprise the subset of constant symbols; and the 0-ary predicate symbols are known as the propositional symbols. The following notations are used to represent the above symbols:

1. $\phi, \varphi, \ldots,$ etc. for n-ary, $n \geq 1$, predicate symbols,
2. $P, Q, \ldots,$ etc. for propositional symbols,
3. $f, g, \ldots,$ etc. for n-ary, $n \geq 1$, function symbols,
4. $c, d, \ldots,$ etc. for constant symbols, and
5. $y, z, \ldots,$ etc. for variable symbols.
6. Binary predicate symbols (the equality symbols) $\approx$ is based in the standard infix fashion.
7. Quantifier symbols $\forall$ and $\exists$, denoting universal and existential quantification, are applied to individual variable symbols according to the usual rules regarding scope of quantifiers.

**Syntax of $\mathcal{L}$** The terms of $\mathcal{L}$ are defined inductively by the following rules:

1. Each constant $c$ is a term.
2. Each variable $y$ is a term.
3. If $t_1, \ldots, t_n$ are terms and $f$ is an $n-$ary function symbol, then $f(t_1, \ldots, t_n)$ is a term.

The atomic formulas of $\mathcal{L}$ are defined by the following rules:

1. Each 0-ary predicate symbol (i.e., atomic proposition) is an atomic formula.
2. If $t_1, \ldots, t_n$ are terms and $\phi$ is an $n-$ary predicate, then $\phi(t_1, \ldots, t_n)$ is an atomic formula.
3. If $t_1$ and $t_2$ are terms, then $t1 \approx t_2$ is also an atomic formula.

Finally, the (compound) formulas of $\mathcal{L}$ are defined inductively as follows:

1. Each atomic formula is a formula.
2. If $p, q$ are formulas then $(p \wedge q)$, $\neg p$ are formulas.
3. If $p$ is a formula and $y$ is a free variable in $p$, then $\exists\ y\ p$ is a formula.

**Semantics of $\mathcal{L}$.** The semantics of $\mathcal{L}$ is provided by an interpretation $I$ over some domain $D$. $B$ represents the Boolean domain, which is a domain $D$ with *true* and *false* values. The interpretation $I$ assigns an appropriate meaning over $D$ to the (non-logic) symbols $\mathcal{L}$ as follows:

- For an n-ary predicate symbol $\psi$, $n \geq 1$, the meaning $I(\psi)$ is a function $D^n \rightarrow B$.
- For a proposition symbol $P$, the meaning $I(P)$ is an element of $B$.
- For an n-ary function symbol $f, n \geq 1$, the meaning $I(f)$ is a function $D^n \rightarrow D$.
- For an individual constant symbol $c$, the meaning $I(c)$ is an element of $D$.
- For an individual variable symbol $y$, the meaning $I(y)$ is an element of $D$.

The interpretation $I$ is extended to arbitrary terms inductively:

$$I(f(t_1, \ldots, t_n)) = I(f)(I(t_1), \ldots, I(t_n))$$

The truth meaning of a formula $P$ under interpretation $I$, written $I \models P$, is defined as follows:

- $I \models P$, where $P$ is an atomic proposition, iff $I(P) = true$.

- $I \models \psi(t_1, \ldots, t_n)$, where $\psi$ is an n-ary predicate and $t_1, \ldots, t_n$ are terms, $I(\psi)(I(t_1), \ldots, I(t_n)) = true$.
- $I \models t_1 \approx t_2$ iff $I(t_1) = I(t_2)$
- $I \models p \wedge q$ iff $I \models p$ and $I \models q$.
- $I \models \neg p$ iff not the case the $I \models p$.
- $I \models \exists y\ p$, where $y$ is a free variable in $p$, iff there exists some $d \in D$ such that $I[y \leftarrow d] \models P$, where $I[y \leftarrow d]$ is the interpretation identical to $I$ except that $y$ is assigned a value $d$.

### 3.3.3 Temporal Logics

Properties of model checking are described as some kinds of temporal logics. A temporal logic [30] is a formalism for describing sequences of transitions between states in a reactive system. It provides a formal system for qualitatively describing and reasoning about how the truth values of assertions change over time. There are four basic operators in temporal logic:

- $\mathsf{G}P$ ("always $P$", also read as "henceforth $P$") is true in state $s$ if $P$ is true in all future states from $s$ (including $s$).
- $\mathsf{F}P$ ("sometimes $P$", also read as "eventually $P$") is true in state $s$ if $P$ is true in some future states from $s$.
- $\mathsf{X}P$ ("nexttime $P$") is true in state $s$ if $P$ is true in the next state from $s$.
- $P\ \mathsf{U}\ Q$ ("$P$ until $Q$") 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.

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

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

- *Liveness properties* - assert that eventually something "good" happens, typically represented as $\models P \rightarrow \mathsf{F}Q$, i.e., in all models, if $P$ is initially true, then $Q$ will eventually be true;

- *Precedence properties* - assert the precedence order of events, typically represented as $\models P \;\mathsf{U}\; Q$, i.e., in all models, $P$ will hold until $Q$ becomes true.

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

**Propositional Linear Temporal Logic**

In a Propositional Linear Time Temporal Logic (PLTL) the underlying structure of time is assumed to be isomorphic to the natural numbers with their usual order $(N, <)$ [30]. Let $AP$ be an underlying set of atomic proposition symbols. A linear-time structure $M = (S, x, L)$ is defined such that $S$ is a set of states, $x : N \rightarrow S$ is an infinite sequence of states, and $L : S \rightarrow 2^{AP}$ is a labeling of each state with the set of atomic propositions in $AP$ that are true in the state.

Usually, the notation $x = (s0, s1, s2, \cdots) = (x(0), x(1), x(2), \cdots)$ is employed to denote the timeline $x$, which is also referred to as a full path, or a computation sequence, or a computation.

The basic temporal operators of PLTL are $\mathsf{F}$ , $\mathsf{G}$ , $\mathsf{X}$, and $\mathsf{U}$. The syntax of PLTL formulas is generated by the following rules:

- Each atomic proposition $P$ is a formula;

- If $p$ and $q$ are formulas, then $p \;\wedge\; q$ and $\neg p$ are formulas;

- If $p$ and $q$ are formulas, then $p \;\mathsf{U}\; q$ and $\mathsf{X}\; p$ are formulas.

The other formulas can be introduced as abbreviations in the usual way: For the propositional connectives, $p \vee q$ abbreviates $\neg(\neg p \wedge \neg q)$; $p \;\rightarrow\; q$ abbreviates $\neg p \;\vee\; q$;

$p \leftrightarrow q$ abbreviates $(p \rightarrow q) \wedge (q \rightarrow p)$; $p \mathsf{R} q$ (release, the dual of $\mathsf{U}$) abbreviates $\neg(\neg p \mathsf{U} \neg q)$. The Boolean constant *true* abbreviates $p \vee \neg p$, while **false** abbreviates $\neg true$. Then the temporal connective $\mathsf{F}\, p$ abbreviates $true \mathsf{U} p$, and $Gp$ abbreviates $\neg\mathsf{F}\neg p$.

**Semantics.** The semantics of a formula $p$ of PLTL with respect to a linear-time structure $M = (S, x, L)$ is defined as follows: We write $M, x \models p$ to mean that "in structure $M$, formula $p$ is true on time line $x$", $x^i$ denotes the suffix path $s_i, s_{i+1}, s_{i+2}, \ldots$. Although it is not explicitly stated, those PLTL properties are checked on all paths.

1. $M, x \models p$ iff $p \in L(s_0)$, for atomic proposition $p$;
2. $M, x \models \neg p$ iff not $M, x \models p$;
3. $M, x \models p \wedge q$ iff $M, x \models p$ and $M, x \models q$;
4. $M, x \models \mathsf{X}p$ iff $x^1 \models p$;
5. $M, x \models (p\mathsf{U}q)$ iff $\exists j (x^j \models q$ and $\forall k < j (x^k \models p))$;
6. $M, x \models (p\mathsf{R}q)$ iff $\forall j (x^j \models q$ or for some $k < j (x^k \models p))$;
7. $M, x \models \mathsf{F}p$ iff $\exists j (x^j \models p)$;
8. $M, x \models \mathsf{G}p$ iff $\forall j (x^j \models p)$

The duality between the linear temporal operators is illustrated by the following assertions:

- $\models \mathsf{G}\neg p \equiv \neg\mathsf{F}p$;
- $\models \mathsf{F}\neg p \equiv \neg\mathsf{G}p$;
- $\models \mathsf{X}\neg p \equiv \neg\mathsf{X}p$;

We say that a PLTL formula $p$ is satisfiable if there exists a linear-time structure $M = (S, x, L)$ such that $M, x \models p$, and any such structure defines a model of $p$.

**Computation Tree Logic**

Different kinds of Branching Time Temporal Logic (BTTL) have been proposed, depending on the exact set of operators allowed. Their common feature is that they are interpreted over branching tree-like time structures, where each moment may have many successor moments. The structure of time corresponds to an infinite tree. The usual temporal operators $(\mathsf{F}, \mathsf{G}, \mathsf{X}, \mathsf{U})$ are regarded as state quantifiers. Additional quantifier called the path quantifier is provided to represent all path $(\mathsf{A})$ and some path $(\mathsf{E})$ from a given state. Here we only describe the Computation Tree Logic (CTL), a restricted form of BTTL. $CTL^*$ extends CTL by allowing those operators composed by a path quantifier followed by an arbitrary linear temporal operator and propositional combinations and nesting of linear temporal operators. CTL* is sometimes informally referred to as a full branching time logic.

CTL severely restricts the types of formulas that can appear after a path quantifier — only single linear time operator $\mathsf{F}, \mathsf{G}, \mathsf{X}$, or $\mathsf{U}$ can follow a path quantifier and time operators cannot be combined directly with propositional connectives. The syntax of CTL is:

- Every atomic proposition is a CTL formula.
- If $p$ and $q$ are CTL formulas, then so are $\neg p$, $(p \wedge q)$, $\mathsf{AX}p$, $\mathsf{EX}p$, $\mathsf{A}(p\mathsf{U}q)$, $\mathsf{E}(p\mathsf{U}q)$.

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

- $p \vee q = \neg(\neg p \wedge \neg q)$.
- $\mathsf{AF}p = \mathsf{A}(true\mathsf{U}p)$.
- $\mathsf{EF}p = \mathsf{E}(true\mathsf{U}p)$.
- $\mathsf{AG}p = \neg\mathsf{E}(true\mathsf{U}\neg p)$.
- $\mathsf{EG}p = \neg A(true\mathsf{U}\neg p)$.

Because all operators are prefixed by $\mathsf{A}$ or $\mathsf{E}$, the truth or falsehood of a formula depends only on the given state $s$, and not on the particular branch. The semantics of CTL is defined on a Kripke structure $M$. Given an $M = (S, S_0, R, L)$ and an initial state $s_0$, an infinite computation tree $T$ is generated with a root at $s_0$, and expanded with all possible nondeterministic transitions at every state. The truth of a CTL formula is defined on $T$ inductively as follows:

- $(M, s_0) \models p$ iff $p \in P(s_0)$, where $p$ is an atomic proposition.
- $(M, s_0) \models \neg p$ iff not $(M, s_0) \models p$.
- $(M, s_0) \models p \wedge q$ iff $(M, s_0) \models p$ and $(M, s_0) \models q$.
- $(M, s_0) \models \mathsf{AX}p$ iff for all states $t$ such that $(s_0, t) \in R$, $(M, t) \models p$.
- $(M, s_0) \models \mathsf{EX}p$ iff for some states $t$ such that $(s_0, t) \in R$, $(M, t) \models p$.
- $(M, s_0) \models \mathsf{A}(p\mathsf{U}q)$ iff for all paths $(s_0, s_1, s_2, \ldots)$, $\exists k \geq 0$ such that $(M, s_k) \models q$, and $\forall i, 0 < i < k, (M, s_i) \models p$.
- $(M, s_0) \models \mathsf{E}(p\mathsf{U}q)$ iff for some paths $(s_0, s_1, s_2, \ldots), \exists k \geq 0$ such that $(M, s_k) \models q$, and $\forall i, 0 < i < k, (M, s_i) \models p$.

**PLTL versus PBTTL.** In linear time logics, temporal operators are provided for describing events along a single future time line. Although when a linear formula is used for specification, there is usually an implicit universal quantification over all possible futures. In contrast, in branching time logics the operators usually reflect the branching nature of time by allowing explicit quantification over possible futures. One argument presented by the supporters of branching time logic is that it offers the ability to reason about existential properties in addition to universal properties [42].

### The Propositional $\mu-$Calculus

Symbolic computations, operating on the sets of states, are conveniently described by formulas of $\mu$-calculus. The $\mu-$calculus [27] is obtained from the first-order predicate logic by adding the least ($\mu$) and the greatest ($\nu$) fixpoint operators. Given a set of atomic propositions $A$ and a set of variables $\mathcal{V}$, we use the following syntax, which corresponds to the propositional $\mu-$Calculus.

- If $f \in A \cup \mathcal{V}$, then $f$ is a formula;
- if $f \in A$, then $\neg f$ is a formula;
- if $f$ and $g$ are formulas, so are $f \wedge g$, $EX\ f$, $EY\ f$, $AX\ f$ and $AY\ f$;
- if $Z \in \mathcal{V}$ and $f$ is a formula, then $\nu Z.\ f$ and $\mu Z.\ f$ are formulas.

The semantics of $\mu-$calculus are defined with respect to a labeled transition system $M = (S, I, R, L)$, where $S$ is a set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation over $S$, and $L : S \to 2^A$.

Let $\varepsilon = \{e : \mathcal{V} \to 2^S\}$ be the set of environments, that is , the set of functions that associate a set of states to each variable. Let $\Phi$ be the set of $\mu-$calculus formulas over $A$ and $\mathcal{V}$. The function $\mathcal{I} : \Phi \times \varepsilon \to 2^S$ associates a set of states to a formula $\phi \in \Phi$ in a given $e \in \varepsilon$. Let $e[S'/Z]$ be the environment that coincides with $e$, except that $e[S'/Z](Z) = S'$. The function $\mathcal{I}$ extends $\varepsilon$ in a natural way, where

$$\begin{aligned}
\mathcal{I}(EX\ f, e) &= \{s \in S : \exists (s, s') \in R, s' \in \mathcal{I}(f, e)\},\\
\mathcal{I}(AX\ f, e) &= \{s \in S : \forall (s, s') \in R, s' \in \mathcal{I}(f, e)\},\\
\mathcal{I}(EY\ f, e) &= \{s \in S : \exists (s', s) \in R, s' \in \mathcal{I}(f, e)\},\\
\mathcal{I}(AY\ f, e) &= \{s \in S : \forall (s', s) \in R, s' \in \mathcal{I}(f, e)\},\\
\mathcal{I}(\mu Z.\ f, e) &= \bigcap \{S' \subseteq S : S' \supseteq \mathcal{I}(f, e[S'/Z]\}, \text{and}\\
\mathcal{I}(\nu Z.\ f, e) &= \bigcup \{S' \subseteq S : S' \subseteq \mathcal{I}(f, e[S'/Z]\}.
\end{aligned}$$

The following abbreviations correspond to the future and past tense CTL operators:

$$\begin{aligned}
E\ p \mathsf{U}\ q = \mu Z.\ q\ \vee\ (p\ \wedge\ EX\ Z), &\quad E\ p\ S\ q = \mu Z.\ q \vee\ (p\ \wedge\ EY\ Z)\\
EG\ p = \nu Z.\ p\ \wedge\ EX\ Z, &\quad EH\ p = \nu Z.p\ \wedge EY\ Z\\
EF\ p = E\ \text{true}\mathsf{U}\ p, &\quad EP\ p = E\ \ \text{true}\ S\ p,
\end{aligned}$$

The $\mathsf{EX}$ (or preimage) operator maps a set of states to the set of their direct predecessors. This corresponds to one step of backward symbolic breadth-first search. Similarly, $\mathsf{EY}$ (image or a forward step) maps a set of states to the set of all their direct successors. The operators $\mathsf{EX}$, $\mathsf{AX}$, and all the abbreviations defined in terms of them are called future-tense or backward operators. $\mathsf{EY}$, $\mathsf{AY}$, and all the abbreviations defined in terms of them are called past-tense or forward operators.

**The First Order Temporal Logic**

The First Order (FO) temporal logic is achieved by refining the propositions of the proposition TL with the symbols of first-order language. The symbols, including constants, variables, functions and predicates, are divided into two classes: the class of global symbols and the class of local symbols. Each global symbol has the same interpretation over all states; the interpretation of a local symbol may vary, depending on the state at which it is evaluated. All function symbols and all $n$-ary predicate symbols, for $n \geq 1$, are global. Propositional (0-ary predicates) symbols and variable symbols may be local or may be global.

FOTL can be classified into FO Linear time TL (FOLTL) and FO Branching time TL (FOBTL). The terms of FOLTL are composed by constants, variables, functions and temporal operator $\mathsf{X}$. The atomic formulas consist of predicates of the terms and equation relations between the terms. The compound formulas are composed with Boolean connective $\wedge$, $\neg$ and temporal operators $\mathsf{U}$ and $\mathsf{X}$.

The semantics of FOLTL is provided by a first-order linear time structure $M$ over a domain $D$. $M = \{S, x, L\}$, where $S$ is a set of states, $x : N \to S$ is an infinite sequence of states, and $L : S \to 2^{\mathcal{L}}$ associates with each state $s$ an interpretation $\mathcal{L}(s)$ of all the symbols at $s$. Global interpretation $I$ of $M$ assigns a meaning to each global symbol, while the local interpretation $L(-)$ associated with $M$ assigns a meaning to each local symbol. A formula $p$ of FOLTL is valid if and only if for every first-order linear time structure $M = (S, x, L)$ we have $M, x \models p$. The formula $p$ is satisfiable iff there exists $M = (S, x, L)$ such that $M, x \models p$.

The FOBTL is obtained by combining the rules for generating a system of propositional Branching Temporal logic plus a first-order language. The underlying structure is extended so that it associates with each state $s$ an interpretation of local and global symbols including in particular local variables as well as local atomic propositions. The semantics is given by the usual definition of truth.

**Abstract_CTL$^*$**

*Abstract_CTL$^*$* is a *partial* first-order branching temporal logic defined in [82], and a subset is used to represent the properties in the MDG regular model checking. A partial interpretation FOTL assumes a specific domain for each variable, but leaves the function of symbols uninterpreted. The syntax and semantics of *Abstract_CTL$^*$* are defined as following [82].

**Syntax of Abstract_$CTL^*$.** Given an abstract description of an ASM $D = (X, Y, Z, F_I, F_T.F_O)$ and a set of *ordinary variables* which are available for use in the specification of the property, the syntax of an abstract_$CTL^*$ can be defined as follows.

(s1) If $t_1$ is an ASM_variable, $t_2$ is an ASM_variable, or a constant, or an ordinary variable, then equation $t_1 = t_2$ is a state formula.

(s2) If $p, q$ are state formulas, then so are $!p$ (not p), $p\&q$ (p and q), $p|q$ (p or q), $p \rightarrow q$.

(s3) If $t$ is an ASM_variable, $v$ is an *ordinary variable*, and $p$ is a state formula, then $\mathsf{LET}\ (v = t)\ \mathsf{IN}\, p$ is a state formula.

(s4) If $p$ is a path formula, then $\mathsf{A}p$ and $\mathsf{E}p$ are state formulas.

The *path formulas* are defined as follows:

(p1) Each state formula is also a path formula.

(p2) If $p$, $q$ are path formulas then so are $!p$ (not p), $p\&q$ (p and q), $p|q$ (p or q), $p \rightarrow q$, $\mathsf{X}p$, $\mathsf{G}p$, $\mathsf{F}p$ and $p\mathsf{U}q$.

(p3) If $t$ is an ASM_variable, $v$ is an *ordinary variable*, and $p$ is a path formula, then $\mathsf{LET}\,(v = t)\ \mathsf{IN}\,p$ is a path formula.

**Semantics of Abstract_$CTL^*$.** Given an interpretation $\psi$ and $\psi-$compatible assignment $\phi$, the semantics of $Abstract\_CTL^*$ is defined on the infinite computation tree $T$ expanded by $M = (S, R)$ descried by $D = \{X, Y, Z, F_I, F_T, F_O\}$, where

$$S = \{s = (\phi, \phi', \phi") \in \Phi_X^{\psi} \times \Phi_Y^{\psi} \times \Phi_Z^{\psi} | \psi, \phi \cup \phi' \cup \phi" \models F_O\}$$

the transition relation

$$R = \{(s_i, s_j) = ((\phi_i, \phi_i', \phi"_i), (\phi_j, \phi_j', \phi"_j)) | \psi, \phi_i \cup \phi_i' \cup (\phi_j' \circ \eta^{-1}) \models F_T\}.$$

A path $\pi$ of $T$ is an infinite sequence of $s_0, s_1, \cdots$ such that $(s_i, s_{i+1}) \in R$ and $i \geq 0$. $\pi_i = \{s_i, s_{i+1}, \cdots\}$ a suffix path starting from $s_i$. A state formula (resp. path formula) has a meaning relative to a state (or a path) and the assignment to the ordinary variables. Let $Val_{s\cup\sigma}(t)$ denote the value of variable $t$ on the state $s$, and a $\psi-$compatible assignment $\sigma$ to the ordinary variables.

$$\begin{array}{lcl}
s, \sigma & \models & t_1 = t_2 \ \text{ iff } \ Val_{s\cup\sigma}(t_1) = Val_{s\cup\sigma}(t_2). \\
s, \sigma & \models & !p \ \text{ iff } \ \text{it is not the case that } s, \sigma \models p. \\
s, \sigma & \models & p\&q \ \text{ iff } \ s, \sigma \models p \text{ and } s, \sigma \models q. \\
s, \sigma & \models & p \mid q \ \text{ iff } \ s, \sigma \models p \text{ or } s, \sigma \models q. \\
s, \sigma & \models & p \rightarrow q \ \text{ iff } \ s, \sigma \models !p \text{ or } s, \sigma \models q. \\
s, \sigma & \models & \mathsf{LET}\ (v = t)\mathsf{IN}\ p \ \text{ iff } \ s, \sigma' \models p \\
 & & \text{where } \sigma' = \sigma \setminus \{(v, \sigma(v))\} \cup \{(v, Val_{s_0\cup\sigma}(t))\} \\
s_i, \sigma & \models & \mathsf{A}p \ \text{ iff } \ \pi_i, \sigma \models p \text{ for every path } \pi_i = (s_i, s_{i+1}, \ldots). \\
s_i, \sigma & \models & \mathsf{E}p \ \text{ iff } \ \pi_i, \sigma \models p \text{ for some path } \pi_i = (s_i, s_{i+1}, \ldots).
\end{array}$$

$$\begin{array}{lcl}
\pi_i, \sigma & \models & p \;\text{ where p is a state formula, iff }\; s_i, \sigma \models p. \\
\pi_i, \sigma & \models & !p \;\text{ iff }\; \text{it is not the case that } \pi_i, \sigma \models p. \\
\pi_i, \sigma & \models & p\&q \;\text{ iff }\; \pi_i, \sigma \models p \text{ and } \pi_i, \sigma \models q. \\
\pi_i, \sigma & \models & p \mid q \;\text{ iff }\; \pi_i, \sigma \models p \text{ or } \pi_i, \sigma \models q. \\
\pi_i, \sigma & \models & p \rightarrow q \;\text{ iff }\; \pi, \sigma \models !p \text{ or } \pi_i, \sigma \models q. \\
\pi_i, \sigma & \models & \mathsf{X}p \;\text{ iff }\; \pi_{i+1}, \sigma \models p. \\
\pi_i, \sigma & \models & \mathsf{G}p \;\text{ iff }\; \pi_j, \sigma \models p \text{ for all } j \geq i. \\
\pi_i, \sigma & \models & \mathsf{F}p \;\text{ iff }\; \pi_j, \sigma \models p \text{ for some } j \geq i. \\
\pi_i, \sigma & \models & p\mathsf{U}q \;\text{ iff }\; \text{for some } k \geq 0, \pi_k, \sigma \models q, \\
 & & \;\text{and } \pi_j, \sigma \models p \text{ for all } j(i \leq j < k). \\
\pi_i, \sigma & \models & \mathsf{LET}\ (v = t)\mathsf{IN}\ p \;\text{ iff }\; \pi, \sigma' \models p \\
 & & \text{where } \sigma' = \sigma \setminus \{(v, \sigma(v))\} \cup \{(v, Val_{s_i \cup \sigma}(t))\}
\end{array}$$

$\mathcal{L}_{MDG}$

$\mathcal{L}_{MDG}$ is a language proposed in [81] defined as a subset of *Abstract_CTL*$^*$ [82] to describe the property in the MDG MC application. The syntax of $\mathcal{L}_{MDG}$ is given in BNF (Backus Normal Form). A *terminal symbol* is written in bold style, a *nonterminal symbol* is written in regular style, starting with an upper case letter. Square brackets denote options. The start symbol is the Property-file.

Property-file ::=
    Property;

Property ::=
    A (Next-let-formula)
    | AG(Next-let-formula)
    | AF(Next-let-formula)
    | A (Next-let-formula) U (Next-let-formula)
    | AG((Next-let-formula) ⇒ (F(Next-let-formula)))
    | AG((Next-let-formula )⇒ ((Next-let-formula) U (Next-let-formula)))

Next-let-formula ::=

X Next-let-formula

| LET (Let-equation) IN (Next-let-formula)

| Next-let-formula $\rightarrow$ Next-let-formula

(Note: the first Next-let-formula can only contain concrete variables)

| Next-let-formula & Next-let-formula

| Next-let-formula | Next-let-formula

| ! Next-let-formula

(Note: the Next-let-formula can only contain concrete variables)

| (Next-let-formula)

| Basic-formula

Basic-formula ::=

Lterm = Rterm | T | F

Lterm ::= ASM-variable

(Note: The input, output and state variables of the ASM)

Rterm ::= ASM-variable

| OrdVar

| IntegerConstant

| SymbolicConstant

| Function (only applies in a Next-let-formula prefixed by Let-equation)

Let-equation ::=

Let-equation & Let-equation

| (Let-equation)

| OrdVar = ASM-variable

$\mathcal{L}_{MDG}$ is constructed from $Abstract\_CTL^*$ by imposing some restrictions. First, $\mathcal{L}_{MDG}$ only allows the universal path quantification and requires it appearing at the beginning of the formula. Thus, $\mathcal{L}_{MDG}$ is built on Linear Time Temporal (LTL) structure. Recall that an LTL formula works on all paths of the structure with an implicit universal quantification representation. Second, $\mathcal{L}_{MDG}$ restricts the application of temporal operators by Next-let-formula, which can only accept temporal operator $\mathsf{X}$. Third, $\mathcal{L}_{MDG}$ allows limited nesting of temporal operators, as shown in the expression of *property*. Thus, $\mathcal{L}_{MDG}$ defines a very restricted subset of $Abstract\_CTL^*$. This restriction is needed for the model checking algorithms proposed in [81, 82].

### 3.3.4 Comparison of Logics

To summarize the above kinds of temporal logics, we proceed a comparison as shown in Figure 3.3.

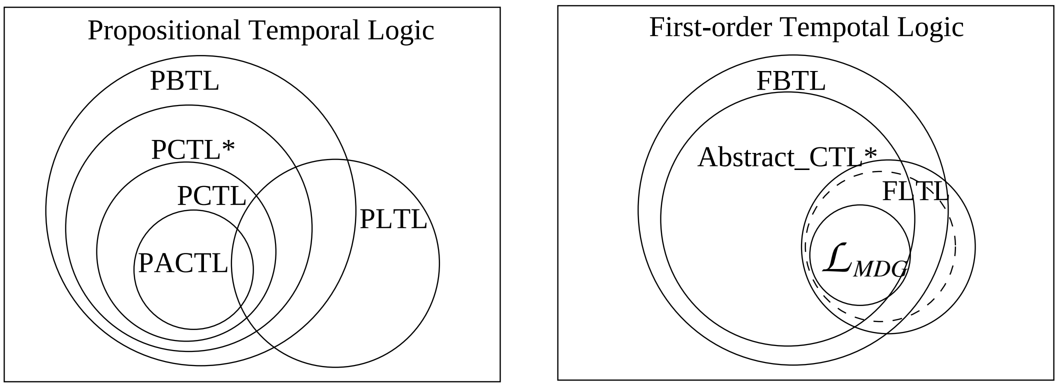



Figure 3.3: The comparison of temporal logics

Figure 3.3 shows the containment relation between various propositional temporal logics and the containment relation between first-order temporal logics. However, this graph does not show the containment between first-order temporal logics and their corresponding propositional counterparts, since first-order temporal logics naturally contain their propositional counterparts. The dashed circle on the right side of Figure 3.3 indicates the temporal logic, $\mathcal{L}_{MDG}{}^*$, that will be developed in the remainder of this thesis.

## 3.4 $\omega$-Automata based Model Checking

In this section, we describe the $\omega$-automata based model checking [18, 39] techniques. Given a finite state machine $M$ and a propositional Linear Temporal Logic (LTL) formula $\varphi$, one can construct an $\omega-$automaton $\mathcal{B}_\varphi$, accepting exactly the behaviors (time line structures) satisfying the formula $\varphi$, and an $\mathcal{B}_M$ accepting all behaviors (paths) of $M$. We consider the behaviors as $\omega-$words over $2^{AP}$ since either paths or time line structures can be viewed as a sequence of states, and each state is labeled with a set $\mathcal{L}(AP)$ of propositions that $s$ satisfies. The verification problem thus is reduced to checking the language containment $\mathcal{L}_\omega(\mathcal{B}_M) \subseteq \mathcal{L}_\omega(\mathcal{B}_\varphi)$ [57]. It is equivalently to check that the automaton accepting $\mathcal{L}_\omega(\mathcal{B}_M) \cap \mathcal{L}_\omega(\mathcal{B}_\neg\varphi)$ is empty [18, 39, 57], where $\mathcal{L}_\omega(\mathcal{B}_\neg\varphi) = \Sigma^\omega - \mathcal{L}_\omega(\mathcal{B}_\varphi)$ accepts all $\omega-$words that do not satisfy $\varphi$ since a behavior satisfies either $\varphi$ or $\neg\varphi$.

In summary, the $\omega$-automata based model checking procedure can be divided into the following steps:

1. Build a *generalized Büchi automaton* $\mathcal{B}_\neg\varphi$ (accepting those behaviors violating $\varphi$) for the negation of the formula $\varphi$ (Negating the formula $\varphi$ is simply done by prefixing it with the negation operator.)

2. Compute the product of the system design automaton $M$ and the automaton $\mathcal{B}_\neg\varphi$.

3. Check if the language of this product automaton is non-empty.

Next, we will detail each step in the following subsections.

### 3.4.1 Constructing Büchi Automaton from LTL

The construction of Büchi Automaton from LTL is based on the intimate connection between LTL and automaton theory. The computation is defined as an infinite sequence of states, and every state is described by a finite set of atomic propositions in the verification application, so a computation can be viewed as an infinite word over the alphabet of truth assignments to the atomic propositions. Therefore, temporal logic formulas can be viewed as finite state acceptors. More precisely, given any propositional temporal formula, one can construct a finite automaton on infinite words that accepts precisely the computations satisfied by the formula.

Next, we show how to construct a *generalized Büchi automaton* $\mathcal{B}_\varphi$ for a given LTL formula $\varphi$ such that $\mathcal{B}_\varphi$ accepts precisely those runs over which $\varphi$ holds.

The LTL formulas that one deals with are in negation normal form, which can be transformed by pushing all negations inside until they precede only propositions. The core of the algorithm is based on expanding the formula using two expansion rules:

$$\psi_1 \mathsf{U} \psi_2 = \psi_2 \vee (\psi_1 \wedge X(\psi_1 \mathsf{U} \psi_2)),$$

$$\psi_1 \mathsf{R} \psi_2 = \psi_2 \wedge (\psi_1 \vee X(\psi_1 \mathsf{R} \psi_2)),$$

where $\psi_1, \psi_2$ are LTL formulas.

Given a LTL formula $\phi$, these rules are applied to expand $\varphi$ until the resulting expression is a propositional formula in terms of elementary subformula of $\phi$, where an elementary formula is referred to as a constant, an atomic proposition, or a formula starting with $\mathsf{X}$. The expanded formula, put in disjunctive normal form, is an elementary *cover* of $\phi$. Each term of the *cover* identifies a state of the automaton. The atomic propositions and their negations in the term define the label of the state. The remaining elementary subformula of the term forms the next part of the term; they are *LTL* formulas that identify the obligations that must be fulfilled to obtain an accepting run. The expansion process is applied to the next part of each state, creating new *covers* until new obligations are produced. In this way, a closed set of the elementary *covers* is obtained. This set is closed in the sense that there is an elementary *cover* in the set of the next part of each term of each *cover* in the set [26, 39].

The automaton is built from the nodes obtained by connecting each state to the states in the *cover* for its next. The states in the elementary *cover* of $\phi$ are the initial states. Acceptance conditions are added to the automaton for each elementary subformula of the form $\mathsf{X}(\psi_1 \cup \psi_2)$. The acceptance condition contains all the states $s$ such that the label of $s$ does not imply $\psi_1 \cup \psi_2$ or the label of $s$ implies $\psi_2$.

### 3.4.2 Product Operating of Büchi Automaton

The model checking of a LTL property $\varphi$ on a system design $M$ can be reduced to check whether the language defined by the product of $M$ and $\mathcal{B}_{\neg}\varphi$ is empty or not.

Formally, assume that given a Kripke structure $M = (S, S_0, R, L)$ over a set $AP$ of atomic propositions and $\mathcal{B}_{\neg}\varphi = (Q, Q_0, \delta, \ \mathcal{F})$ that accepts precisely those $\omega-$words that do not satisfy $\varphi$. The model checking algorithm operates on pairs $(s, q)$ of system states and automaton states. A pair $(s_0, q_0)$ is initial if $s_0 \in S_0$ and $q_0 \in Q_0$ are initial states for $M$ and $\mathcal{B}_{\neg}\varphi$, respectively. A pair $(s', q')$ is a successor of $(s, q)$ if both $(s, s') \in R$ and $(q, q') \in \delta$ hold: $M$ and $(B)_{\neg}\varphi$ make joint transition. Thus, a pair $(s, q)$ is accepting if $q \in \ \mathcal{F}$ is an accepting automaton state; recall that $M$ does not define any accepting condition.

### 3.4.3 Language Emptiness Checking Algorithms

The language emptiness checking of a Büchi automaton, $\mathcal{B} = (Q, Q_0, \delta, \ \mathcal{F})$, can be reduced to a fair cycle detection problem. A fair cycle is a cycle reachable from the initial states and has at least a state in the acceptance set. A trivial method is as follows: for each $s \in F$, check whether there is a cycle and if there is a cycle, then further check if $s$ is reachable from the initial states. The complexity of the computation is high, the running time is at least $|Q| \circ |F|$ quadratic in the size of the automaton.

Another way to detect fair cycles is to compute Maximal Strongly Connected Components (MSCCs). $C \subseteq Q$ is an SCC of $\mathcal{B}$ iff for all $q, q' \in C$ we have $q \rightarrow^* q'$ (q' is reachable from q); $C$ is maximal if there does not exist an SCC $C'$ such that $C \subseteq C'$. $C$ is trivial iff $C = \{q\}$ and $q \nrightarrow q((q, q) \notin \delta)$. A Nested Depth First Search (DFS) [23] algorithm is used to compute maximal strongly connected components in the explicit model checking method. DFS mainly uses two phases either interleaved or one by one. The first phase starts a depth-first search at each initial state (abort at states that were visited previously during the first phase), number the states in post order, and assign a number to state $s$ when the search backtracks from $s$. The second phase considers the accepting states in the post order imposed by the first phase, and does the following for each accepting state $q$: starting a depth-first search at $q$, cutting off the search at states visited previously during the second phase. If the search "comes back" reaches $q$, a cycle around $q$ has been found. The running time is linear in $|Q| + |T|$ (each state and each transition are visited at most twice). There is an implementation called "on-the-fly" model checking which constructs an automaton during the depth-first search [39].

A symbolic approach to check the emptiness of a Büchi automaton was initially based on the transformation to fair CTL model checking [19] and only recently SCC-hull computation algorithms are presented [35, 73]. An SCC hull is a set of states containing all MSCCs. These algorithms [35, 45, 48, 53] are based on the computation of two fixpoints:

$$Reachability(S) = \mu Z.\ (S \cup\ image(Z))$$

and

$$Elimination(S) = \nu Z.\ (S \cap\ image(Z))$$

where $\mu$ and $\nu$ are the least and the greatest fixpoint operators respectively.

The function $Reachability(S)$ computes the set of states that are reachable from the set $S$. The function $Elimination(S)$ computes the set of all states either in any cycles in $S$ or reachable from any cycles. The complexity of the algorithm is quadratic, since it involves computation of nested fixed points. We will detail these algorithms in Chapter 6.

## 3.5 Conclusion

In this chapter, we reviewed the basic concepts on $\omega$-automata based model checking. We first presented the system design model: Kripke structure and its ROBDD representation. We then introduced $\omega$-automata theory. We also described the property language: propositional logic, first-order logic and temporal logic. We finally gave the principle of the $\omega$-automata model checking method. This chapter provides a basic knowledge for the rest of this thesis.

# Chapter 4

# MDG Language Emptiness Checking Approach

Previous chapter described a verification method of using $\omega$-automata to check if a finite state machine defines a model of a propositional temporal logic formula. In this chapter, we propose to embody this method in the MDG tool set. Our method models the system design as an Abstract State Machine (ASM) and describes the property in a first-order temporal logic. To check the satisfaction of the temporal logic formula on all computations of the ASM, we propose a transformation algorithm. This algorithm constructs a bisimular state machine to the ASM to be checked with respect to the property. We further check the property on the constructed state machine.

In this chapter, we start with describing the structure of our Language Emptiness Checking (LEC) method and then present the specification language $\mathcal{L}_{MDG}{}^{*}$. After that, we present the transformation algorithm and its proof. We finish this chapter by illustrating our method using an example of Abstract Counter.

## 4.1 The Structure of the MDG LEC

The MDG based model checking using $\omega$-automata accepts a first-order temporal logical ($\mathcal{L}_{MDG}{}^{*}$) formula as a property and an ASM as the system model and answers if the property is satisfied by all the computations of the system design or not. The task can be divided into four steps as shown in Figure 4.1.

1. Transform the $\mathcal{L}_{MDG}{}^{*}$ formula $\varphi$ into a propositional LTL (PLTL) formula $\phi$ by constructing an ASM for each atomic formula in $\varphi$ and compose the constructed ASMs with the original ASM model $\mathcal{C}$.

2. Generate a *generalized Büchi automaton* $\mathcal{B}_{\neg\phi}$ from $\phi$ using an existing expansion algorithm (see Section 3.4.2).

3. Compose $M$ and $\mathcal{B}_{\neg\phi}$ to produce a product automaton. The product is composed of pairs $(s, q)$ of transition machine states $s$, and the automaton states $q$. A pair $(s_0, q_0)$ is an initial state if $s_0$ is in the initial states of $M$, and $q_0$ is in the initial states of $\mathcal{B}_{\neg\phi}$ respectively. A pair $(t, p)$ is a successor of $(s, q)$ if $(s, t)$ is in the transition relation of $M$ and $(q, p)$ in the transition relation of $\mathcal{B}_{\neg\phi}$. Thus, a pair $(s, q)$ is accepted if $q$ is an accepting automaton state.

4. Finally, check if the language of the product machine is empty or not using a LEC algorithm, which will be discussed in the next chapter.

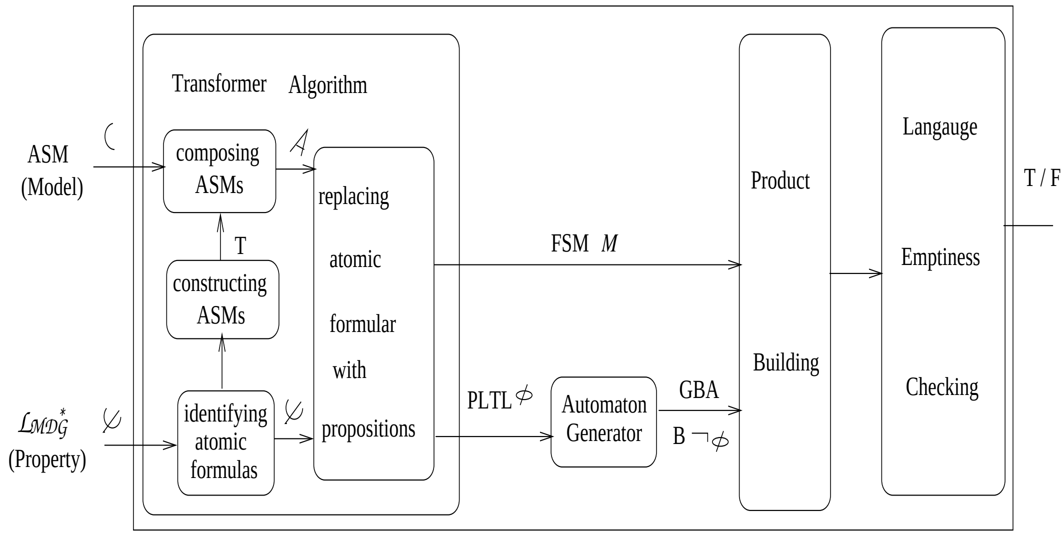



Figure 4.1: The structure of MDG LEC

Next, we will detail the specification language $\mathcal{L}_{MDG}{}^*$ and the transformation algorithm. We start with $\mathcal{L}_{MDG}{}^*$.

## 4.2 $\mathcal{L}_{MDG}{}^*$

$\mathcal{L}_{MDG}{}^*$ is a first-order temporal logic derived from a linear time structure $M$ with a many-sorted first-order language (see Chapter 2) interpretation to the states of $M$. $M = (S, x, L)$, where $S$ is a set of states, $x : N \to S$ is an infinite sequence of states, and $L : S \to 2^{\mathcal{L}}$ is a labeling of each state with the symbols.

**Symbols of** $\mathcal{L}_{MDG}{}^*$ are composed of sorts, constants, variables and function symbols. Sorts consist of concrete sorts and abstract sorts. Concrete sorts have an enumeration while abstract sorts have not. Constants, variables and functions symbols have sorts.

**Syntax of** $\mathcal{L}_{MDG}{}^*$ The terms of $\mathcal{L}$ are defined inductively by the following rules:

**T1** Each constant $a$ of sort $\alpha$ is a term of type $\alpha$;

**T2** Each variable $y$ of sort $\alpha$ is a term of type $\alpha$;

**T3** If $A_1, \ldots, A_n$ are terms of sort $\alpha_1$, $\alpha_2$, ..., $\alpha_n$, and $f$ is an $n-$ary function symbol $\alpha_1 \times \ldots \times \alpha_n \to \alpha_{n+1}$, then $f(A_1, \ldots, A_n)$ is a term of sort $\alpha_{n+1}$.

The atomic formulas of $\mathcal{L}$ are defined by the following rules:

**A1** $\mathsf{T}$(truth) and $\mathsf{F}$(falsity) are atomic formulas;

**A2** If $A_1, A_2$ are terms of the same sort $\alpha$, then $A_1 = A_2$ is atomic formula;

**A3** If $p$ is an atomic formula, then $\mathsf{X}\ p$ is an atomic formula ($\mathsf{X}$-formula);

**A4** If $p$ is an atomic formula, then $\mathsf{LET}\,(v = t)\ \mathsf{IN}\,p$ is an atomic formula, where $t$ is a variable of abstract sort and $v$ is an *ordinary variable* used to remember the value of $t$ at the current state.

Finally, the (compound) formulas are defined inductively as follows:

**F1** Each atomic formula is a formula;

**F2** If $p$ and $q$ are formulas, then $(p \wedge q)$, $\neg p$ are formulas;

**F3** If $p$, $q$ are formulas, then so are $p \; \mathsf{U} \; q$, $\mathsf{X} \; p$.

**Semantics of $\mathcal{L}_{MDG}{}^*$.** The semantics of $\mathcal{L}_{MDG}{}^*$ is defined on the linear-time structure $M = (S, x, L)$. Given an interpretation $\psi$ and $\psi-$compatible assignment $\phi$ to all variables, the interpretation $\psi$ assigns an appropriate meaning over each sort domain to the $\mathcal{L}_{MDG}{}^*$ symbols as follows:

- $(M, x)(\alpha) = \psi(\alpha)$, where $\alpha$ is an abstract sort and $\psi(\alpha)$ is a nonempty set.
- $(M, x)(\alpha) = \psi(\alpha)$, where $\alpha$ is a concrete sort with enumeration $\{a_1, \ldots, a_n\}$, $\psi(\alpha) = \{\psi(a_1), \ldots, \psi(a_n)\}$ and $\psi(a_i) \neq \psi(a_j)$ for $1 \leq i < j \leq n$.
- $(M, x)(a) = \psi(a)$, where $a$ is a constant of abstract sort $\alpha$, $\psi(a) \in \psi(\alpha)$.
- $(M, x)(a) = \psi(a)$, where $a$ is a constant in the enumeration of sort $\alpha$, $\psi(a)$ is the interpretation of $a$.
- $(M, x)(y) = \phi(y)$, where $y$ is a variable of sort $\alpha$, and $\phi(y)$ is an element of $\psi(\alpha)$.
- $(M, x)(f) = \phi(f)$, where $f$ is a $n$-ary function symbol $(n \geq 1)$ of $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, $\phi(f)$ is an element of $\psi(\alpha_{n+1})$.

- $(M, x) \models A_1 = A_2$ iff $(M, x)(A_1) = (M, x)(A_2)$.
- $(M, x) \models (\mathsf{X} \; p)$ iff $(M, x^1) \models p$.
- $(M, x) \models (\mathsf{LET} \, (v = t) \mathsf{IN} \; p)$ iff $(M', x) \models p$, where $M'$ assign the value $t$ to $v$ at state $s_0$ and otherwise coincides with M; and $x = (s_0 s_1 s_2 \ldots)$.
- $(M, x) \models \neg p$ iff not the case $(M, x) \models p$.

- $(M, x) \models p \wedge q$ iff $(M, x) \models p$ and $(M, x) \models q$.
- $M, x \models (p \mathsf{U} q)$ iff $\exists\, j\ (M, x^j \models q$ and $\forall k < j(M, x^k \models p))$.

A formula $f$ of $\mathcal{L}_{MDG}{}^*$ is valid iff for every first-order linear time structure $M$, we have $M, x \models f$. The formula of $f$ is satisfiable iff there exists $M$ such that $M, x \models f$.

Recall that properties in $\mathcal{L}_{MDG}$ defined in Chapter 3.3.3 are composed of assigned templates. On the other hand, $\mathcal{L}_{MDG}{}^*$ properties can be an arbitrary combination of the atomic formulas with temporal operators. Thus, $\mathcal{L}_{MDG}{}^*$ breaks through the limitations of the nesting of temporal operators in $\mathcal{L}_{MDG}$. Also, for every $\mathcal{L}_{MDG}$ formula, we can find a corresponding $\mathcal{L}_{MDG}{}^*$ formula, which have the same semantics. However, there are some properties in $\mathcal{L}_{MDG}{}^*$ that cannot be represented by $\mathcal{L}_{MDG}$. For example, $\mathsf{G}(req_{11} = 1 \wedge req_{22} = 1 \rightarrow \mathsf{F}\ (Dout_1 = Din_1) \wedge \mathsf{F}\ (Dout_2 = Din_2))$ cannot be written in $\mathcal{L}_{MDG}$. Thus, $\mathcal{L}_{MDG}{}^*$ is a superset of $\mathcal{L}_{MDG}$.

Next, we give some examples of properties in $\mathcal{L}_{MDG}{}^*$.

**Example 4.2.1** *From state $fecth\_st$, if the input is $inc2$, then the counter $pc$ has been increased by two (twice increases by one) in three transition steps.*

$$G((state = fectch\_st \wedge input = inc2) \rightarrow \mathsf{LET}\,(v = pc)\ \mathsf{IN}\ X(X(X(pc = inc(inc(v))))),$$

*where $inc$ is an abstract function symbol to represent increase by one.*

**Example 4.2.2** *From state $fecth\_st$, if the input is $inc2$, then the machine always reaches state $inc2\_st$ in two transition steps.*

$$G((state = fectch\_st\ \wedge\ input = inc2) \rightarrow \mathsf{X}(\mathsf{X}(state = inc2\_st))).$$

**Example 4.2.3** *If there is a request ($req = 1$), then the (abstract) data at input port $Din$ will show up at outport $Dout$ in the next state.*

$$G(req = 1\ \rightarrow\ \mathsf{LET}\,(v = Din)\ \mathsf{IN}\ \mathsf{X}\ (Dout = v)$$

**Example 4.2.4** *If there is a request ($req = 1$), then an acknowledgment ($ack = 1$) will be eventually generated.*

$$G(req = 1 \rightarrow \mathsf{F}(ack = 1))$$

**Example 4.2.5** *Whenever a pedestrian presses the button of the traffic light ($req = 1$), he/she will receive the green light, $light\_color = green$, within 3 clock cycles.*

$$G(req = 1 \rightarrow \mathsf{X}(light\_color = green) \vee$$

$$\mathsf{X}(\mathsf{X}(light\_color = green)) \vee \mathsf{X}(\mathsf{X}(\mathsf{X}(light\_color = green))))$$

**Example 4.2.6** *If inport #1 requests output #1 ($req_{11} = 1$) and inport #2 requests output #2 ($req_{22} = 1$), then the (abstract) data at inport $Din_1$ will show up at outport $Dout_1$ and the data at inport $Din_2$ will show up at outport $Dout_2$ in the future, but the input data may not be shown up at the same time.*

$$G(req_{11} = 1 \wedge req_{22} = 1 \ \rightarrow \ \mathsf{F}\ (Dout_1 = Din_1) \wedge \mathsf{F}(\ Dout_2 = Din_2))$$

## 4.3 Transformation Algorithm

To transform an ASM $\mathcal{C}$ into a state machine $\mathcal{A}$, the algorithm starts with the set of atomic formulas $P$ from the property $\varphi$. We build an ASM $T$ for each atomic formula $p \in P$. A key property of the construction is that for every path of $\mathcal{C}$ with respect to $p$, there is a corresponding path in $T$. We further compute the product machine $\mathcal{A}$ and label the composed states with $p$. $\mathcal{A}$ and $\mathcal{C}$ are bisimular related with respect to $P$. Thus, the verification of $\mathcal{C}$ with respect to $\forall\varphi$ is equivalent to the verification of $\mathcal{A}$ with respect to $\forall\varphi$. We next illustrate this algorithm with an example.

### 4.3.1 Example: An Alarm Setting Controller

Consider a controller [6] that sets an alarm if a frequently read-in input value grows too fast.

The state machine has one input variable $X = \{x\}$ and 4 state variables denoted by $Y = \{l, k, in, alarm\}$, where $in$ and $alarm$ are of the Boolean sort $B$, a concrete sort with enumeration $\{\mathsf{T}, \mathsf{F}\}$, and $x, l, k$ are of abstract sort $S$. The intended interpretation of $S$ is an infinite set equipped with a total order $\leq$, i.e, the set of natural numbers. A graph representation of the *controller* state machine is shown in Figure 4.2. The circles correspond to the control transition of the machine. The transition label specifies the conditions under which each transition is taken and an assignment of values to the next-state variables $l'$, $alarm'$, $in'$ and $k'$.

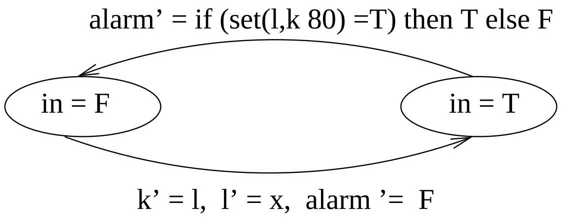



Figure 4.2: An alarm setting controller

The machine stores in $k$ the old input value and reads in $l$ a new value. The controller sets the alarm using an operator $set(a, b, 80)$ such that for any two values $a$ and $b$ of sort $S$, $set(a, b, 80) = \mathsf{T}$ if and only if $a$ minus $b$ is greater or equal to 80. The intended denotation of $set$ is the characteristic function of the ordering relation $\geq$ with $(a - b)$ and 80.

In the abstract description of the controller state machine, the abstract sort $S$ is uninterpreted and $set$ is an uninterpreted cross-operator of type $S \times S \times S \to B$. The initial states are represented by a *directed formula* $F_I$: $in = \mathsf{F} \ \wedge \ l = l_0 \ \wedge \ k = k_0 \ \wedge \ alarm = \mathsf{F}$, where $l_0$ and $k_0$ are two generic constants of sort $S$. The state transition relation can be described by the *directed formula* $F_T$: $(in = \mathsf{F} \ \wedge \ x = x_i \ \wedge \ alarm' = \mathsf{F} \ \wedge \ l' = x_i \ \wedge \ k' = l \ \wedge \ in' = \mathsf{T}) \ \bigvee \ (in = \mathsf{T} \ \wedge \ set(l, k, 80) = 1 \ \wedge \ alarm' = \mathsf{T} \ \wedge \ l' = l \ \wedge \ k' = k \ \wedge \ in' = \mathsf{F}) \ \bigvee \ (in = \mathsf{T} \ \wedge \ set(l, k, 80) = 0 \ \wedge \ alarm' = \mathsf{F} \ \wedge \ l' = l \ \wedge \ k' = k \ \wedge \ in' = \mathsf{F})$.

The specification of the controller is

$$G(in = \mathsf{T} \ \rightarrow \ (\mathsf{LET}\,(v = l) \ \mathsf{IN} \ \mathsf{XX} \ set(l, v, 80) = \mathsf{T} \rightarrow \mathsf{XXX}(alarm = \mathsf{T}))). \quad (4.1)$$

The atomic formulas $P$ from this property are $p1$, $p2$ and $p3$, where $p1$ is $in = \mathsf{T}$, $p2$ is $\mathsf{LET}\,(v = l) \ \mathsf{IN} \ \mathsf{X}(\mathsf{X}(set(l, v, 80) = \mathsf{T}))$, and $p3$ is $alarm = \mathsf{T}$. The first step is to construct an ASM for each atomic formula.

We construct an ASM $T_1$ for $p1$ that contains all computations $R_1$, where $r \in R_1$ is $t_0t_1t_2t_3\ldots$ such that $\mathcal{L}(r) = \mathcal{L}(t_0)\ \mathcal{L}(t_1)\ \mathcal{L}(t_2)\ldots$ and $\mathcal{L}(t_i) \in \{in = \mathsf{T}, \neg(in = \mathsf{T})\}$ for all $i \geq 0$. The transition system $T_1 = \{S,\ \delta,\ L\}$ labeled with atomic formula $in = \mathsf{T}$ is shown in Figure 4.3, where $S = \{s_0, s_1\}$, $\delta$ is as shown in the figure, and the labeling of the states are $L(s_0) = \{in = \mathsf{T}\}$ and $L(s_1) = \{\neg\ (in = \mathsf{T})\}$.

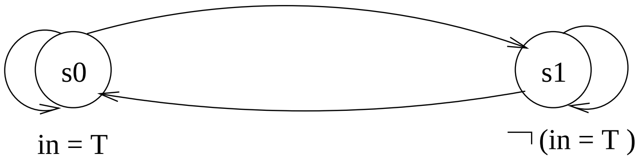



Figure 4.3: The ASM for $in = \mathsf{T}$

We then construct an ASM $T_2$ for $p2$ that contains all computations $R_2$, where $r \in R_2$ is $t_0t_1t_2t_3\ldots$ such that $\mathcal{L}(r) = \mathcal{L}(t_0)\ \mathcal{L}(t_1)\ \mathcal{L}(t_2)\ldots$, $\mathcal{L}(t_0) = \mathcal{L}(t_1) = \{\mathsf{LET}\,(v = l)\ \mathsf{IN}\ \mathsf{X}(\mathsf{X}(set(l, v, 80) = \mathsf{T}))\}$; and $\mathcal{L}(t_i) \in \{\mathsf{LET}\,(v = l)\ \mathsf{IN}\ \mathsf{X}(\mathsf{X}(set(l, v, 80) = \mathsf{T})), \neg(\mathsf{LET}\,(v = l)\ \mathsf{IN}\ \mathsf{X}(\mathsf{X}(set(l, v, 80) = \mathsf{T})))\}$ for all $i \geq 2$. The transition system $T_2 = \{S, \delta, L\}$ for the atomic formula $\mathsf{LET}\,(v = l)\ \mathsf{IN}\ \mathsf{X}(\mathsf{X}(set(l, v, 80) = \mathsf{T}))$ is shown in Figure 4.4, where $S = \{s_0,\ s_1,\ s_2,\ s_3\}$, $\delta$ is as shown in the figure, and the labeling of the states are $\mathcal{L}(s_0) = \mathcal{L}(s_1) = \{\mathsf{LET}\,(v = l)\ \mathsf{IN}\ \mathsf{X}(\mathsf{X}(set(l, v, 80) = \mathsf{T}))\}$ and $L(s_2) = \{\mathsf{LET}\,(v = l)\ \mathsf{IN}\ \mathsf{X}(\mathsf{X}(set(l, v, 80) = \mathsf{T}))\}$, and $L(s_3) = \{\neg(\mathsf{LET}\,(v = l)\ \mathsf{IN}\ \mathsf{X}(\mathsf{X}(set(l, v, 80) = \mathsf{T})))\}$.

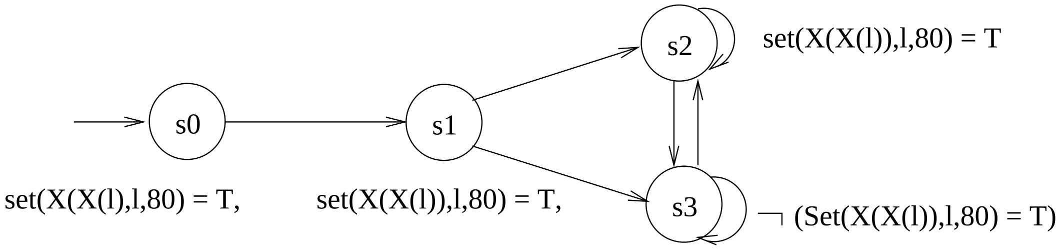



Figure 4.4: The ASM for $\mathsf{LET}\,(v = l)\ \mathsf{IN}\ \mathsf{X}(\mathsf{X}(set(l, v, 80) = \mathsf{T}))$

Finally, an ASM $T_3$ is constructed for $p3$. $T_3$, shown in Figure 4.5, is the same as $T_1$ except that the labeling $in = \mathsf{T}$ of states is replaced by $alarm = \mathsf{T}$.

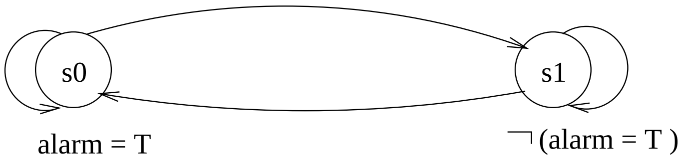



Figure 4.5: The ASM for $alarm = \mathsf{T}$

The transition systems $T_1$, $T_2$ and $T_3$ are composed into a transition system $T$ in a suitable way, which will be described in the next subsection, to comprise all computations $R$ in the three structures $R_1$, $R_2$, and $R_3$; and each $r \in R$ is $t_0t_1t_2\ldots$, where $t_i = (t_i^1, t_i^2, t_i^3)$ composed of states from $T_1$, $T_2$ and $T_3$; and $\mathcal{L}(r) = \mathcal{L}(t_0)\ \mathcal{L}(t_1)\ \mathcal{L}(t_2)\ldots$ and $\mathcal{L}(t_0) \in \{p1, \neg p1\} \times \{p2\} \times \{p3, \neg p3\}$ for $i = 0$ and $i = 1$. $\mathcal{L}(t_i) \in \{p1, \neg p1\} \times \{p2, \neg p2\} \times \{p3, \neg p3\}$ for all $i \geq 2$. $T$ is further composed with the system under verification $\mathcal{C}$ to construct a machine $\mathcal{A}$ and labels states with $p1, p2$ and $p3$.

### 4.3.2 Transformation Algorithm

The alarm setting controller illustrated the key ingredients of our transformation algorithm, which is summarized as follows:

**Algorithm 4.3.1** *(Transformation Algorithm)*

1. *Identify the set of atomic formulas $P = \{p_1, \cdots, p_n\}$ in the property formula;*

2. *Construct an ASM for each atomic formula;*

3. *Compose the constructed ASMs and the original ASM and label the composed states with atomic formulas;*

4. *Replace each atomic formula $p$ with a proposition $b$ in both the composed machine and the property formula.*

The main parts of the Algorithm 4.3.1 are step 2 and 3, which will be detailed in the following.

### The Construction of transition ASMs

We construct a transition ASM for each atomic formula in step 2 of algorithm 4.3.1. Recall that the atomic formulas of ${\mathcal{L}_{MDG}}^*$ can be classified into two classes: $\mathsf{X}-$formulas and formulas without $\mathsf{X}$-operator. The system models differ with these two classes.

The system model for an atomic formula $p_i$ without $\mathsf{X}-$operator can be described by a two-state structure as shown in Figure 4.6.

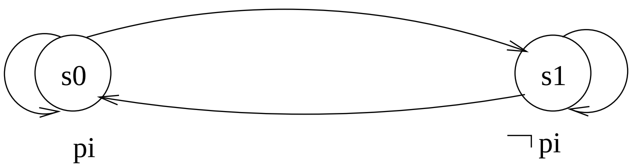



Figure 4.6: The transition system for an atomic formula without $\mathsf{X}$-operator

The system model for an atomic formula $p_j$ with $\mathsf{X}$-operator is described as a structure, shown in figure 4.7. The number of the states is determined by the maximal number of the nesting of $\mathsf{X}$. The labeling of $s_1$, $s_2$, ..., $s_i$ are either $\{\neg p_j\}$ or the $\{p_j\}$ depending on the temporal operators it follows. If $p_j$ follows the temporal operator $F$ or $U$, the labeling are $\{\neg p_j\}$. If $p_j$ follows operator $G$ or in $p_j\mathsf{U}q$,the labeling are $\{p_j\}$.

### The MDG-HDL components

Using MDG-HDL, we generate a circuit description for each constructed ASM. The circuit description is described in a Prolog-style HDL, MDG-HDL, which allows the use of abstract variables for representing data signals. To construct the circuit model, we use some predefined MDG-HDL components such as *and*, *reg*, *or* and *transform*. *and* implements Boolean $\wedge$ operation of inputs of Boolean sort; *or* implements the

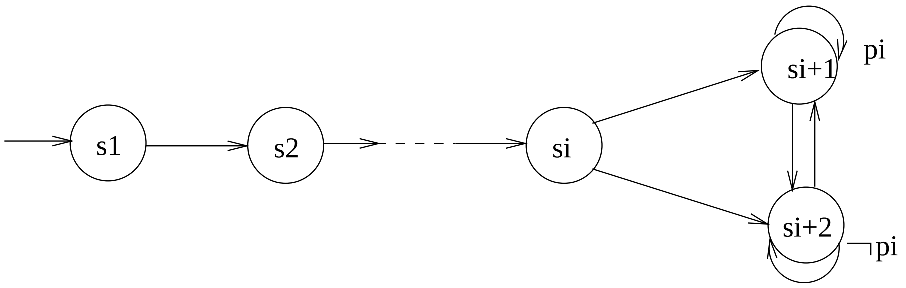



Figure 4.7: The transition system for an atomic formula with $\mathsf{X}$-operator

Boolean $\vee$ operation of inputs of Boolean sort; *reg* stores the valuation of inputs for one transition step; the syntax of *reg* is

$$reg(input(Input : Sort), output(Output : Sort)),$$

where *Output* is the state variable. The initial value of *Output* should be given. The component *transform* is used to build uninterpreted function symbols. The syntax of *transform* is

$$transform(inputs(Inputs : [Sort1, Sort2, ..., Sortn]), function(Func\_symbol),$$

$$output(Output : Sort)).$$

The declaration is viewed as a black-box, and the function is represented using the uninterpreted function symbol $Func\_symbol$. We define a most frequently used function $Comp$ with instantiation of the $transform$ as

$$transform(inputs(Inputs : [Sort1, Sort1, \ldots, Sort1]), function(Comp),$$

$$output(Output : Boolean)).$$

The functionality of *Comp* is accomplished with the rewriting rule $Comp(X, X) = \mathsf{T}$, that is, if the denotations of two inputs are the same *Comp* outputs $\mathsf{T}$. Otherwise, *Comp* outputs $\mathsf{F}$.

Note also that since the MDG package does not use the concrete function symbols, we cannot build a component from *transform* to determine the truth of the atomic formula of terms of concrete sorts. However, to support the circuit description, we define a new component *Equv* as a tabular description. The functionality of the table is

$$b_i = Equv(A_1) = \begin{cases} \mathsf{T} & \phi^{\psi}(A_1) = \psi(A_2), \\ \mathsf{F} & otherwise \end{cases}$$

where $\psi$ is an interpretation, $\phi$ is a $\psi$-compactible assignment and $A_1$ and $A_2$ are terms of concrete sort.

### Rules for building circuit models

With these MDG components, the circuit construction for a constructed ASM can be implemented by the following rules:

First, we describe the rules to build a circuit model for transition ASM as shown in Figure 4.6, which is straightforward. We build a component *Comp* if the atomic formulas are terms of abstract sort and build a component *Equv* if the atomic formula are terms of concrete sort. The inputs of *Comp* or *Equv* are the LHS and RHS of the atomic formula; and the output is the atomic formula which goes to a component *reg*. If the LHS is a function, we build a *transform* component whose inputs are terms consisting of function symbols and output goes to *Comp* or *Equv*. For example, to build a circuit for $set(l, k, 80) = \mathsf{T}$, we first build an *Equv* whose inputs are $\mathsf{T}$ and *set(l,k,80)*. We further build a *transform* with the inputs *l, k* and 80 of sort $S$, function symbol *set* and output connecting to the $set(l, k, 80)$ input of *Equv*.

It is obvious that the constructed circuit model describes the ASM shown in Figure 4.6.

Second, we build a circuit model for the transition ASM shown in Figure 4.7, where we have to consider how to label the states. To assume the property are not failed before we get to the states assigned by temporal operator $\mathsf{X}$, we write the property accordingly. More specifically, we define the following rules:

I. Rewrite the atomic formula $p$ with $\mathsf{X}-$operator. If $p$ follows the temporal operator $F$ or $U$, we rewrite $p$ into $p' = \mathsf{T} \wedge p$. If $p$ follows operator $G$ or in $p \mathsf{U} q$, we rewrite into $p' = \mathsf{F} \vee p$.

II. Circuit Building.

IIa. Build the circuits for the atomic formulas without considering the $\mathsf{X}$ operator and Let_equation. We build an *and* for $\wedge$ and an *or* for $\vee$, and connect the signals accordingly. $\mathsf{T}$ and $\mathsf{F}$ are atomic formulas, which can be represented by the MDG constant component *constant_signal*,

$$constant\_signal(value(Const), signal(Signal : Sort)).$$

*Signal* takes *Const* as its constant value. If *Sort* is a concrete sort, then *Const* must be an individual constant in the enumeration of *Sort*. Or if *Sort* is an abstract sort, then *Const* must be a generic constant of sort *Sort*. For example, for a Boolean constant $\mathsf{F}$, we use an extra signal *signal0* of Boolean sort to represent it and use the following component to generate $signal0$, $constant\_signal(value(\mathsf{F}), signal(signal0)))$.

IIb. Count the number $Num(\mathsf{X})$ of $\mathsf{X}$ operators and construct *RegSeq*, a sequence of $Num(\mathsf{X})$ components *reg*. For Let_equation, we build a connection between the *ordinary variable* $v$ and the abstract variable whose value is remembered by $v$ and add *RegSeq* between them. We also add *RegSeq* to $\mathsf{T}$ (or $\mathsf{F}$) and assign the initial states to $\mathsf{F}$ (or $\mathsf{T}$).

It is obvious the above circuit model realizes the ASM shown Figure 4.7.

For the above property 4.1, we build circuit descriptions for atomic formulas $in = \mathsf{T}$, $\mathsf{LET}\,(v = l)\mathsf{IN}\,\mathsf{X}(\mathsf{X}(set(l, v, 80) = \mathsf{T}))$, and $alarm = \mathsf{T}$. Figure 4.8 shows the resulting descriptions according to the above rules. The construction follows directly from the above rules.

### Composition of the transition systems

Following the Algorithm 4.3.1, a product machine of the abstract state machine $\mathcal{C}$ and the constructed ASMs $T_i$ will comprise all paths that are in $\mathcal{C}$.

Let $\mathcal{C} = (S_{\mathcal{C}}, R_{\mathcal{C}}, L_{\mathcal{C}})$ be the system design model and $T_i = (S_{T_i}, R_{T_i}, L_{T_i})$ be the ASMs for $p_i$, which is an element of the set $P = \{p_1, \cdots, p_n\}$ of atomic formulas from property $\phi$. We use $L(s)|_P$ to denote $L(s) \cap P$.

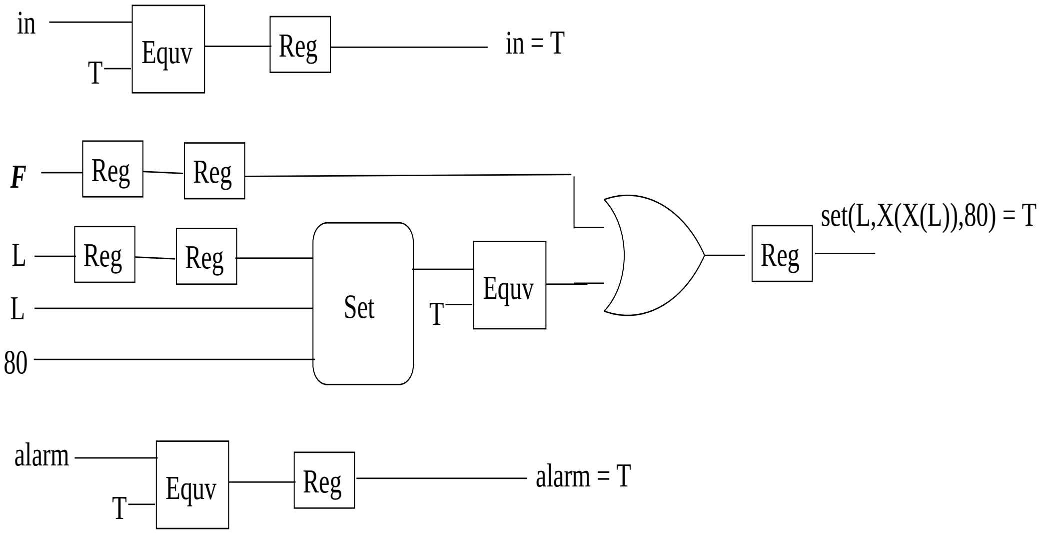



Figure 4.8: The constructed ASMs for atomic formulas $in = \mathsf{T}$, $\mathsf{LET}\ (v = l)\ \mathsf{IN}$ $\mathsf{X}(\mathsf{X}(set(l,v,80) = \mathsf{T}))$ and $alarm = \mathsf{T}$.

**Definition 4.3.1** *Let $n$ be the number of the transition ASMs and $\mathcal{C}$ be the system ASM model. The product $\mathcal{A} = (S_{\mathcal{A}}, R_{\mathcal{A}}, \mathcal{L}_{\mathcal{A}})$ of them consists of*

- *Set of states*

$$S_{\mathcal{A}} = \{(s_1, s_2, \cdots, s_n, t) | s_i \in S_{T_i}, t \in S_{\mathcal{C}},\ \text{and}\ L_{\mathcal{C}}(t)|_P = \bigcup_{1 \le i \le n} L_{T_i}(s_i)\}$$

- *Transition relation*

$$R_{\mathcal{A}} = \{(s_1, \ldots, s_n, t), (s'_1, \ldots, s'_n, t') | (s_i, s'_i) \in R_{T_i}\ \text{and}\ (t, t') \in R_{\mathcal{C}}\}$$

- *Labeling function*

$$L_{\mathcal{A}}(s_1, \ldots, s_n, t) = \bigcup_{1 \le i \le n} L_{T_i}(s_i).$$

**Remark 4.3.1** *In MDG LEC, the product is implicitly computed in the procedure of the language checking. To use it, we need represent the ASMs in MDG-HDL, which are compiled into the internal MDG representations.*

### 4.3.3 Proof of the Transformation Algorithm

The following theorem proves the correctness of the transformation algorithm.

**Theorem 4.3.1** *For every $\mathcal{L}_{MDG}{}^*$ formula $\varphi$ and an ASM $\mathcal{C}$ $\mathcal{C} \models \forall\varphi$ iff $\mathcal{A} \models \forall\varphi$, where $\mathcal{A}$ is transformed from $\mathcal{C}$ by the above transformation algorithm, $\forall$ is the universal quantifier.*

In order to prove this theorem, we need the following lemma.

**Lemma 4.3.1** *The abstract state machine $\mathcal{A}$ generated by the above algorithm bisimulates the original state machine $\mathcal{C}$ with respect to the set of atomic formulas $P$.*

**Proof**. Note that both $\mathcal{A}$ and $\mathcal{C}$ are deterministic machines, it is sufficient to prove the trace equivalence for the bisimulation relation between them, where a trace (path, computation) is a sequence of states starting from an initial state and the adjacent states are in the transition relation. Also the bisimulation relation $H$ over $\mathcal{A}$ and $\mathcal{C}$ is established by the Definition 4.3.1, from which it follows that $H$ is unique with a fixed property formula $\varphi$. As a result, to prove Lemma 4.3.1 it suffices to prove: (i) For each path in $\mathcal{C}$, there is a path in $\mathcal{A}$, and (ii) for each path in $\mathcal{A}$, there is a path in $\mathcal{C}$.

Note that $\mathcal{A}$ is the product of $\mathcal{C}$ and the constructed ASM $T_i$. By definition, the product machine $\mathcal{A}$ contains all the paths in $\mathcal{C}$. Therefore, for each path $\pi$ in $\mathcal{C}$ labeled by the atomic formulas $P$, we can find a path in $\mathcal{A}$, which proves (i).

Next, we give the proof of (ii) by induction on the structure of the atomic formula $p_i$.

Case A). $p_i$ is an atomic formula without $X$-operator. If $p_i$ has not an $\mathsf{X}$-operator, then the ASM $T_i$ shown in Figure 4.6 consists of 2 states $s'_0$ and $s'_1$. It is easy to see that the transition relation $R_{\mathcal{A}}$ is determined by $R_{\mathcal{C}}$ because the transition relation $R_{T_i}$ is complete and total. Let $\pi'$ be a path $t_0, t_1, \ldots$ in $\mathcal{A}$, where

$$(t_i, t_{i+1}) = \{(s'_i, s_i), (s'_{i+1}, s_{i+1}) \mid L(s_i) \cap L(s'_i) = L(s'_i) \text{ and }$$
$$L(s_{i+1}) \cap L(s'_{i+1}) = L(s'_{i+1}) \text{ and } (s_i, s_{i+1}) \in R_{\mathcal{C}}\}.$$

There is no restriction on the transition relation over $s'_i$ and $s'_{i+1}$ in the above formula, which implies that there is a path $\pi$ in $\mathcal{C}$ such that $\pi = \{s_0, s_1, \ldots\}$. Therefore, for every path in $\mathcal{A}$, we can find a path in $\mathcal{C}$.

Case B). $p_i$ is an atomic formula with $X$-operators and following operator $G$ or in $p_i\mathsf{U}q$. In this case, the constructed ASM is shown in Figure 4.7, where the number of states are determined by the number of $\mathsf{X}$ and all the states labeled with $\{p_i\}$ except two states labeled with $\{p_i\}$ and $\{\neg p_i\}$, respectively. Let $\pi'$ be a path $\{t_0, t_1, \ldots\}$ in $\mathcal{A}$, where $t_i = (s_i, s'_i)$, $s_i \in \mathcal{C}$, $s'_i \in T_i$, and $L(s_i)|_P = L(s'_i)$. The labeling of $s_i$ is $L(s_i)|_P = \{p_i \mid s_i, \psi \models p_i\}$, where $s_i, \psi \models p_i$ iff the LHS and RHS of $p_i$ have the same denotations with a interpretation $\psi$, and $\psi$-compatible assignment $\phi$, (see definition in the Section 3.3.3). One problem of the labeling is that we cannot check if $s_i, \psi \models p_i$ hold or not on those states appearing before the states assigned by $\mathsf{X}$-operator in the time structure since we have not the valuation of variable. In order not to affect the property checking, we assign the labels with $p_i$. From the facts that: the labeling of $s_i$ and $s'_i$ are the same for $i < Num(X)$ and the transition relation of $T_i$ over $s_i$, $i \geq Num(X)$ is total and complete. We can conclude that for every path $\pi' = t_0t_1t_2\ldots$ in $\mathcal{A}$, where $t_i = (s_i, s'_i)$, there is a path $\pi = s_0s_1s_2\ldots$ in $\mathcal{C}$. Thus, for every path $\pi'$ in $\mathcal{A}$, we can find a path $\pi$ in $\mathcal{C}$.

Case C). $p_i$ is an atomic formula with $X$-operator and following the temporal operator $F$ or $U$. The only difference between this situation and Case B) is labeling of the states. Instead of using $\{p_i\}$ to label other states, we use $\{\neg p_i\}$ (see Figure 4.6). Let $\pi'$ be a path $t_0, t_1, \ldots$ in $\mathcal{A}$, where $t_i = (s_i, s'_i)$. Following the assumption that the labeling of those states appearing before the states assigned by $X$-operator in the time structure are $\neg p_i$, we can conclude that for every path $\pi'$ in $\mathcal{A}$, we can find a path $\pi$ in $\mathcal{C}$ such that $L(\pi') = L(\pi)|_p$. Thus, in this situation, we prove ii).

Case D). The property consists of $n$ atomic formulas. By induction on the number of the ASMs and the definition 4.3.1, we obtain that for every path in $\mathcal{A}$, there is a corresponding path in $\mathcal{C}$.

This completes the proof of lemma 4.3.1. Q.E.D.

**Proof of Theorem 4.3.1**: From lemma 4.3.1, we obtain $\mathcal{A}$ is bisimuliar to $\mathcal{C}$ with respect to $P$. Theorem 4.3.1 is a direct consequence of theorem 3.1.1 with the condition of $\mathcal{A}$ bisimuliar to $\mathcal{C}$. Q.E.D.

## 4.4 An Example: Abstract Counter

We consider an example, called abstract counter [24], which sets a *Program Counter*(*pc*) according to the input command. The state machine has one input variable $Inst = \{inc1, inc2, load, no-op\}$ and 3 state variables $Y = \{state, pc, double\}$, where *double* is of the Boolean sort $B$; *state* is of concrete sort *State* with enumeration $\{st\_fetch$, $st\_inc1$, $st\_inc2$, $st\_load\}$; and *pc* is of abstract sort $S$. A graph representation of the state machine is shown in Figure 4.9. Depending on the input *Inst*, the counter *pc* will get a new value *loadin* (a generic constant of sort $S$), or increase by one, $inc(pc)$, where *inc* is an abstract function symbol of type $S \rightarrow S$, or keep the old value *pc*.

The initial states are described as the *directed formula* $F_I$: $state = st\_fetch \wedge double = 0 \wedge pc = pc_0$, where $pc_0$ is a generic constant of sort $S$. The transition relation can be described by the *directed formula* $F_T$:

$(state = st\_fetch \wedge Inst = no\_op \wedge state' = st\_fetch \wedge pc' = pc \wedge double' = double) \bigvee$
$(state = st\_fetch \wedge Inst = inc1 \wedge state' = st\_inc1 \wedge double' = 0 \wedge pc' = pc) \bigvee$
$(state = st\_fetch \wedge Inst = inc2 \wedge state' = st\_inc1 \wedge double' = 1 \wedge pc' = pc) \bigvee$
$(state = st\_inc1 \wedge double = 1 \wedge state' = st\_inc2 \wedge double' = 0 \wedge pc' = inc(pc)) \bigvee$
$(state = st\_inc1 \wedge double = 0 \wedge state' = st\_fetch \wedge double' = 0 \wedge pc' = inc(pc)) \bigvee$
$(state = st\_inc2 \wedge state' = st\_fetch \wedge double' = 0 \wedge pc' = inc(pc)) \bigvee$
$(state = st\_fetch \wedge Inst = load \wedge state' = st\_load \wedge double' = 0 \wedge pc' = pc) \bigvee$
$(state = st\_load \wedge state' = st\_fetch \wedge double' = 0 \wedge pc' = loadin)$.

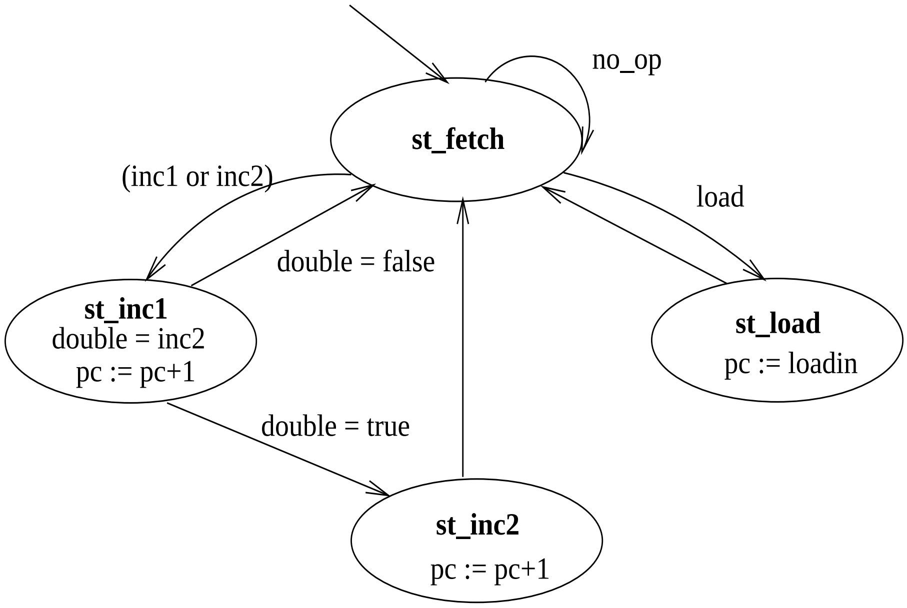



Figure 4.9: The control flow of an abstract counter

The property of the counter is

$$\begin{aligned} &\mathsf{G}(state = st\_fetch \ \wedge \ input = inc2) \rightarrow \\ &\mathsf{LET}\,(v = pc)\mathsf{IN}\,(\mathsf{X}(\mathsf{X}(\mathsf{X}(pc))) = (inc(inc(v))) \end{aligned} \tag{4.2}$$

To check the Property 4.2 on the ASM model $\mathcal{C}$ of *the abstract counter*, we follow the MDG LEC procedure to generate a product automaton as shown in Figure 4.10. The procedure consists of three steps: Step 1 uses the transformation algorithm to construct a state machine $M$ as follows: identify a set of atomic formulas $\{state = fetch\_st, input = inc2, \mathsf{LET}\,(v = pc)\mathsf{IN}\,(\mathsf{X}(\mathsf{X}(\mathsf{X}(pc))) = (inc(inc(v))\}$, construct circuit models $T$ for them and compose with $\mathcal{C}$ to get $\mathcal{A}$, and replace atomic formulas using propositions $b1$, $b2$ and $b3$ to generate the PLTL formula. Step 2 transforms the PLTL formula into a *generalized Büchi automaton* $\mathcal{B}$ for $\neg\mathsf{G}(b1 = \mathsf{T} \ \wedge \ b2 = \mathsf{T} \rightarrow X(X(X(b3 = \mathsf{T}))))$. Step 3 composes the *generalized Büchi automaton* with $M$ to generate the product machine.

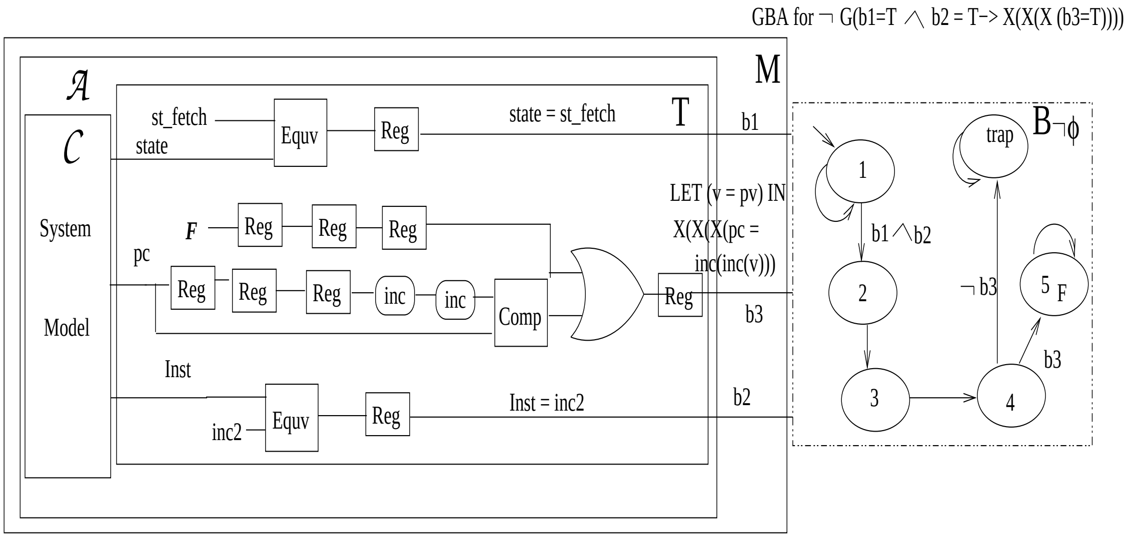



Figure 4.10: The illustration of MDG LEC on Abstract Counter with ${\mathcal{L}_{MDG}}^*$ formula $\mathsf{G}((state\ =\ fetch\_st\ \ \wedge\ \ input\ =\ inc2) \rightarrow \mathsf{LET}\ (v\ =\ pc)\ \mathsf{IN}\ \mathsf{X}(\mathsf{X}(\mathsf{X}(pc))) = inc(inc(pc)))$

## 4.5 Conclusion

In this chapter, we proposed a method of checking an ASM model $\mathcal{C}$ with respect to an ${\mathcal{L}_{MDG}}^*$ property formula $\forall\varphi$ based on a transformation algorithm. The transformation algorithm transforms $\mathcal{C}$ into a state machine $\mathcal{A}$ by building an ASM for each atomic formula and transforms $\varphi$ into a propositional formula $\phi$ by replacing each atomic formula from $\varphi$ with a Boolean variable. The verification of $\forall\varphi$ on $\mathcal{C}$ is converted into a verification of $\forall\varphi$ on $\mathcal{A}$. The ASM is modeled as a circuit using those constructing rules.

To verify $\varphi$ on the composed machine $\mathcal{A}$, we replace every atomic formula that labels $\mathcal{A}$ and in $\varphi$ with a proposition, obtain a state machine $M$ and a propositional formula $\phi$. To check if $M \models \forall\phi$, we first translate the LTL $\phi$ into a *generalized Büchi automaton* $\mathcal{B}_{\neg}\phi$ using the existing procedures, described in Section 3.4.2. We further compute the product of $M$ and $\mathcal{B}_{\neg}\phi$, where $M$ is viewed as a *generalized Büchi automaton* without any acceptance condition. Therefore, the verification of the $\varphi$ on the all computations ASM $\mathcal{C}$ is finally transformed into the Language Emptiness Checking (LEC) on the generated product automaton. We will explain the three algorithms using MDGs in the next chapter.

# Chapter 5

# MDG LEC Algorithms

In the previous chapter, we defined the specification language on structure of our MDG based language emptiness checking approach. In this chapter we describe the language emptiness checking algorithms for the generated $\omega$-automaton by our method, which is defined with an ASM and generalized Büchi acceptance condition, using the MDG operators. We propose three algorithms. The first two algorithms are adapted from two existing SCC-hull algorithms: EL/EL2 [32, 35, 48], and the third is based on the detection of the cycles. We also give several examples to illustrate these algorithms, and prove their correctness. We start this chapter by introducing some preliminaries.

## 5.1 Preliminaries

In this section, we describe some concepts, give the MDG representation of *generalized Büchi automaton.*

### 5.1.1 Generalized Büchi Automaton and MDG

To develop the checking language algorithms, we first define *generalized Büchi automaton* 's MDG representation. A *generalized Büchi automaton* consists of a finite state transition and an acceptance condition. The acceptance condition is defined as a collection of sets of states.

**Definition 5.1.1** *[79] A generalized Büchi automaton* $\mathcal{B} = (Q, Q_0, \delta, \mathcal{F})$ *over an alphabet* $\Sigma$ *is given by a finite set* $Q$ *of states, a non-empty set* $Q_0 \subseteq Q$ *of initial states, a transition relation* $\delta \subseteq Q \times \Sigma \times Q$*, and a collection of acceptance sets* $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$*, where* $F_i \subseteq Q$ *is an acceptance set or a fairness condition.*

A *run (path, computation)* $r$ of $\mathcal{B}$ over an $\omega$-word $\omega = a_0a_1\ldots \in \Sigma^\omega$ is an infinite sequence $r = q_0q_1\ldots$ of states $q_i \in Q$ such that $q_0 \in Q_0$ and $(q_i, a_i, q_{i+1}) \in \delta$ hold for all $i \geq 0$. The *run* $r$ is accepted iff for each acceptance set $F_i$, there exists a state $q_i \in F_i$ such that $q_i$ occurs infinitely many times in $r$. The *language* $\mathcal{L}(\mathcal{B}) \subseteq \Sigma^\omega$ is the set of $\omega$-words, over which there exist some accepted runs of $\mathcal{B}$. We say that a state $s$ can be reached from a state $t$ if there exists a word such that $s \xrightarrow{*} t$ from $s$ to $t$. A cycle is a set of states in which each state can be reached from itself. The *fair cycle* is a cycle $C$ reachable from the initial states and for each acceptance set $F_i$, $C \cap F_i \neq \emptyset$. The language of $\mathcal{B}$ is nonempty iff there is a fair cycle in it.

To represent a *generalized Büchi automaton* $\mathcal{B}$, we use *directed formulas* to describe the sets of states, the transition relation and the fairness conditions. A *directed formula* of type $U \to V$ is defined in Chapter 2. Let $X, Y, Y'$ be three sets of variables of concrete sort ($X$ denotes the set of alphabet, $Y$ denotes the set of state variables, and $Y'$ denotes the set of next-state variables). MDGs represent $\mathcal{B} = (Q, Q_0, \delta, \mathcal{F})$ as *directed formulas* , where $Q$, $Q_0$, and $F_i \in \mathcal{F}$ are represented as *directed formulas* over $Y$, and $\delta$ is represented as the *directed formula* of type $X \times Y \to Y'$. Next, we use an example to illustrate the above definitions.

**Example 5.1.1** *A generalized Büchi automaton* $\mathcal{B}$ *over alphabet* $\Sigma = \{p, q\}$ *is shown in Figure 5.1. To represent* $\mathcal{B}$ *using MDGs, we first define a set of variables ASM_variables* $=\{p, q, state, state'\}$*, where p and q are of concrete sort with enumeration* $\{\mathsf{T}, \mathsf{F}\}$*; state and state′ are of concrete sort with enumeration* $\{0, 1, 2, 3\}$*. We then define the following directed formulas to represent* $\mathcal{B}$*.*

- *The directed formula for the states is*

$$state = 0 \vee state = 1 \vee state = 2 \vee state = 3.$$

- *The directed formula as initial states $Q_0$ is*

$$state = 0.$$

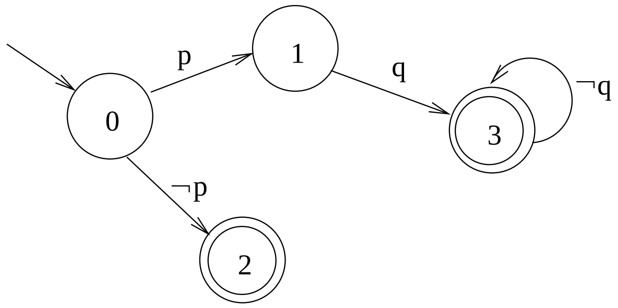
- *The directed formula of the transition relation is*

$$state = 0 \wedge p = \mathsf{T} \wedge state' = 1 \vee$$
$$state = 0 \wedge p = \mathsf{F} \wedge state' = 2 \vee$$
$$state = 1 \wedge q = \mathsf{T} \wedge state' = 3 \vee$$
$$state = 3 \wedge q = \mathsf{F} \wedge state' = 3 \vee .$$

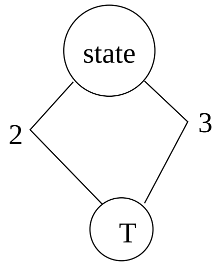
- *The directed formula of the acceptance set $F_1$ is*

$$state = 2 \vee state = 3.$$

*The MDG for this latter directed formula is shown in the right side of the Figure 5.1.*



A Genralized Buchi Automaton



MDG for fainess condition

Figure 5.1: The MDG representation of a GBA

### 5.1.2 Graph and SCC

A *generalized Büchi automaton* $\mathcal{B}$can be interpreted as a graph $G = \{V, E\}$, where states are viewed as vertices, transition relations as edges, fairness conditions as sets of vertices, and cycles as the Strongly Connected Components (SCCs) of $G$. The transition graph of $\mathcal{B}$ is defined as follows.

**Definition 5.1.2** *A transition graph is a pair $G = (V, E)$, with $V = \{v_1, \ldots, v_n\}$ a finite set of states, and $E \subseteq V \times V$ the set of edges. A path from $v_1 \in V$ to $v_k \in V$ is a sequence $(v_1, \ldots, v_k) \in V^+$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k$. If a path from $u$ to $v$ exists, we say that $u$ reaches $v$ (denoted by $u \longrightarrow v$). A Strongly Connected Component (SCC) of $G$ is a maximal set of states $U \subseteq V$ such that for each pair $(u, v) \in U$, $u$ reaches $v$. An SCC $U$ is trivial if the subgraph of $G$ induced by $U$ has no edges. A fair SCC is an SCC that intersects with all the acceptance sets.*

To illustrate the above defined concepts, we provide an illustration example.

**Example 5.1.2** *Given generalized Büchi automaton shown in Figure 5.2. The acceptance condition $\mathsf{F} = \{F_1, F_2, F_3\}$ consists of fairness conditions defined by the following sets*

$$F_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\},$$

$$F_2 = \{a, c, d, e, f\},$$

*and*

$$F_3 = \{11, 22, 33, 44, 55, 66, 77, 88\}.$$

*The generalized Büchi automaton has 8 SCCs and 5 fair SCCs. For example, the set $C = \{1, a, 11, temp3\}$ is a fair SCC. In fact, $C$ is an SCC because every state in it can be reached from the other states in $C$. $C$ is fair since $C \cap F_1 = \{1\}$, $C \cap F_2 = \{a\}$, and $C \cap F_3 = \{11\}$.*

## 5.2 MDG EL/EL2 Algorithms

The first two MDG based LEC algorithms have been adapted from two existing SCC-hull algorithms: EL [32] and EL2 [48, 35]. Due to the appearance of the abstract variables, no backward operators are available in the MDG package. Also the forward operators $ES_i$ and $EY$ cannot be implemented using MDGs because no conjunction of two MDGs with the same primary abstract variables is possible. Thus, SCC-hull algorithms cannot be directly used in MDGs. However, we implanted them into the MDG package and developed an MDG implementation by adaptations. This section

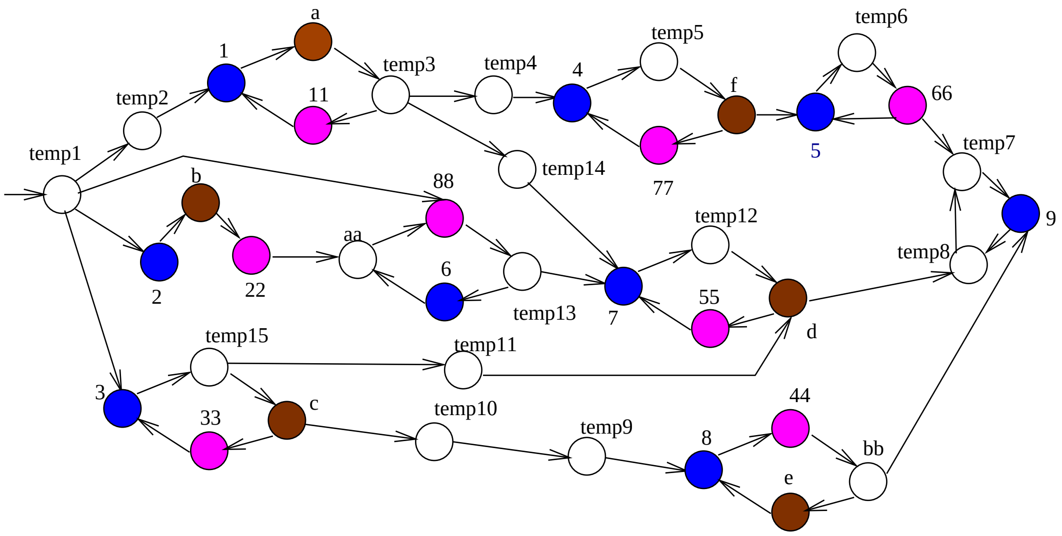



Figure 5.2: An example of a GBA

first introduces the EL and EL2 algorithms, then proposes respective version based on MDGs. The correctness of the adaption is also proved in this section.

EL and EL2 can be viewed as instances of the GSH algorithm [70, 73], which is a generalized SCC-Hull algorithm. In the following, we describe the GSH algorithm and present EL and EL2 as two instantiations.

### 5.2.1 Generic SCC Hull Algorithm

The language emptiness checking algorithm of a *generalized Büchi automaton* $\mathcal{B}$ is equivalent to the detection of the existence of fair SCC of its transition graph $G = (V, E)$. The computation of fair SCCs on $G$ can be implemented by an implicit enumeration technique [80], which is based on the observation that the SCC containing a state $v$ is the set of states with both a path to $v$ and a path from $v$. Several symbolic algorithms use the breadth-first search and compute a set of states that contains all the fair SCCs, called $SCC-Hull$, without enumerating them. An *SCC-hull* [32, 48, 75, 53] algorithm returns an empty set when there is no fair SCC and computes an SCC-hull otherwise. The Generic SCC-Hull (GSH) algorithm is a generalized algorithm, of which most of these algorithms can be instanced as special cases [73].

To describe the GSH algorithm on a *generalized Büchi automaton* $\mathcal{B} = (Q, Q_0, \delta, \mathcal{F})$, where $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$, we use the following denotations. Let

$$T_P = \{ES_1, \cdots, ES_m, EY\}$$

be a set of forward (past-tense) operators over $V$, defined by $\mu$-calculus (see Chapter 3), where $ES_i$ is defined as

$$\lambda Z.E\ Z\ S(\ Z\ \wedge\ F_i)$$

and $EY$ is defined as

$$\lambda Z.Z \wedge EYZ.$$

Similarly, let

$$T_F = \{EU_1, \cdots, EU_m, EX\}$$

be a set of backward (future-tense) operators over $V$, where $EU_i$ is defined as

$$\lambda Z.E\ Z\ U(\ Z\ \wedge\ F_i)$$

and $EX$ is defined as

$$\lambda Z.Z \wedge EXZ.$$

We further denote $T_B = T_F \cup T_P$.

Using these notations, the GSH algorithm can be described as follows [70, 73]:

**Algorithm 5.2.1** *(GSH Algorithm [73])*

*Step 1) Calculate the set of states $Z$ reachable from the initial states.*

*Step 2) Fairly pick an operator $\tau$ from $T_P \cup T_F$. Apply $\tau$ to $Z$, and let $Z = \tau(Z)$.*

*Step 3) Check if $Z$ is a fixpoint of all the operators in $T_P \cup T_F$. If yes, stop; otherwise, go to Step 2.*

**Remark 5.2.1** *In Step 2, "fairly pick" means the operator $\tau$ is selected under the following restriction: An operator $ES$ (or $EU$) cannot be selected again unless other operators in $T_F$ (or $T_P$) make changes to $Z$, and the operator $EY$ (or $EX$) cannot be selected again if it makes no change to $Z$ unless other operators in the set $T_F$ (or $T_P$) make changes to $Z$.*

The GSH algorithm first computes the set of reachable states from the initial states, and then recurrently removes the states that cannot be reached from the fair SCCs and those that cannot reach the fair SCCs until a fixpoint is reached. The forward operators in $T_P$ remove the states that cannot be reached from the fair SCCs, where $ES_i$ removes the states that cannot be reached from the accepting set $C_i$ within the current set, and $EY$ deletes the set of states that cannot be reached from a cycle within the set. The backward operators remove the states that cannot reach any fair SCCs, where $EU_i$ removes the states that cannot reach the accepting set $C_i$ within the current set, and $EX$ deletes states that cannot reach a cycle within the set.

The soundness of the algorithm is expressed in the following theorem.

**Theorem 5.2.1** *([73]) The GSH algorithm returns the empty set if no fair SCC exists. If a fair SCC exists, it returns a set containing all states on a fair SCC.*

It is noted that by only using forward operator $T_P$, we can derive a set of states with a path to a fair SCC. That is, GSH can be instanced by using only forward operator as long as we ensure that each operator $\tau \in T_P$ is monotonic and downward [73].

### 5.2.2 EL/EL2 Algorithms

EL and EL2 are two specializations of GSH by giving their schedules. They can be described with the following schedules.

$$EL[32] : EU_1, EX, \ldots, EU_m, EX, EU_1, EX, \ldots$$

$$EL2[48,\ 35] : EU_1, EU_2, \ldots, EU_m, EX, \ldots, EX, EU_1, EU_2, \ldots$$

Upon replacing each future tense operator with its past tense counterpart, we get the following two instantiations. Their correctness follows from the proof of Theorem 5.2.1 (see [73]).

$$EL : ES_1, EY, \ldots, ES_m, EY, ES_1, EY, \ldots \tag{5.1}$$

$$EL2 : ES_1, ES_2, \ldots, ES_m, EY, EY, \ldots, ES_1, ES_2, \ldots \tag{5.2}$$

We first illustrate the EL algorithm on the *generalized Büchi automaton* used in Example 5.1.2.

**Example 5.2.1** *We first compute all reachable states from the initial state, which includes all states of* $\mathcal{B}$*.*

$$Z = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33, 44, 55, 66, 77, 88, a, b, c, d, e, f, temp1, temp2, temp3,$$

$$temp4, temp5, aa, bb$$

$$temp6, temp7, temp8, temp9, temp10, temp11, temp12, temp13, temp14, temp15\}.$$

*Applying the EL algorithm on* $\mathcal{B}$*, we obtain the following result.*

$$Z_1^1 = ES_1(Z) = Z - \{temp1, temp2\}$$

$$Z_2^1 = EY(Z_1^1) = Z - \{temp1, temp2, 2\}$$

$$Z_3^1 = ES_2(Z_2^1) = Z - \{temp1, temp2, 2\}$$

$$Z_4^1 = EY(Z_3^1) = Z - \{temp1, temp2, 2, b\}$$

$$Z_5^1 = ES_3(Z_4^1) = Z - \{temp1, temp2, 2, b\}$$

$$Z^2 = EY(Z_5^1) = Z - \{temp1, temp2, 2, b, 22\}$$

*since* $Z \nsubseteqq Z^2$*, we continue the iterative computation, and get the following results.*

$$Z_1^2 = ES_1(Z^2) = Z - \{temp1, temp2, 2, b, 22, \}$$

$$Z_2^2 = EY(Z_1^2) = Z - \{temp1, temp2, 2, b, 22\}$$

$$Z_3^2 = ES_2(Z_2^2) = Z - \{temp1, temp2, 2, b, 22, aa, 6, 88, temp13\}$$

$$Z_4^2 = EY(Z_3^2) = Z - \{temp1, temp2, 2, b, 22, aa, 6, 88, temp13\}$$

$$Z_5^2 = ES_3(Z_4^2) = Z - \{temp1, temp2, 2, b, 22, aa, 6, 88, temp13\}$$

$$Z^3 = EY(Z_5^2) = Z - \{temp1, temp2, 2, b, 22, aa, 6, 88, temp13\}$$

*since* $Z^2 \not\subseteq Z^3$*, we begin the third iteration, where we get* $Z_1^3 = ES_1(Z_1^3) = EY(Z_1^3) = ES_2(Z_1^3) = ES_3(Z_1^3) = Z - \{temp1, temp2, 2, b, 22, aa, 6, 88, temp13\}$*, thus we get a fixpoint. The algorithm terminates with returning a SCC-hull.*

We then illustrate the $EL2$ algorithm on the same *generalized Büchi automaton* used in Example 5.1.2 to show the difference with $EL$.

**Example 5.2.2** *We first compute all reachable states from the initial state, which includes all states of* $\mathcal{B}$*.*

$$Z = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33, 44, 55, 66, 77, 88, a, b, c, d, e, f, temp1, temp2, temp3,$$

$$temp4, temp5, aa, bb$$

$$temp6, temp7, temp8, temp9, temp10, temp11, temp12, temp13, temp14, temp15\}.$$

*Applying the EL2 algorithm on* $\mathcal{B}$*, we obtain the following results.*

$$Z_1^1 = ES_1(Z) = Z - \{temp1, temp2\}$$

$$Z_2^1 = ES_2(Z_1^1) = Z - \{temp1, temp2, 2, \}$$

$$Z_3^1 = ES_3(Z_2^1) = Z - \{temp1, temp2, 2, b\}$$

$$Z_4^1 = EY(Z_3^1) = Z - \{temp1, temp2, 2, b, 22\}$$

$$Z^2 = Z_5^1 = EY(Z_4^1) = Z_4^1$$

*Thus, we get a fixpoint of the EY operator. Since* $Z \not\subseteq Z^2$*, we begin the second iterative computation, where we get*

$$Z_1^2 = ES_1(Z^2) = Z - \{temp1, temp2, 2, b, 22\}$$

$$Z_2^2 = ES_2(Z_1^2) = Z - \{temp1, temp2, 2, b, 22, aa, 8, 6, temp13\}$$

$$Z_3^2 = ES_2(Z_2^2)$$

$$Z_4^2 = EY(Z_3^2)$$

$$Z^3 = Z_4^2$$

*Thus, we reach the fixpoint of the EY operator. Since $Z^2 \nsubseteq Z^3$, we start the third iterative computation, where no changes are made by every operator. Thus, the algorithm terminates with returning an SCC-hull.*

### 5.2.3 MDG EL/EL2 Algorithms

The EL and EL2 algorithms work very efficiently [35]; however, it cannot be directly implanted in the MDG package because MDGs do not have backward operators nor conjunction operation for MDGs with the same abstract variables. To overcome these problems, we propose new algorithms derived from the EL/EL2 algorithms in this subsection. We first discuss the derivation and its proof, then propose the MDG based implementations.

#### MDG EL/EL2 Algorithms

The operations of $ES_i$ and $EY$ are built on the conjunction computation of sets of states. However, there is no conjunction operation for MDGs with the same abstract primary variables, we cannot implement the operators $ES_i$ and $EY$ in the MDG package. [73] also concluded if a set $Z$ of states is **forward-closed**, i.e., $Z = EP\ Z$, then $Z \geq EY\ Z$ and $EY\ Z$ is forward-closed. A deeper observation reveals that under the assumption of $Z$ being forward-closed, the operators $ES_i$ can be replaced by $EP_i$, and the operator $\lambda Z.Z \wedge EY\ Z$ can be replaced by $\lambda Z.EY\ Z$.

Following this observation, we propose MDG based EL/EL2 algorithms using following schedules:

$$EL : EP_1, EY, \ldots, EP_m, EY, EP_1, EY, \ldots$$

$$EL2 : EP_1, EP_2, \ldots, EP_m, EY, EY, \ldots, EP_1, EP_2, \ldots$$

where

$$EY = \lambda Z.EYZ$$

and

$$EP_i\ Z\ = \lambda Z.(E \text{ true } S(Z \wedge F_i)). \tag{5.3}$$

Note that in the above definition $EP_i$ still contains a conjunction operation $Z \wedge F_i$. However, since $Z$ and $F_i$ do not have the same abstract variables, this conjunction operation is feasible with MDGs. Recall that $F_i$ does not use any abstract variables.

The correctness of the MDG based EL/EL2 algorithms can be proved by the following theorem, which implies that the adaptations are correct.

**Theorem 5.2.2** *If a set of states $Z$ is forward-closed, then (i) $\lambda Z.Z \wedge EYZ$ can be replaced by $\lambda Z.EYZ$; (ii) the operators $ES_i$ can be replaced by $EP_i$.*

**Proof**. To prove (i), it suffices to show that $\lambda Z.\ Z \wedge EY(Z) = \lambda Z.EY(Z)$ and $\lambda Z.EY(Z)$ is forward-closed. In fact, since $Z$ is forward-closed, $Z$ can be expressed as

$$Z = Z \vee EY(Z) \vee \cdots \vee EY^j(Z) \vee \cdots, \tag{5.4}$$

implying $EY^j(Z) \leq Z, \forall j$. Therefore, we get

$$\lambda Z.Z \wedge EY(Z) = \lambda Z.EY(Z). \tag{5.5}$$

On the other hand, letting $EY(\cdot)$ operate on the right side of (5.4) yields

$$EY(Z) = EY(Z) \vee EY^2(Z) \vee \cdots = EP(EY(Z)),$$

which implies that $EY(Z)$ is forward-closed and proves (i).

Now, we proceed to prove (ii). By definition, to prove (ii) it suffices to show

$$EP_i(Z) = EP(ES_i(Z)) = ES_i(Z). \tag{5.6}$$

Note that

$$\begin{aligned}&ES_i(Z) = \lambda Z.EZ\ S\ (Z \wedge F_i) = \lambda Z.\mu y.(Z \wedge F_i) \vee (Z \wedge EYy)\\ &= \lambda Z.\bigvee_{j>0} \phi_Z^j(false),\end{aligned}$$

where $\phi_Z(false) = (Z \wedge F_i) \vee (Z \wedge EY(y))|_{y=false}$ and $\phi_Z^j(y) = \phi_Z(\phi_Z^{j-1}(y))$. So, to calculate $ES_i(Z)$ we need to compute the value of $\phi_Z^j(false), \forall j > 0$. A direct computation gives

$$\phi_Z^1(false) = (Z \wedge F_i) \vee (Z \wedge EY(false)) = (Z \wedge F_i)$$

$$\begin{aligned}&\phi_Z^2(false) = \phi_Z(\phi_Z(false)) = (Z \wedge F_i) \vee (Z \wedge EY(Z \wedge F_i))\\ &= (Z \wedge F_i) \vee EY(Z \wedge F_i).\end{aligned}$$

The last equation is due to the fact that $EY(Z \wedge F_i) \subseteq EY(Z) \subseteq Z$ since $Z$ is forward-closed.

Inductively, we obtain

$$\phi_Z^j(false) = (Z \wedge F_i) \vee EY(Z \wedge F_i) \vee EY^2(Z \wedge F_i) \vee \cdots \vee EY^{j-1}(Z \wedge F_i),$$

from which it follows that $\phi_Z^j(false)$ is increasing with respect to $j$. Therefore, we have

$$\begin{aligned}&\bigvee_{j>0} \phi_Z^j(false) = (Z \wedge F_i) \vee EY(Z \wedge F_i) \vee \cdots EY^{j-1}(Z \wedge F_i) \cdots\\ &= EP(Z \wedge F_i)\end{aligned} \tag{5.7}$$

Moreover, since $EP(Z) = EP(EP(Z))$, we obtain

$$EP(ES_i(Z)) = EP(EP(Z \wedge F_i)) = EP(Z \wedge F_i) = ES_i(Z) \tag{5.8}$$

Combining (5.7) and (5.8) completes the proof of (ii). Q.E.D.

### MDG Operators

To implement the MDG EL/EL2 algorithms, we first present the needed MDG operators.

### EY (image) - Relational Product (Relp)

Given a graph $G = \{V, E\}$ and a set of states $Z \subseteq V$, $EY$ operating on $Z$ yields a set of states $EY(Z) = \{t \in V \mid \exists (s,t) \in E, s \in Z\}$.

In the MDG package, the operation *Relational Product* (RelP) can be used to compute the *image* of the set of abstract states $Z$. RelP takes as inputs a set of MDGs $P_i, 1 \leq i \leq n$, of type $U_i \rightarrow V_i$ (for $1 \leq i \leq n$ and $i \neq j$, $V_i$ and $V_j$ must not have any abstract variables in common), a set of variables $E$ to be existentially quantified, and a renaming substitution $\eta$. It produces an MDG $R = Relp(\{P_i\}_{1 \leq i \leq n}, E, \eta)$ such that $\models R \Leftrightarrow (((\exists E)(\wedge_{1 \leq i \leq n} P_i)) \cdot \eta)$. The result is obtained by computing the conjunction of the $P_i$, existentially quantifying the variables in $E$, and applying the renaming substitution $\eta$ (see Chapter 2).

Given the ASM $D = (X, Y, O, I, T, F_O)$, where $X$, $Y$, and $O$ are the input, state, and output variables, respectively; $I$ is an MDG of type $U \rightarrow Y$; $T$ is an MDG of type $X \cup Y \rightarrow Y'$; and $F_O$ is an MDG of type $X \cup Y \rightarrow O$. We define $EY\ Z$ as $Relp(\{Input,\ Z, T\}, (X \cup Y), Y' \rightarrow Y)$, where $Input$ is an MDG of type $U \rightarrow X$ representing the set of input vectors, $Z$ is an MDG of type $U \rightarrow Y$ representing the current set , and $T$ is the MDG of transition relation. The result would be an MDG of type $U \rightarrow Y$.

**Fixpoint checking operation: Pruning-by-Subsumption(PbyS)**

To check if a fixpoint is reached or not, it is sufficient to check if both $\zeta \subseteq Z$ and $Z \subseteq \zeta$, where $\zeta$ and $Z$ are MDGs representing two successive results of the computation.

Due to the appearance of abstract variables in MDGs, we cannot compute the difference of two sets of abstract states. To alleviate this problem, in the MDG package, PbyS ($\zeta$, Z) has been developed to approximate the logic difference of two sets of abstract states, represented by MDGs $\zeta$ and $Z$, by pruning the paths of $\zeta$ from $Z$. Let $Diff$ be the MDG representing the result of PbyS ($\zeta$, Z). Their relation can be written as

$$\mathrm{Set}^{\psi}(\zeta) \backslash \mathrm{Set}^{\psi}(Z) \subseteq \mathrm{Set}^{\psi}(Diff) \subseteq \mathrm{Set}^{\psi}(\zeta).$$

If $Diff = \mathsf{F}$, where $\mathsf{F}$ is an MDG for $\emptyset$, then $\mathrm{Set}^{\psi}(\zeta) \subseteq \mathrm{Set}^{\psi}(Z)$.

**EP: ReAn*(G, Z)**

Given a state transition $G = (S, S_0, R)$ and a set of states $Z$, the $EP$ operation mainly implements the computation of the set of all reachable states from $Z$. The $EP$ $Z$ algorithm is described as follows.

**Algorithm 5.2.2** *(EP(Z))*

```
begin
  Frontier := Z, Reached := Z;
  loop {
    Image := EY(Frontier);
    Frontier := Image - Reached;
    Reached := Reached + Frontier;
  } until (Frontier = ∅)
 return Reached;
end
```

The $EP$ $Z$ algorithm first assigns the initial value of reachable states $Reached$ as $Z$, and the initial value of Frontier set $Frontier$ as $Z$, then iteratively adds the newly reached states $Frontier$ $(Image - Reached)$ to all the reached states $Reached$ $(Reached + Frontier)$, where $Image = EY(Frontier)$. The computation repeats until $Frontier = \emptyset$.

In the MDG package, the procedure ReAn(D,C) has been developed to implicitly enumerate all reachable states of $D$. During the procedure, ReAn also checks if an invariant condition $C$ holds at the output of every state (see Chapter 2). To develop an $EP$ algorithm using MDGs, we use a simplified version, ReAn* by removing the invariant checking from ReAn.

Given an ASM $D = (X, Y, O, I, T, F_O)$, the $ReAn^*$ algorithm used to compute all the states reachable from a set of states $Z$ is described as follows.

**Algorithm 5.2.3** *(ReAn*(G,Z))*

```
1  begin
2     Reached := Z, Frontier := Z, K := 0;
3     Do {
4       K := K+1;
5       Input := NewInputs(X,K);
6       Image := RelP({Input, Frontier, T }, X ∪ Y, Y' → Y);
8       Frontier := PbyS(Image,Reached);
9       Reached := Disj(Reached, Frontier);
10    } until (Frontier = ∅)
11   return R;
12 end
```

In the ReAn*(G,Z) procedure, $Input, Frontier, Image$, and *Reached* are MDGs representing the *input vectors*, *frontier set*, *image set* and the set of all reached states, respectively. In line 5, NewInputs(X,K) constructs a one-path MDG representing a conjunction of equations $x = u$, one for each abstract input variable $x \in X$, where $u \in U$ is a fresh variable depending on $K$. At line 6, RelP is used to compute $Image$ by taking the set of MDGs representing for input vector *Input*, $Frontier$ set and transition relation $T$, respectively. Line 8 uses PbyS to get the difference of the set $Frontier$ and the set *Reached*. Line 9 uses Disj to add set $Frontier$ to set *Reached*. The algorithm terminates once $Frontier$ = F, where F is an MDG representing an empty set.

**Remark 5.2.2** *The algorithm may be non-terminating when the ReAn*(G,Z) procedure reaches a set of states that cannot be represented by a finite MDG [86]. One solution to this problem is initial state generalization, that is, to generalize the set of initial states so as to obtain a larger set of reachable states that are representable by a finite MDG while still satisfy the condition [64].*

### MDG based EL/EL2 Algorithms

The MDG based EL and EL2 algorithms take as their arguments a state transition $G = (X, Y, I, T)$, where X and Y are sets of input and state variables, respectively.

$I$ is an MDG representing the set of initial states, $T$ is an MDG representing the transition relation, and $\mathcal{F} = \{F_1, ..., F_m\}$ is a set of fairness conditions.

The algorithms work as follows: First compute the set of states $Z$ reachable from the initial states $I$, and then iteratively apply the operators $EP_1, EP_2, \ldots, EP_m, EY, \cdots, EY$ (EL2) or the operators $EP_1, EY, EP_2, EY, \ldots, EP_m, EY$ (EL) until no changes to $Z$ can be made. Note that since the computation is decreasing, to determine if the fixpoint is reached, it suffices to check $Z^1 \subseteq Z^2$. If the fixpoint is empty, the algorithm returns "Verified"; otherwise, it returns "Failed".

The MDG EL algorithm can be described as follows, where $\zeta, F, Z, Input$ are sets and $K$ is an integer.

**Algorithm 5.2.4** *(MDG EL Algorithm* $(G = (X, Y, I, T),\ \mathcal{F} = \{F_1, F_2, \ldots, F_m\})$*)*

1 *begin*
2 $\zeta := \emptyset$, $K := 0$;
3 $Z :=$ ReAn$^*(G, I)$;
4 *Do* {
5 $\zeta := Z$;
6 *For each* $F \in \mathcal{F}$ {
7 $Z_F :=$ Conj$(Z,F)$;
8 $Z_R :=$ ReAn$^*(G, Z_F)$;
9 $K := K+1$;
10 $Input :=$ *NewInputs*$(X,K)$;
11 $Z :=$ RelP$(\{Input, Z_R, T\}, X \cup Y, Y' \to Y)$;
12 }
13 *If* $(Z =$ F$)$ *then return "Verified"*
14 } *until* (PbyS($\zeta$, Z) = F)
15 *return "Failed"*
16 *end*

In the above algorithm, line 3 computes the set of reachable states using $ReAn^*$ (G, I) to perform the operation $EP(I)$. Lines 4 - 14 represent the main body of the algorithm. Lines 7 - 8 compute the set of states reached from $Z \wedge F_i, i \leq m$.

Lines 9 - 11 compute the image of $Z$, where RelP essentially performs the operation $EY(Z)$. The operations are iteratively applied to $Z$ until no changes can be made. The fixpoint is reached if $\mathsf{PbyS}(\zeta, Z) = \mathsf{F}$. In case $\zeta \subseteq Z$, which means that there exists at least one fair SCC, and thus the property fails. Otherwise, we continue the computation until we get $Z = \emptyset$. In this case, the property succeeds.

The EL2 algorithm can be obtained by playing another schedule on the same set of forward operators. Let $\zeta, F, Z, I$ be sets and $K$ an integer; the following is the EL2 algorithm.

```
MDG EL2 Algorithm (G = (X, Y, I, T), 𝓕 = {F_1, F_2, ..., F_m})
 ζ:= ∅, K :=0;
 Z := ReAn*(G, I);
 Do {
    ζ := Z;
    For each F ∈ 𝓕 {
       Z_F := Conj(Z,F);
       Z_R := ReAn*(G, Z_F);
          }
     K := K+1;
     Input:=NewInputs(X,K);
     Z1 :=RelP({Input, Z_R, T}, X ∪ Y, Y' → Y);
     While (PbyS(Z, Z1) ≠ F) Do{
       Z := Z1;
       K:= K+1;
       Input:=NewInputs(X,K);
      Z1 :=RelP({Input, Z, T}, X ∪ Y, Y' → Y);
       }
    If (Z = F) then return "Verified"
 } until (PbyS(ζ, Z) = F)
 return "Failed";
```

Evidently, in the above algorithm there are some steps which appeared in MDG EL. For the sake of conciseness, in the following we only describe the steps that are not subsumed in MDG EL and omit the others since their descriptions are the same as that of MDG EL. Lines 10 - 18 apply $EY$ to $Z$ until the fixpoint is reached. It is obvious that $Z_1 \subseteq Z$ in each iteration since the operation Relp removes some states from the set $Z$. Therefore, to test if the fixpoint is reached it suffices to test $Z \subseteq Z_1$, which is done at line 13 using $\mathsf{PbyS}(Z, Z_1)$.

**Remark 5.2.3** *As pointed out in Remark 5.2.2,* ReAn*(G, Z) *may not terminate, which may lead to the non-terminating on the* **EL** *and* **EL2** *algorithms. We can apply the same approaches mentioned earlier to solve the problem.*

## 5.3 MDG Fair Cycle Detection Algorithm

The MDG EL/EL2 algorithms first compute all the reachable states, then remove those states that cannot be reached from a fair SCC using EP and EY operations. Combining these two steps into one signal procedure, we propose a Fair Cycle Detection (FCD) algorithm in this section. The FCD algorithm detects the existence of fair cycles in the process of computing reachable states. More specifically, we compute those states, that satisfy fairness conditions, on each path starting from a state in the set of reachable states. Instead of getting all the states satisfying the fairness condition on the path, we only compute the first one. Then we detect the existence of the fair cycles using the sets of states. To see it clearly, let us assume that there is only one path *trace* in the given state transition and one fairness condition $F$ in the acceptance condition. For this special case, FCD computes the state $s^1$ that satisfies $F$ and is the nearest to the initial state $s_0$ on *trace*. Then FCD computes the direct successor $t$ of $s^1$. After that, the FCD algorithm checks if $t = s_0$. If it is the case, the FCD terminates by returning "failed". If this is not the case, FCD continues computing $s^2$ that satisfies $F$ and is the nearest to $t$ on *trace*; and compute the next state of $s^2$. In the case there is no such state existing, the FCD algorithm terminates with returning "verified".

This section first describes the FCD algorithm. We then illustrate it by an example and analyze its correctness.

### 5.3.1 FCD Algorithm

The FCD algorithm detects the existence of fair cycles by computing the states satisfying the condition on each path; and checking the existence of cycles using those states. Given a transition relation $G = (X, Y, I, T)$ and a set of fairness conditions $\mathcal{F} = \{F_1, \ldots, F_m\}$. FCD first searches every path that starts from an initial state in $I$ to obtain a state that satisfies fairness condition $F_1$. These states compose a set $Z_1^1$. FCD then search every path that starts from a state in $Z_1^1$ to obtain a state that satisfies the fairness condition $F_2$. These states compose a set $Z_2^1$. FCD continues the similar computation until it gets $Z_m^1$. FCD then computes all direct successors $Z^2$ of $Z_m^1$ and tests if either $Z^2 = \emptyset$ or $Z^1 \subseteq Z^2$. If this is the situation, the FCD algorithm terminates and returns *verified* (*failed*) if $Z^2 = \emptyset$ ($Z^1 \subseteq Z^2$).

Next we summarize the above procedure as an algorithm.

**Algorithm 5.3.1** *(FCD Algorithm($G = (X, Y, I, T)$, $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$))*

*begin*
    $Z^1 = Z_0^1 := I,\ i := 1,\ j := 1;$
    *loop*
        *For each* $F_i \in \mathcal{F}\{$
        $Z_i^j := \{s | s$ *is on a path that starts from a state in* $Z_{i-1}^j$ *and satisfies* $F_i\};$
        $\}$
        $Z^{j+1} :=$ *next states of* $Z_m^j;$
        *If* $(Z^{j+1} = \mathsf{F})$ *then return "Verified";*
        *If* $Z^j \subseteq Z^{j+1}$ *then return "Failed";*
        $j = j + 1,\ i = 1,\ Z_{i-1}^j = Z^j;$
    *end loop*
*end*

We illustrate the FCD algorithm on a *generalized Büchi automaton* from Example 5.1.2.

**Example 5.3.1** *Assume an order of fairness condition* $F_1, F_2, F_3$*, FCD runs on* $\mathcal{B}$ *yielding the following results:*

$$Z^1 = Z^1_0 = \{temp1\},$$

$$Z^1_1 = \{1, 2, 3, 6, 7\},$$

$$Z^1_2 = \{a, b, c, d\},$$

$$Z^1_3 = \{11, 22, 33, 44, 55, 66, 77\},$$

$$Z^2 = \{1, aa, 3, bb, 7, 5, 4, temp7\}.$$

*Since* $Z^1 \nsubseteq Z^2$*, FCD starts the second iteration, yielding the following results:*

$$Z^2_1 = \{1, 3, 4, 5, 6, 7, 8, 9\},$$

$$Z^2_2 = \{a, c, d, e, f\},$$

$$Z^2_3 = \{11, 33, 44, 55, 66, 77\},$$

$$Z^3 = \{1, 3, bb, 7, 5, 4, temp7\}.$$

*Since* $Z^2 \nsubseteq Z^3$*, FCD starts the second iteration, producing the following results:*

$$Z^3_1 = \{1, 3, 4, 5, 7, 8, 9\},$$

$$Z^3_2 = \{a, c, d, e, f\},$$

$$Z^3_3 = \{11, 33, 44, 55, 66, 77\},$$

$$Z^4 = \{1, 3, bb, 7, 5, 4, temp7\}.$$

*Since* $Z^3 \subseteq Z^4$*, the FCD terminates and returns failed.*

In order to compute the set $Z_i^j$ of states from $Z_{i-1}^j$ used in Step 2 of the FCD algorithm, we propose a method based on a fixpoint computation. We first split $Z_i^j$ into two sets: one set $Z_{F_i}$ containing all states that satisfy the condition $F_i$, another set $Z_{\neg(F_i)}$ containing all states that do not satisfy the condition $F_i$. We then compute all states $Sg$ reachable from $Z_{\neg(F_i)}$ that do not satisfy $F_i$. We further compute the successors $S$ of $Sg$ and derive all states $Z_{\Sigma(\neg F_i)}$ that satisfy $F_i$, which further joins with $Z_{F_i}$.

The correctness of the algorithm can be proved by using the following lemma.

**Lemma 5.3.1** *For a transition graph $G = \{S, T\}$ of a state machine, where $S$ is the set of states and $T$ denotes the transition relation. Assume $Z^1 \subseteq S$ and $Z^2 \subseteq S$ are two sets of states, and $Z^2$ is reached from $Z^1$ in $n$ transition steps for some $n \geq 1$. If $Z^1 \subseteq Z^2$, then there must be a state $s \in Z^1$ such that there exists a cycle from $s \rightarrow s$.*

**Proof**. Since $Z^2$ is reached from $Z^1$, then for all $t' \in Z^2$, there must be a state $t \in Z^1$ such that $t \longrightarrow t'$ ($t$ reaches $t'$). We define a set $T = \{t | t \longrightarrow t' \text{ and } t' \in Z_2\}$. Obviously, $T \subseteq Z^1$. Since $Z^1 \subseteq Z^2$, $T \subseteq Z^2$. Consequently, every state $t \in T$ has an incoming edge. Let $m$ be the number of states in $T$. Consider a path starting from $t_0 \in T$ and having length $m+1$, denoted by $\mathcal{T} = t_0 t_1 \ldots t_m$. Since there are at most $m$ different states in $t_0, t_1, \cdots, t_{m-1}$, there must exist $i_1, i_2$ such that $t_{i_1} = t_{i_2}$. This proves Lemma 5.3.1. Q.E.D.

### 5.3.2 MDG FCD Algorithm

To implement the FCD algorithm using MDGs, we develop an algorithm *ReAn*$^{\#}$ to compute, of the state transition $G = (X, Y, I, T)$, all the states that are reachable from a set $Z$ of states and do not satisfy a fairness condition $F$.

**Algorithm 5.3.2** *(ReAn$^{\#}$(G,Z, F))*

*begin*

*Reached := Z; Frontier := Z; K := 0*

*Do* {

```
4        K := K+1;
5        Input := NewInputs(X,K);
6        Image := RelP({Input, Frontier, T}, X ∪ Y, Y′ → Y);
7        NewUnsatStates := PbyS(Image,F);
8        Frontier := PbyS(NewUnsatStates,Reached);
9        Reached := Disj(Reached, Frontier);
10   } until (Frontier = ∅)
11   return R;
12  end
```

The ReAn$^{\#}$ algorithm adds one more step (line 7) to the ReAn$^{*}$ algorithm described earlier. Line 7 uses PbyS to compute the new states that do not satisfy the condition $F$ in all the newly reachable states $Image$.

The MDG based FCD algorithm is implemented using the MDG operators: Conj, PbyS, RelP, Disj, and *ReAn$^{\#}$(G,Z, F)*. Assume the transition relation $G = \{X, Y, I, T\}$, in which $X$ and $Y$ are sets of input and output variables, $I$ is the initial state and $T$ is the transition relation. The acceptance condition $\mathcal{F} = \{F_1, \ldots, F_m\}$, where $F_i$ is the $i^{th}$ fairness condition (acceptance set). The MDG based FCD algorithm is shown as follows:

**Algorithm 5.3.3** *(MDG FCD Algorithm (G, $\mathcal{F}$) )*

```
begin{
     Z:= I, K := 0;
     loop
     Z-old:= Z;
         For F ∈ 𝓕 {
              Z_F := Conj(Z, F);
              Z_¬F := PbyS(Z, F);
              Sg := ReAn#(G,Z_¬F, F);
              K := K+1;
              Input := NewInputs(X,K);
              S := RelP({Input, Sg, T}, X ∪ Y, Y′ → Y);
```

12 $\quad\quad\quad Z_{\Sigma\neg F}$ := Conj$(S, F)$;
13 $\quad\quad\quad Z$ := Disj$(Z_{\Sigma\neg F}, Z_F)$;
14 $\quad\quad$ }
15 $\quad\quad$ *K := K+1;*
16 $\quad\quad$ *Input := NewInputs(X,K);*
17 $\quad\quad$ *Z:=* RelP$(\{Input, Z, T\}, X \cup Y, Y' \to Y)$;
18 $\quad\quad$ *if (Z =* F*) then return* 'Verified'*;*
19 $\quad\quad$ *Zv :=* PbyS*(Z-old,Z);*
20 $\quad\quad$ *if (Zv=* F*) then return* 'Failed'*;*
21 $\quad$ *end loop*
22 *end*

The algorithm begins with assigning the current state $Z$ with the initial states $I$. From lines 5 to 14, we compute the $Z_i^j$ for each fairness condition in the order of $F_1, \ldots, F_m$. To derive the states satisfying the acceptance condition $F$ along all paths, we distinguish the states satisfying $Z_F$ (line 6) and unsatisfying $Z_{\neg F}$(line 7). We compute all states $Z_{\Sigma\neg F}$ reachable from $Z_{\neg F}$ and satisfying $F$ (line 8 to 12); we finally join $Z_{\Sigma\neg F}$ and $Z_F$ together in line 13. The computation of $Z_{\Sigma\neg F}$ is done by a reachability analysis to compute all states $Sg$ reachable from $Z_{\neg F}$ and not satisfying $F$ (line 8), a computation of the direct successor $S$ of $Sg$ (line 9 -11) and a conjunction with $F$ (line 12). lines 15 -17 compute the direct successors $Z$ of $Z_m^j$ and line 18 checks if $Z$ is empty and returns "Verified" if it is the case. Otherwise, we check if $Z - old \subseteq Z$ (line 19 - 20). If this is the case, the algorithm returns "Failed"; Otherwise, we begin the next loop execution.

### 5.3.3 An Example: MinMax

We have introduced the MinMax [21] in Chapter 3 to illustrate the concept of ASM. Here, we illustrate our FCD algorithm on it. The ASM of the MinMax, shown in Figure 2.2, has 2 input signals: reset $r$ and input $x$, and 3 state signals: $c, m, M$. The machine works as the following: In *reseting state* $c = 1$, if the reset signal $r = 0$, then the machine goes into the *running state* ($c = 0$) and assigns the input values to

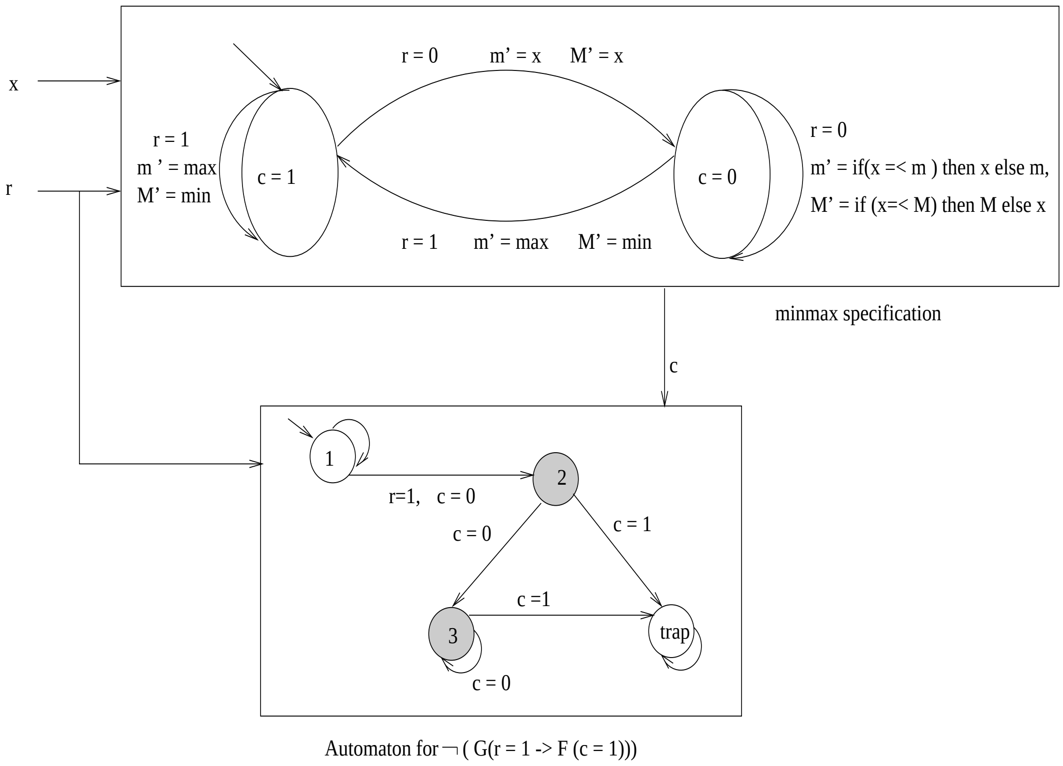



Figure 5.3: Product automaton for MinMax

$M$ and $m$; if the reset signal $r = 1$, the machine remains in the reseting state and assigns the initial values to $M$ and $m$, respectively. In the *running state* $c = 0$, if the reset signal $r = 0$, then the machine remains in state $c = 0$ and assigns $m$ and $M$ new values depending on the less or equal (*leq*) cross-operator outcome; if the reset signal $r = 1$, the machine goes to *reseting state* $c = 1$ and assigns the initial values to $M$ and $m$, respectively. One property of the MinMax ASM says: In any states, if the reset signal $r = 1$, the machine goes back to *reseting state* $c = 1$. An $\mathcal{L}_{MDG}{}^*$ formula $\phi$ to represent this specification is $G(r = 1 \ \rightarrow \ F(c = 1))$.

To verify $\phi$ on the MinMax ASM machine, we first generate a product automaton which is shown in Figure 5.3. Then, we use the FCD algorithm to compute its language. Figures 5.4 and 5.5 contain a collection of sample MDGs (a) to (l) generated throughout the execution of the FCD algorithm on this example. Initially, acceptance condition $\mathcal{F} = \{F_1\} = \{\{2,3\}\}$, where the fairness condition $F_1$ is represented by a *directed formula* $(aut\_state = 2) \vee (aut\_state = 3)$(Figure 5.4(b)); and $Z$ is the initial states described by a *directed formula* $(aut\_state = 1) \wedge (c = 1) \wedge (m = max) \wedge (M = min)$ (Figure 5.4(a)).

In the first iteration, we restore $Z$ into $Z - old$ and compute the new value of $Z$. For this purpose, we first compute all the states $Z_F$ that satisfy the condition $F$ (Figure 5.4(c)) and all states $Z_{\neg F}$that do not satisfy the condition $F$ $(aut\_state = 1) \wedge (c = 1) \wedge (m = max) \wedge (M = min)\} = Z$ ( Figure 5.4 (d)). Then, we compute all states $Sg$ (Figure 5.5 (e)) that do not satisfy the fairness condition $F$ using the reachability analysis algorithm ReAn$^{\#}$. We further compute the states $S$ (Figure 5.5 (f)) reachable from $Sg$ by one step, and obtain all the states $Z_{\Sigma\neg F}$ (Figure 5.5 (g)) that satisfy $F$ in $S$; and finally we get $Z$ by joining $Z_{\Sigma\neg F}$ with the satisfying states $Z_F$, $(aut\_state = 2) \wedge (c = 1) \wedge (m = max) \wedge (M = min)$.

Next, we update the value of $Z$ by its direct successor set (Figure 5.5 (h)). Then we proceed to check if $Z - old \subseteq Z$ or $Z = \emptyset$, and both of them fail. Thus, we start the second iteration, in which we get the following resulting MDGs: $Z_F$ is shown in Figure 5.5(i), $Z_{\neg F}$ shown in Figure 5.5(h), $Sg$ shown in Figure 5.5(j), $S$ shown in Figure 5.5(k), $Z_{\Sigma\neg F}$ and $Z$ are shown in Figure 5.5(l). Since $Z = \mathsf{F}$, the algorithm terminates with returning "Verified".

## 5.4 Conclusion

In this chapter, we presented three MDG based language emptiness checking algorithms: FCD, EL and EL2. The FCD algorithm implemented the reachability analysis and the fair cycle detection at the same time; the MDG EL/EL2 algorithms first computed the reachable states and then eliminated the states not reachable from the fair SCCs. The correctness of the algorithms were also proved, and several examples were used to illustrated them. We will test the efficiency of this method in the next chapter.

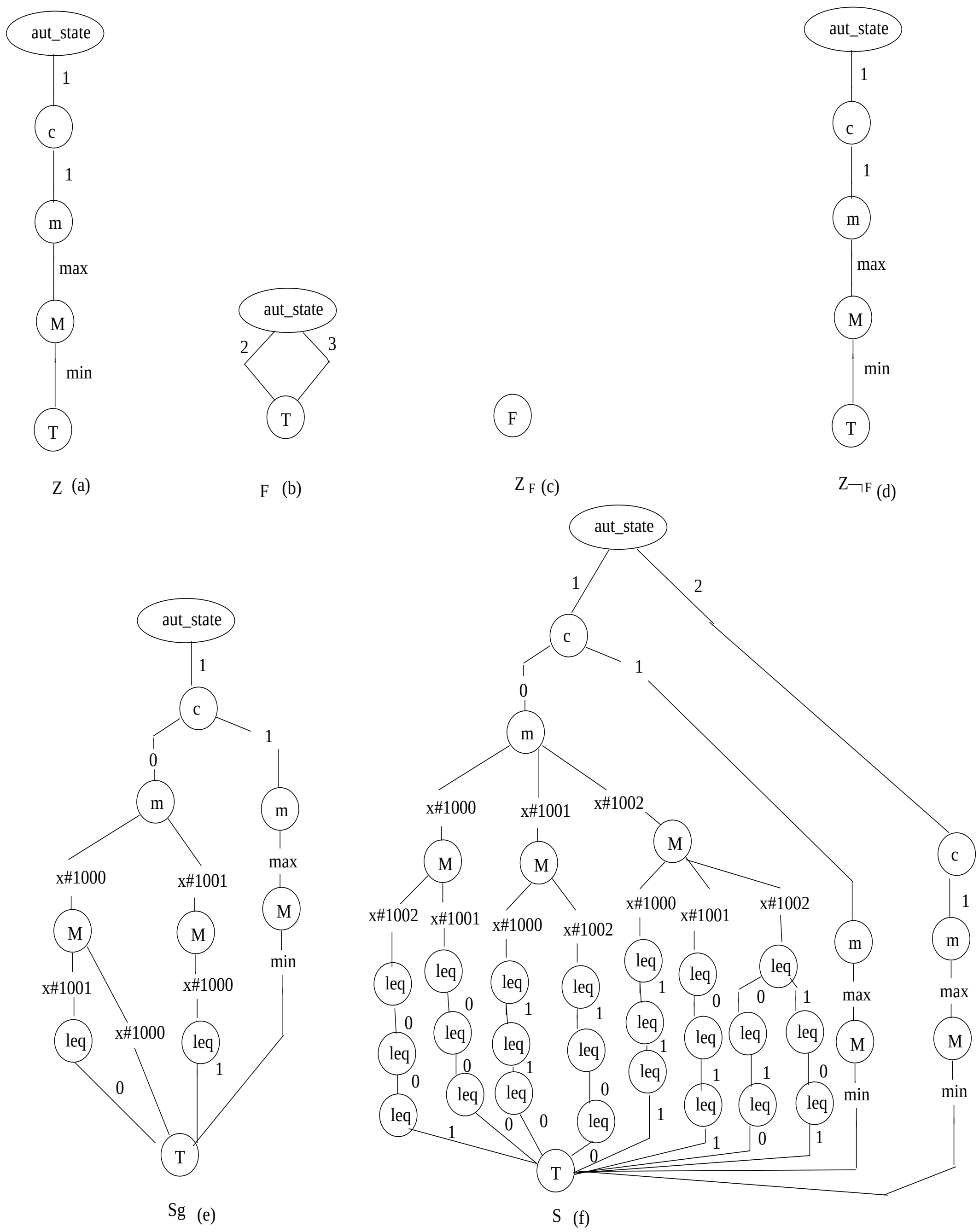



Figure 5.4: MDG FCD algorithm execution on MinMax

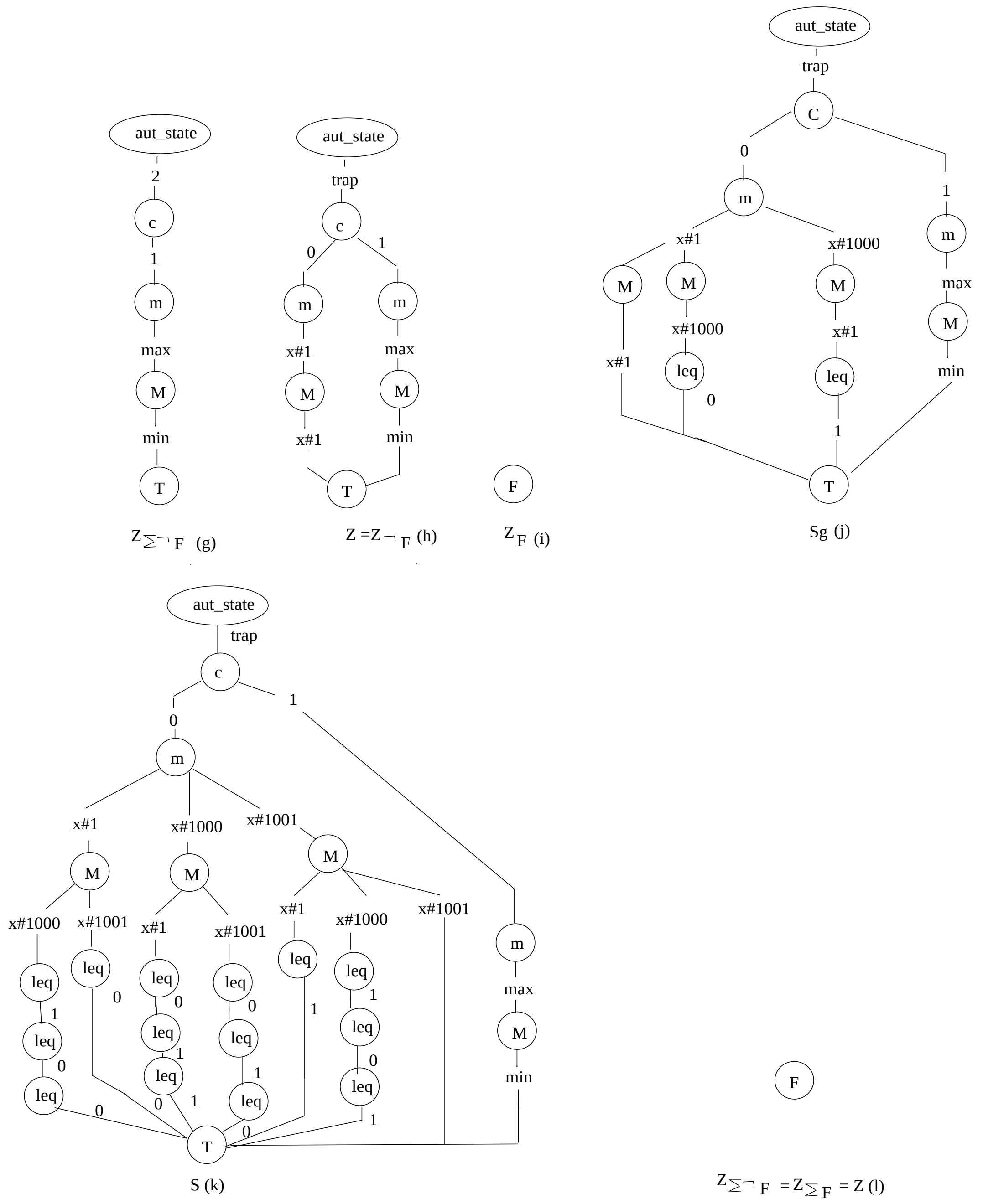



Figure 5.5: MDG FCD algorithm execution on MinMax(cont'd)

# Chapter 6

# Case Studies

This chapter presents two case study: an ATM switch fabric verification and an Island Tunnel Controller verification using the developed MDG LEC approaches.

## 6.1 ATM Switch Fabric

In this section we present the verification of a Fairisle $4 \times 4$ ATM (Asynchronous Transfer Mode) switch fabric. The device was in use for real applications in the Cambridge Fairisle network [47], designed at the Computer Laboratory of the University of Cambridge. We first introduce the Fairisle ATM Switch fabric model. We then give its hardware description. Finally, we explain the property checking of the ATM model.

### 6.1.1 System Description

The $4 \times 4$ Fairisle switch consists of three types of components: the input port controllers, the output port controllers and the switch fabric, as shown in Figure 6.1. An (Fairisle) ATM cell consists of a header (one-byte tag containing routing information as shown in Figure 6.2) and a fixed number of octets. Cells are switched from the input ports to the output ports according to their headers.

The behavior of the switch is cyclical. In each cycle or frame, the input port

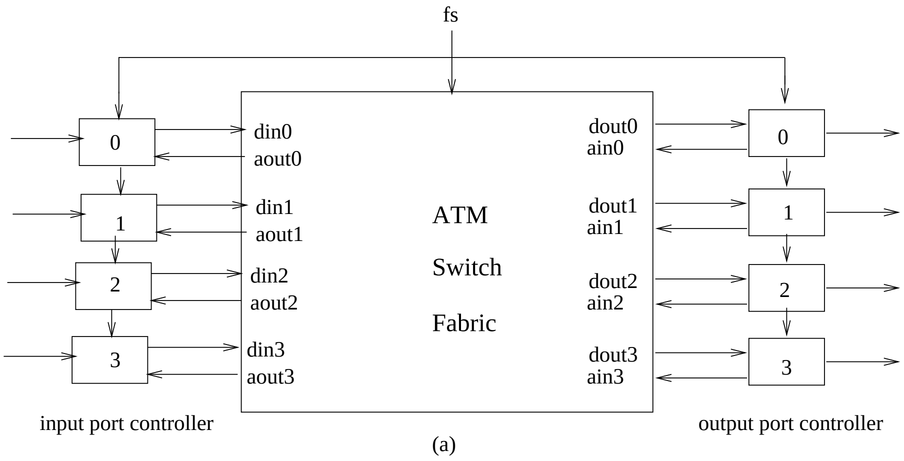



Figure 6.1: The Fairisle ATM Switch

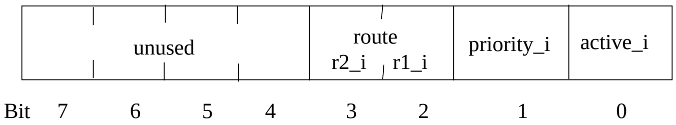



Figure 6.2: The header (routing tag) of a Fairisle ATM cell

controllers synchronize the incoming data cells, append the control information in the front of the cells, and send them to the fabric. The fabric waits for cells to arrive, strips off the headers, arbitrates between cells destined to the same port, sends successful cells to the appropriate output port controllers, and passes acknowledgments from the output port controllers to the input port controllers.

If different port controllers inject cells destined for the same output port controller (as indicated by the route bits in the header) into the fabric at the same time, then only one will succeed. The others must retry later. The routing tag also includes priority information (priority bit) that is used by the fabric for arbitration, which takes place in two stages. High priority cells are given precedence before the remaining cells. The choice between cells of the same priority is made on a round-robin basis. The input controllers are informed of whether their cells were successful using acknowledgment signals. The fabric sends a negative acknowledgment to the unsuccessful input ports, but passes the acknowledgment from the requested output port to

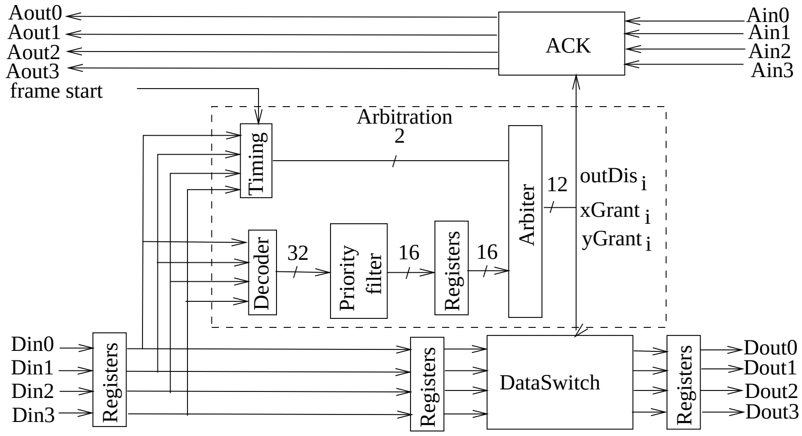



Figure 6.3: The block diagram of the Fairisle ATM Switch Fabric

the successful input port. The port controllers and the switch fabric all use the same clock, hence bytes are received synchronously on all links. They also use a higher-level cell frame clock - the frame start signal $fs$. It ensures that the port controllers inject data cells into the fabric synchronously so that the routing tags arrive at the same time. If no input port raises the active bit throughout the frame then the frame is inactive - no cells are processed. Otherwise it is active.

Figure 6.3 shows a block diagram of a $4 \times 4$ switch fabric. The inputs to the fabric consist of the cell data lines, the acknowledgments that pass in the reverse direction, and the frame start signal $fs$ which is the only external control signal. The outputs consist of the switched data, and the switched and modified acknowledgment signals. The switch fabric is composed of an arbitration unit, an acknowledgment unit and a dataswitch unit. The arbitration unit reads the headers, makes arbitration decisions when two or more cells are destined for the same output port, passes the result to the other modules using grant signals and controls the timing of the other units using output disable signals. The dataswitch performs the actual switching of data from an input port to an output port according to the most recent arbitration decision. The acknowledgment unit passes appropriate acknowledgment signals to the input ports. Negative acknowledgments are sent until arbitration is completed.

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

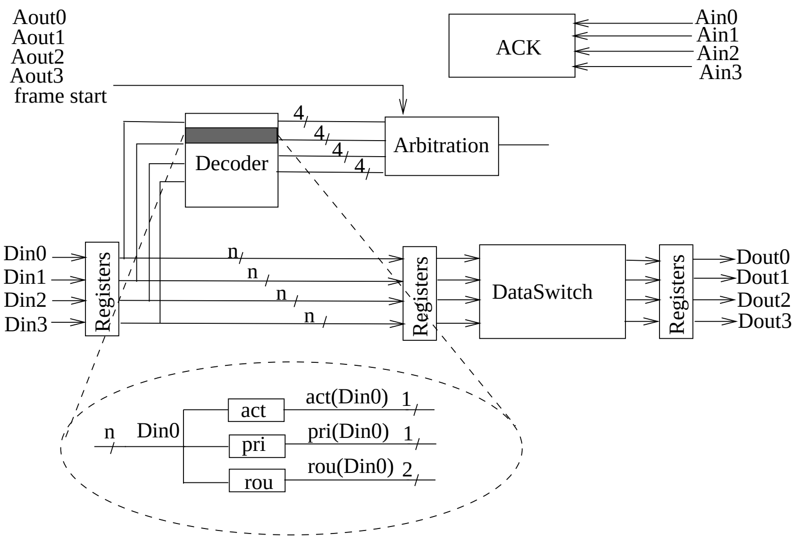



Figure 6.4: Model abstraction of the Switch Fabric

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

### 6.1.2 System Model

Figure 6.4 described the RTL implementation of the fabric based on the gate-level description by describing the dataswitch using multiplexors instead of logic gates [60]. The data signals $Din_i$ / $Dout_i$ (i = 0, 1, 2, 3) are modeled as $n$-bit words and assigned an abstract sort $wordn$. The control fields contained in the cell headers, i.e. active, priority and route bits, are extracted from the abstract data signals using cross-operators (functions) $act$, $pri$ of type $wordn \rightarrow bool$ and $rou$ of type $wordn \rightarrow word2$ (word2 = 0, 1, 2, 3) respectively. The ASM model is thus obtained by compiling the abstract description of the RTL implementation.

### 6.1.3 Verification

The switch fabric interface with the port controllers can be modeled as a 14-state finite state machine, as shown in Figure 6.7. Circles denote the environment states ($env\_st$) and arrows denote state transitions. States 1 to 5 are related to the

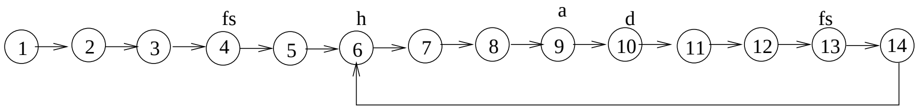



Figure 6.5: The ATM Switch Fabric environment

initialization of the fabric. States 6 to 14 denote the cyclic behavior of the fabric. $fs$, $h$, $a$ and $d$ above the states designate that the frame start signal, the header of an active cell, the acknowledgment and the data are, respectively, generated in the states. After receiving the cell header, the fabric needs 3 clock cycles to send acknowledgment signals and 4 clock cycles to send data.

In the following we discuss 5 sample properties that we verified on the RTL implementation.

**P1:** After the switch starts, the default value '0' is put on the acknowledge output port 0 until the fabric makes a decision.

$G(((env\_st = 5) \wedge (env\_st = 6) \wedge (env\_st = 7) \wedge (env\_st = 8) \wedge (env\_st = 14)) \rightarrow (aout0 = 0)));$

**P2:** After the fabric makes a decision, valid acknowledge signal is put on the acknowledge output port 0.

$G(((env\_st = 9) \wedge (env\_st = 10) \wedge (env\_st = 11) \wedge (env\_st = 12) \wedge (env\_st = 13)) \rightarrow ((aout0 = ain0) \vee (aout0 = ain1) \vee (aout0 = ain2) \vee (aout0 = ain3)));$

**P3:** If the input port 0 chooses output port 0, it will eventually succeed to pass the cell to (get the acknowledgment from) output 0.

$G(((r1\_0 = 0) \wedge (r2\_0 = 0) \wedge (active\_0 = 1) \wedge (env\_st = 6)) \rightarrow F(aout0 = ain0));$

**P4:** If two inputs request the same output ports, eventually, these two inputs are all passed to (acknowledgment from) that output.

$G(((r1\_0 = 0) \wedge (r2\_0 = 0) \wedge (r1\_1 = 0) \wedge (r2\_1 = 0) \wedge (active\_0 = 1) \wedge (active\_1 = 1) \wedge (env\_st = 6)) \rightarrow F(aout0 = ain0) \wedge F(aout0 = ain1)));$

Table 6.1: Experimental results with MDG LEC using EL and EL2 for ATM

| | EL (n-bit) | | | EL2 (n-bit) | | |
|---|---|---|---|---|---|---|
| | Time (sec) | Memory (MB) | # MDG Nodes | Time (sec) | Memory (MB) | # MDG Nodes |
| P1 | 300 | 30 | 100325 | 312 | 32 | 100325 |
| P2 | 288 | 32 | 100382 | 280 | 32 | 100382 |
| P3 | 254 | 41 | 129782 | 338 | 52 | 157739 |
| P4 | 314 | 42 | 131132 | 346 | 54 | 159325 |
| P5 | 340 | 47 | 139255 | 329 | 46 | 131775 |

**P5:** If the input port 0 chooses output port 0 with priority bit set and no other port set their priority bits, then after 5 clock the value of $dout_0$ will be $din_0\_delay$, which denotes the data input $din_0$ of 4 clock cycles earlier.

$G(((r1\_0 = 0) \wedge (r2\_0 = 0) \wedge (active\_0 = 1) \wedge (env\_st = 6)) \wedge (priority\_0 = 1) \wedge (priority\_1 = 0) \wedge (priority\_2 = 0) \wedge (priority\_3 = 0) \rightarrow X(X(X(X(X\ (dout0 = din0\_delay))))));$

The experiments were carried out on a Sun Ultra-2 workstation with 296MHZ CPU and 768MB of memory. The experimental results of the verification of these properties with the MDG EL/EL2 algorithms are summarized in Table 6.1, including CPU time, memory usage and number of MDG nodes generated. To compare our approach with the ROBDD based language emptiness checking methods, we also conducted experiments on the same ATM switch fabric using the ltl_model_check option of the VIS tool [7]. Since VIS requires a Boolean representation of the circuit, we modeled the data input and output as Boolean vectors of 4-bit (which stores the minimum header information), 8-bit, and 16-bit.

Experimental results (see Tables 6.2 and 6.3) show that the verification of P4 (8-bit) as well as the properties P3 - 5 (16-bit) did not terminate (indicated by a "*"), while our MDG EL/EL2 algorithms were able to verify both in a few minutes for n-bit (abstract) data. Thus, MDG EL/EL2 algorithms enlarge the systems can be verified.

To compare the performance with the ROBDD based EL/EL2 algorithms, we

Table 6.2: Experimental results with VIS using EL algorithm for ATM

| | 4-bit | | | 8-bit | | | 16-bit | | |
|---|---|---|---|---|---|---|---|---|---|
| | Time (sec) | Mem (MB) | # BDD Nodes | Time (sec) | Mem (MB) | # BDD Nodes | Time (sec) | Mem (MB) | # BDD Nodes |
| P1 | 27.2 | 52 | 3064K | 31.2 | 52 | 3081K | 36.8 | 52 | 3096K |
| P2 | 11.7 | 45 | 1660K | 14.0 | 45 | 1670K | 21.8 | 46 | 1700K |
| P3 | 12.4 | 40 | 1357K | 899.8 | 629 | 98707K | * | * | * |
| P4 | 533 | 167 | 32771K | * | * | * | * | * | * |
| P5 | 14.7 | 44 | 1597K | 321.8 | 137 | 35106K | * | * | * |

Table 6.3: Experimental results with VIS using EL2 algorithm for ATM

| | 4-bit | | | 8-bit | | | 16-bit | | |
|---|---|---|---|---|---|---|---|---|---|
| | Time (sec) | Mem (MB) | # BDD Nodes | Time (sec) | Mem (MB) | # BDD Nodes | Time (sec) | Mem (MB) | # BDD Nodes |
| P1 | 27.1 | 52 | 3064K | 29.5 | 52 | 3081K | 36.8 | 52 | 3096K |
| P2 | 11.5 | 45 | 1660K | 14.1 | 45 | 1670K | 21.7 | 46 | 1700K |
| P3 | 12.5 | 40 | 1357K | 899 | 628 | 98707K | * | * | * |
| P4 | 533 | 166 | 32771K | * | * | * | * | * | * |
| P5 | 13.8 | 44 | 1597K | 317.6 | 137 | 35199K | * | * | * |

also conducted the experiments using MDG EL/EL2 algorithms on a concrete ATM model of 4-bit, where the data signals are modeled as variables of concrete sort *word4*. However, the verification did not terminate since the size of the model grew beyond the available memory and no automated techniques, such as dynamic term reorder or reduction algorithms, do exist in the current MDG package. On the other hand, VIS uses very powerful cone-of-influence [57] model reduction algorithms and automatic term reordering techniques. When we turned off this model reduction, VIS also failed to verify any of the properties on the 4-bit, 8-bit and 16-bit models.

### 6.1.4 Discussion

The statistics shown in Table 6.1 demonstrate that MDG based model checking of $\omega$-automata can verify properties on a parameterized implementation independent of the datapath width. In our MDG based method, one ASM model and one set of

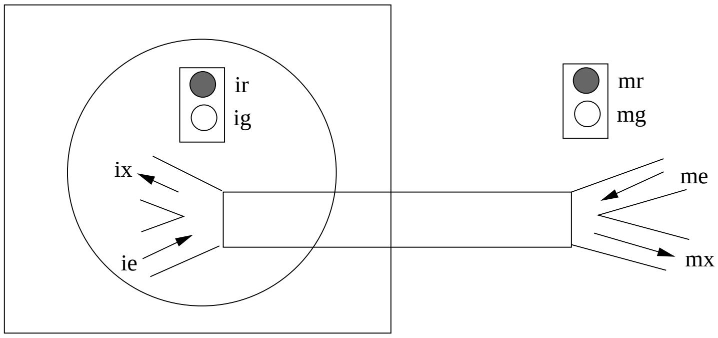



Figure 6.6: The Island Tunnel Controller

properties cover all possibile data widths. However, using the ROBDD based model checker, we have to build separate models and develop separate sets of properties for instances of different widths. From Tables 6.2 and 6.3, we can see with the width of the *data* signal increasing, the model to be verify becomes large, as shown in columns *# BDD Nodes* and *Memory*. The execution time hence increases rapidly, leading to some properties verification not terminating in a reasonable time.

## 6.2 Island Tunnel Controller

In this section we present the experiments on Island Tunnel Controller (ITC) example, we describe the behavioral description of Island Tunnel Controller and explain the verification of the ITC model.

### 6.2.1 System Description

The ITC controls the traffic lights at both ends of a tunnel based on the information collected by the sensors installed at both ends of the tunnel, a single lane tunnel connecting the mainland to the island, as shown in Figure 6.6. At each end of the tunnel, there is a traffic light. There are four sensors for detecting the presence of vehicles: one at the tunnel entrance ($ie$), one at the tunnel exit on the island side ($ix$), one at the tunnel entrance ($me$) and one at the tunnel exit on the mainland side ($mx$).

In [36], the following constraint is imposed: "at most sixteen cars may be on the island at any time". The number "sixteen" can be taken as a parameter and it can be any natural number. The constraint can thus be read as follows: "at most $n (n \geq 0)$ cars may be on the island at any time".

The specification of ITC using three communicating controllers and two counters proposed by [36] is shown in Figure 6.7. The state transition diagrams are shown in Figure 6.7. The island light controller (ILC) (Figure 6.8) has four states: *green, entering, red and exiting.* The outputs $igl$ and $irl$ control the green and red lights on the island side, respectively; $iu$ indicates that the cars from the island side are currently using the tunnel, and $ir$ indicates that the island is requesting the tunnel. The input $iy$ requests the island to yield control of the tunnel, and $ig$ grants control of the tunnel. A similar set of signals is defined for the mainland light controller (MLC). The tunnel controller (TC) processes the requests for access issued by the ILC and MLC. The island counter and the tunnel counter keep track of the numbers of cars currently on the island and in the tunnel, respectively. At each clock cycle, the count $tc$ of the tunnel counter is increased by 1 depending on signals $itc+$ and $mtc+$, or decremented by 1 depending on $itc-$ and $mtc-$, unless it is already 0. The island counter operates in a similar way, except that the increment and decrement signals are $ic+$ and $ic-$, respectively.

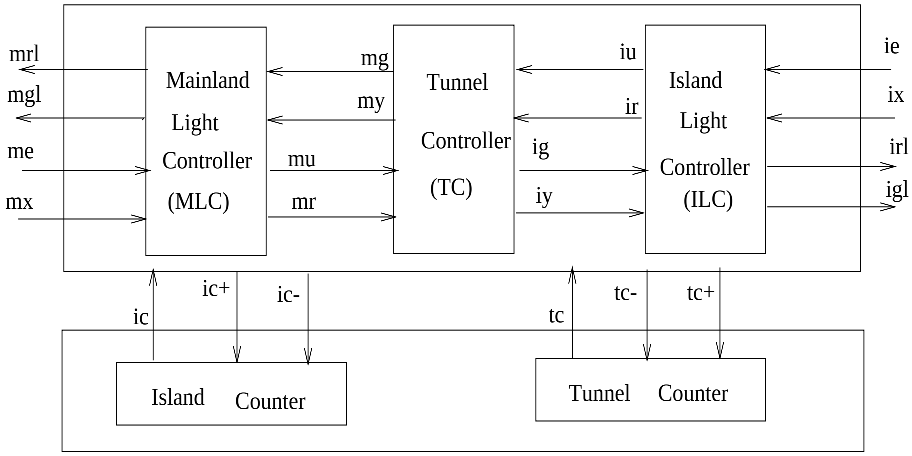



Figure 6.7: The specification of the Island Tunnel Controller

### 6.2.2 System Model

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

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

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

### 6.2.3 Verification

In the following we discuss 4 sample properties that we verified on the RTL implementation.

Table 6.4: Experimental results with MDG FCD algorithm and MDG MC algorithms for the ITC

| | MDG FCD | | | MDG MC Algorithms | | |
|---|---|---|---|---|---|---|
| | time (sec) | Memory (MB) | # of MDG nodes | time (sec) | Memory (MB) | # of MDG nodes |
| P1 | 5.51 | 5.45 | 7651 | 4.74 | 3.50 | 6613 |
| P2 | 19.21 | 21.45 | 23286 | 7.70 | 5.93 | 10152 |
| P3 | 6.03 | 5.83 | 7979 | 14.72 | 11.04 | 17909 |
| P4 | 16.90 | 13.98 | 19696 | 139.4 | 66.28 | 97159 |

**P1:** Cars never travel in both directions in the tunnel at the same time

$$G(\neg(igl = 1 \ \wedge \ mgl = 1)).$$

**P2:** The tunnel counter keeps the old value if ordered to increment and decrement simultaneously

$$G((tc- = 1 \ \wedge \ tc+ = 1) \rightarrow \mathsf{LET}\,(v = tc) \ \mathsf{IN} \ \ X(tc = v)).$$

**P3:** The island will eventually release the control right of tunnel if the tunnel controller requests

$$G((is = green) \rightarrow (F(is = red)))$$

under the fairness constraint

$$\neg((is = green) \ \rightarrow \ (iy = 0)).$$

**P4:** The car at the island entrance will eventually pass the tunnel

$$G((ie = 1) \rightarrow (F(igl = 1))).$$

All these properties were verified by the MDG LEC FCD algorithm. The experiments were carried out on a Sun Ultra-2 workstation with 296MHZ CPU and 768MB of memory. The experimental results are summarized in Table 6.4, including the CPU time, memory usage and number of MDG nodes generated. For the purpose of comparison, we conducted the same experiments on the existing MDG regular model checking (MC) tool, whose experimental results are also shown in Table 6.4.

From the statistics in Table 6.4, we can see that the performance of MDG MC is better than MDG LEC in verification of properties $P1$ and $P2$. MDG LEC is better in $P3$ and $P4$. This is because the performance of $\omega$-automata based model checking mainly dependent on the types of the generated automata [7].

### 6.2.4 Discussion

Usually, the performance of $\omega$-automata based model checking is worse than that of regular model checking method [18]. However, depending on the types of $\omega$-automata generated from the property, sometimes, the former performance outperforms the latter, especially, when the generated automata are terminal or weak automata [7]. This confirms our experimental results shown in Table 6.4.

## 6.3 Conclusion

In this chapter, we applied our MDG based language emptiness checking method on two case studies: the ATM switch fabric, and the Island Tunnel Controller. We compared MDG EL/EL2 algorithms with ROBDD based EL/EL2 algorithms, and compared FCD algorithm with MDG MC algorithms. Using MDG based method, we were able to use abstract variables to describe data signals and uninterpreted function symbols to represent the data operations. Therefore, MDG based method proceed the verification independent of the datapath widths. While, ROBDD based method cannot directly used in such circuit since the state space explosion problem. Hence, MDG based model checker enlarges the circuit can be verified by the existing ROBDD-based method. Compared to the regular MDG model checking method, the performance of MDG LEC is not always better than MDG MC. However, MDG LEC can accept a broader property templates than MDG MC. MDG LEC also uses a signal language checking algorithm for all property templates, while MDG MC builds an algorithm for every template.

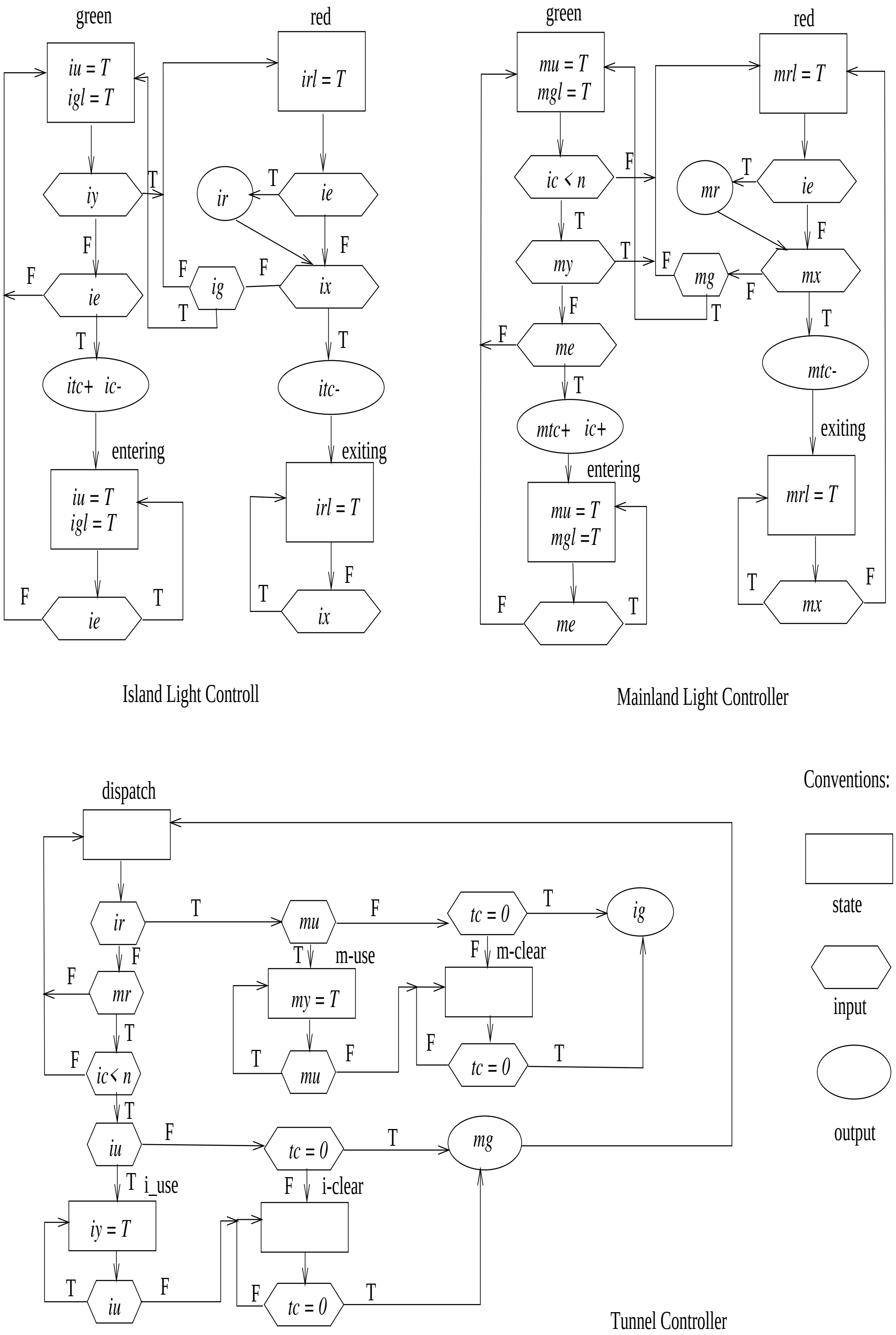



Figure 6.8: State transition graphs of the Island Tunnel Controller

# Chapter 7

# Conclusions

## 7.1 Summary of the Thesis

$\omega$-automata based model checking is a very successful verification technique. However, existing ROBDD based implementations are not adequate for verifying circuits with large and complex datapath because of the Boolean representation of circuits. As a new class of decision graphs, Multiway Decision Graphs (MDGs) have been proposed to use abstract variables to denote data values and uninterpreted function symbols to denote data operations. Furthermore, MDGs provide an Abstract State Machine (ASM) to model the system design. MDG based verification applications can hence handle larger hardware designs.

In this thesis, we explored $\omega$-automata based model checking using MDGs and developed a new application to the MDG tool set. To this end, we defined a first-order temporal logic ($\mathcal{L}_{MDG}{}^*$) to describe the property. We further developed an algorithm to transform the $\mathcal{L}_{MDG}{}^*$ formula into a Propositional Linear time Temporal Logic (PLTL) formula by constructing the ASMs for the property formula. We finally developed three algorithms to check the language of a *generalized Büchi automaton*, which can accept abstract variables to describe the transition system structure. The procedure of our MDG based language emptiness checking is summarized as follows:

First, we presented $\mathcal{L}_{MDG}{}^*$ as a property language based on a subset of a first-order linear time temporal logic, which is obtained by adding the many-sorted (with distinction of abstract sort and concrete sort ) first-order logic on a linear time structure. Compared to a previously defined specification language $\mathcal{L}_{MDG}$, $\mathcal{L}_{MDG}{}^*$ defines a broader language by allowing arbitrary temporal nesting forms, which breaks the template limitation of $\mathcal{L}_{MDG}$.

Second, we presented an algorithm to transform the satisfaction of a $\mathcal{L}_{MDG}{}^*$ formula on all computations of an ASM into the satisfaction of a PLTL formula on all computations of a composed state machine. The algorithm first identified the atomic formulas $P$ in the $\mathcal{L}_{MDG}{}^*$ formula. It then constructed an ASM for each atomic formula. After that, it composed the constructed ASMs with the original ASM, and the generated ASM is bisimular to the original ASM with respect to $P$. It finally replaced atomic formulas $P$ with propositions in both the property formula and composed state machine. The constructed ASMs were described by the circuits with MDG components in MDG-HDL according to a set of proposed rules. We furthermore proved the correctness of transformation.

Third, we presented three algorithms to check emptiness of the language of a *generalized Büchi automaton*, which accepts abstract variables and uninterpreted function symbols to describe the transition system structure. The first two algorithms were adapted from the existing EL/EL2 algorithms to the MDG based approaches (MDG EL/EL2). We proved their correctness through proving the correctness of the adaptation. We further tested these algorithms using a case study on an ATM (Asynchronous Transfer Mode) switch fabric. To compare with existing EL/EL2 algorithms, we also conducted the case study with the EL/EL2 algorithms in VIS. The results showed that the MDG EL/EL2 algorithms could handle larger system designs than the exsting EL/EL2 algorithms. Furthermore, we developed a Fair Cycle Detection (FCD) algorithm, which worked as follows: starting from the initial states $Z^i$, first for each fairness condition, computed a state satisfying the condition on each path staring from $Z^i$; then computed the next states as the new reachable states $Z^{i+1}$; finally checked if $Z^{i+1}$ was empty or there existed a cycle by testing the inclusion $Z^i \subseteq Z^{i+1}$. We also applied the FCD algorithm to the Island Tunnel Controller (ITC) benchmark. The results showed that our FCD algorithm worked very efficiently.

Finally, we implemented our method in the MDG tool set and developed a new application: MDG LEC. Experimental results proved that the new application is superior to the ROBDD based by increasing the range of circuits that can be verified. MDG LEC also provides more property templates than the existing MDG regular model checking application.

## 7.2 Future Research Directions

Although a lot of work has been accomplished on the MDG tool set, there are still several areas in which the system could be extended.

- Solving the non-termination problem

  It is well known that in some situations MDG based algorithms may suffer from the non-termination problem, that is, the state enumeration procedure may not be terminated. In the literature, there exist two approaches proposed to tackle this problem. The first one uses *$\rho$-terms* that can finitely represent infinite sets of states [64]. An extension of the syntax of MDG and MDG-based algorithms could incorporate $\rho$-terms to solve the non-termination problem when the generated set of states exhibit certain repetitive patterns. The second one modifies the original ASM structural description according to certain rules to avoid the non-termination problem [64]. It would be valuable to explore a more general method that could automatically analyze the ASM description, modify the design description and infer $\rho$-terms. Implementing these ideas in the MDG package could extend the applicability of the MDG verification tools.

- Develop a counterexample facility in the MDG package

  If a property fails on the system design model, the model checker should produce a counterexample to give hints to the designers in order to catch the flaws in their designs. This facility is only available in the invariant checking application in the MDG tool set at present. Since the counterexample is critical to the designer, development of this function is imperative. The counterexample application of

MDG invariant checking generates a counterexample by remembering all visited states at each step from the initial states to states violating the condition (bad states). Then back-trace a path that can reach a "bad" state from an initial state. Although this method is feasible in the invariant checking method, it cannot be used when a cycle needs to be detected, which is an necessary step in finding the counterexample in MDG LEC. Moreover, our MDG algorithms may produce false negatives. More specifically, when the algorithm reports failure, which means the property is not true for all the interpretation, it is still possible that the property holds for some specific interpretation. If it is the case, the counterexample is spurious.

- Develop the MDG package with C++

  The MDG package is a prototype implemented in Prolog, which is a functional language widely used in logic programming. The execution of the MDG tools wastes a lot of time in the parsing of complex logic predicates. Thus, the MDG tools cannot be superior to other tools implemented for example in C. The MDG development group is aware of this problem and is currently rewriting the MDG package in C++ to improve the MDG tools' performance.

- Experimental verification of the proposed method and tool in this thesis using more elaborate industrial and academic benchmarks.

# Bibliography

[1] K. D. Anon, N. Boulerice, E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, Y. Xu and Z. Zhou. MDG Tools for the Verification of RTL Designs. In *Computer-Aided Verification*, LNCS 1102: 433-436, Springer Verlag, 1996.

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

[3] M. C. Browne, E. M. Clarke, D.L. Dill and B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transactions on Computers*, 35(12): 509-516, December 1986.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan and D. L. Dill. Sequential Circuit Verification using Symbolic Model Checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference,* pages 46-51, Orlando, USA, June 1990.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. Symbolic Model Checking: $10^{20}$ States and Beyond. *Information and Computation*, 98(2): 142-170, June 1992.

[6] J. Bohn, W. Damm, O. Grumberg, H. Hungar, and K. Laster. First-Order-CTL Model Checking. In *Proc. of 18th International Conference Foundations of Software Technology and Theoretical Computer Science*, pages 283-294, Chennai, December 1998.

[7] R. Bloem, K. Ravi, and F.Somenzi. Efficient Decision Procedures for Model

Checking of Linear Time Logic Properties. In *Computer Aided Verification*, *LNCS* 1633: 222-235, Springer Verlag, 1999.

[8] R. E. Bryant. Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification, In *Proceedings of the International Conference on Computer-Aided Design,* pages 236-243, San Jose, USA, November 1995.

[9] R. K. Brayton *et. al.* VIS: A System for Verification and Synthesis. In *Computer Aided Verification*, LNCS 1102: 428-432, Springer Verlag, July 1996.

[10] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8): 677-691, August 1986.

[11] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *Computer Aided Verification*, LNCS 808: 68-80, Springer Verlag, 1994.

[12] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar and Z. Zhou. Automated Verification with Abstract State Machines Using Multiway Decision Graphs. *Formal Hardware Verification: Methods and Systems in Comparison*, LNCS 1287: 79-113, Springer Verlag, 1997.

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

[14] E. M. Clarke, M. C.Browne and O. Grumberg. Characterizing Kripke Structures in Temporal Logic. *Theoretical Computer Science,* 59(1): 115-131, 1988.

[15] E. M. Clarke, and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Logics of Programs*, LNCS 131: 52-71, Springer Verlag, 1981.

[16] 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): 244-263, April 1986.

[17] E. Clarke, O. Grumberg and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems,* 16(16): 1512-1542, 1994.

[18] E. Clarke, O. Grumberg and K. Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design,* pages 47-71, February 1997.

[19] E. M. Clarke, O. Grumberg, and D. A. Peled. Model Checking. *The MIT Press*, Cambridge, Massachusetts, 1999.

[20] F. Corella, M. Langevin, E. Cerny, Z. Zhou and X. Song. State Enumeration with Abstract Descriptions of State Machines. In *Proceedings Conference on Correct Hardware Design and Verification Methods*, pages 146-160, Frankfurt, Germany, October 1995.

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

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

[23] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2): 275-288, 1992.

[24] D. Cyrluk and P. Narendran. Ground Temporal Logic: A Logic for Hardware Verification. In *Compute Aided Verification*, LNCS 818: 247-259, Springer Verlag, 1994.

[25] D. Cyrluk and M. K. Srivas. Theorem Proving: Not an Esoteric Division, but the Unifying Framework for Industrial Verification. In *Proceeding IEEE International Conference on Computer Designs* Autsin, USA, October 1995.

[26] M. Daniele, F. Giunchiglia, and M.Y. Vardi. Improved Automata Generation for Linear Temporal Logic. In *Computer Aided Verification*, LNCS 1633: 249-260, Springer Verlag, 1999.

[27] D. R. Dams, R. Gerth, and O. Grumberg. Abstract Interpretation of Reactive System *ACM Transaction on Programming Language and Systems,* 19(2): 253-291, 1997.

[28] D. R. Dams, O. Grunberg and R. Gerth. Generation of Reduced Models for Checking Fragement of CTL. In *Computer Aided Verification,* LNCS 697: 479-490, 1993.

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

[30] E. A. Emerson. Temporal and Modal logic. In Handbook of Theoretical Computer Science, Volume B: 995–1072, Elsevier Science Publishers, 1990.

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

[32] E.A. Emerson and C.-L. Lei. Efficient Model Checking in Fragments of the Propositional $\mu$-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 267-278. IEEE Computer Society Press, 1986.

[33] E. A. Emerson, A. P. Sistla. Symmetry and Model Checking. In *Computer Aided Verification,* LNCS 697: 463-478, Springer Verlag, 1993.

[34] K. Etessami, G. Holzmann. Optimizing Büchi Automata, In *Concurrency Theory*, LNCS 1877: 153-167, Springer Verlag, 2000.

[35] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is There a Best Symbolic Cycle-Detection Algorithm? In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2031: 420-434, Springer Verlag, 2001.

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

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

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

[39] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Protocol Specification, Testing, and Verification*, pages 3–18, North-Holland, 1995.

[40] O. Grumberg, D. Long. Model Checking and Modular Verification. *ACM Transaction on Programming Language and Systems,* 16(3): 843-871, 1994.

[41] O. Grumberg. Abstractions and Reductions in Model Checking. In *NATO Science Series*, Vol. 62, 2001.

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

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

[44] G.J. Holzmann. The Model Checker Spin. *IEEE Transaction on software Engineering,* 23(5): 279-295, May 1997.

[45] R. H. Hardin, Robert P. Kurshan, Sandeep K. Shukla, and Moshe Y. Vardi. A New Heuristic for Bad Cycle Detection using BDDs. *Formal Methods in System Design*, 18(2): 131–140, 2001.

[46] R. Hojati and R. K. Brayton. Automatic Datapath Abstraction in Hardware Systems. In *Computer Aided Verification*, LNCS 939: 98-113, Springer Verlag, 1995.

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

[48] R. Hojati, H. Touati, R. P. Kurshan, and R. K Brayton. Efficient $\omega$-regular Language Containment. In *Computer Aided Verification*, LNCS 663: 371-382, Springer Verlag, 1992.

[49] A. J. Hu. Formal Hardware Verification with BDDs: An Introduction. In *Proceedings of IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing,* pages 677-682, Victoria, Canada, August 1997.

[50] H. Hungar,O. Grumberg, and W. Damm. What if Model Checking Must be Truely Symbolic. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1019: 1-20, 1995.

[51] N. Ishiura, H. Sawada and S. Yajima. Minimization of Binary Decision Diagrams based on Exchanges of Variables. In *Proceedings of the International Conference on Computer Aided Design,* pages 473-475, Santa Clara, USA, November 1991.

[52] C. Kern and M. R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4: 123-193, 1999.

[53] Y. Kesten, A. Pnueli and L-o. Raviv. Algorithmic Verification of Linear Tmeporal logic specification. In *Automata, Languages, and programming*, LNCS 1443: 1-16, Springer Verlag, 1998.

[54] S. A. Kripke. Semantical Considerations on Modal Logic. In *Proceedings of Colloquium on Modal and Many-Valued Logics,* Helsinki, 1962.

[55] T. Kropf. Benchmark-Circuits for Hardware Verification. In *Theorem Provers in Circuit Design*, LNCS 901: 1-12, Springer Verlag, 1994.

[56] R. P. Kurshan. Analysis of Discrete Event Coordination. In *Proceedings of Research and Education in Concurrent Systems Workshop,* page 414-453, Mook, The Netherlands, 1989.

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

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

[59] D. Kelley. Automata and Formal Language. *Princeton University Press*, 1995.

[60] I. Leslie and D. McAuley. Fairisle: an ATM Network for the Local Area. *ACM Communication Review*, page 327-336, 1991.

[61] M. C. McFarland. Formal Verification of Sequential Hardware: A Tutorial. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems,* 12(5): 633-654, May 1993.

[62] K. L. McMillan. *Symbolic Model Checking.* Kluwer Acardmeic Publishers, 1998.

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

[64] O. Ait Mohamed, X. Song, and E. Cerny. On the Non-termination of MDG-based Abstract State Enumeration. *Theoretical Computer Science* 300: 161–179, 2003.

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

[66] K. S. Namjoshi and R. P. Kurshan. Syntactic Program Transformations for Automatic Abstraction. In *Computer Aided Verification*, LNCS 1855: 435-449, Springer Verlag, 2000.

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

[68] D. Park. Concurrency and Automata on Infinite Sequences. *Theoretical Computer Science*, pages 167-183, 1981.

[69] Quintus Prolog Manual. *Quintus Corporation,* v. 3.1, 1991.

[70] K. Ravi, R. Bloem, and F. Somenzi. A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles. In *Formal Methods in Computer-Aided Design*, LNCS 1954: 143-160, Springer Verlag, 2000.

[71] C. Seger. An Introduction to Formal Hardware Verification. University of British Columbia Technical Report 92-13, 1992.

[72] F.Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In *Computer Aided Verification*, LNCS 1855: 247-263, Springer Verlag, 2000.

[73] F. Somenzi, K. Ravi, and R. Bloem. Analysis of Symbolic SCC Hull Algorithms. In *Formal Methods in Computer Aided Design*, LNCS 2517: 88-105, Springer Verlag, 2002.

[74] H. J. Touati, R. K. Brayton and R. P. Kurshan. Testing Language Containment for $\omega$-automata Using BDDs. *Information and Computation,* 118(1): 101-109, 1995.

[75] H. J. Touati, R. K. Brayton and R. Kurshan. Testing Language Containment for $\omega-$automata Using BDD. *Information and Computation*, 118(1): 101-109, 1995.

[76] W. Thomas. Automata on Infinite Objects. In Handbook of Theoretical Computer Science, 2: 165-191 Elsevier Science Publishers, 1994.

[77] S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin, and O. Ait-Mohamed. Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 18(7): 956-972, 1999.

[78] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDDs. In *Computer-Aided Design,* LNCS 430: 130-133, Springer Verlag, 1990.

[79] M. Y. Vardi. An Automata-Theoretic Approach to Linear Temporal Logic. In *Logics for Concurrency: Structure Versus Automata*, LNCS 1043: 238-266, Springer Verlag, 1996.

[80] A. Xie, P. A. Beerel. Implicit Enumeration of Strongly Connected Components. In *Proceedings of the International Conference on Computer-Aided Design,* pages 37-40, San Jose, USA, November 1999.

[81] Y. Xu, E. Cerny, X. Song, F. Corella, and O. Ait Mohamed. Model Checking for a First-order Logic using Multiway Decision Graphs. In *Computer Aided Verification*, LNCS 1427: 219 - 231, Springer Verlag, 1998.

[82] Y. Xu. Model Checking for a First-order Temporal Logic using Multiway Decision Graphs. *PhD Thesis*, University of Montreal, 1999.

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

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

[85] 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*, pages 575-580, Chiba, Japan. August, 1995.

[86] 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 *Formal Methods in Computer Aided Design*, LNCS 1166: 233-247, Spring Verleg, 1996.