First-Order LTL Model Checking Using MDGs

Fang Wang, Sofiène Tahar, and Otmane Ait Mohamed

Department of Electrical and Computer Eng
Concordia University, Montreal, Quebec, Canada
{f_wang, tahar, ait}@ece.concordia.ca

Abstract. In this paper, we describe a first-order linear time temporal logic (LTL) model checker based on multiway decision graphs (MDG). We developed a first-order temporal language, $\mathcal{L}_{\mathcal{MDG}}{}^*$, which expresses a subset of many-sorted first-order LTL and extends an earlier language, $\mathcal{L}_{\mathcal{MDG}}$, defined for an MDG based abstract CTL model checking. We derived a set of rules, enabling the transformation of $\mathcal{L}_{\mathcal{MDG}}{}^*$ formulas into generalized Büchi automata (GBA). The product of this GBA and the abstract state machine (ASM) model is checked for language emptiness. We have lifted two instances of the generalized Strongly Connected Component(SCC)-hull (GSH) checking algorithm [17] to support abstract data and uninterpreted functions based on operators available in the MDG package. Experimental results have shown the superiority of our tool compared to the same instances of GSH implemented with BDDs in VIS.

1 Introduction

Formal verification has received considerable attention from the electrical engineering, computer science and the industry communities, where many BDD based formal verification tools being developed over the years. These, however, suffer from the well-known state space explosion problem. Multiway Decision Graphs (MDGs) [5] have been introduced as one way to reduce this problem. MDGs are based on a many-sorted first-order logic with a distinction between concrete and abstract sorts. Abstract variables are used to represent data signals, while uninterpreted function symbols are used to represent data operations, providing a more compact description of circuits with complex data path. Many MDG based verification applications have been developed during the last decade, including invariant checking, sequential equivalence checking, and abstract CTL model checking [21] of abstract state machines (ASM) [5]. The MDG tools are available at [22].

In this paper we introduce a new MDG verification application by implementing automata based model checking of a subset of first-order linear time temporal logic (LTL). Generally, LTL model checking verifies a Kripke structure with respect to a propositional linear time temporal logic (PLTL) formula. A PLTL formula ϕ is valid if it is satisfied by all paths of the Kripke structure M. The validation of ϕ can be done by converting its negation into a Generalized Büchi Automaton (GBA) [19] $B_{\neg\phi}$, composing the automaton with the

F. Wang (Ed.): ATVA 2004, LNCS 3299, pp. 441–455, 2004.

model M, and checking its language emptiness [19]. The main idea of the work we describe in this paper is to lift classical LTL model checking procedures to the language emptiness checking (LEC) of a GBA encoded with MDGs. To this end, we define an extended temporal logic, called $\mathcal{L}_{\mathcal{MDG}}{}^*$, for which we have developed a set of derivation rules that transform $\mathcal{L}_{\mathcal{MDG}}{}^*$ properties into PLTL formulas augmented with a transformation circuit, which will be composed with the system model (ASM) under verification. We use an automata generator to get a GBA for the negation of this PLTL formula. Language emptiness checking based on two instances of the GSH algorithm [17] is finally performed on the product of this latter and the composed ASM described earlier. We call this new MDG verification application MDG LEC.

The rest of the paper is organized as follows: Section 2 describes related work. Section 3 overviews the notion of multiway decision graphs. Section 4 defines the first order linear temporal logic $\mathcal{L}_{\mathcal{MDG}}{}^*$ and related transformation rules. Section 5 describes the language emptiness checking algorithms. Section 6 provides a case study applying the developed MDG LEC tool on an ATM (Asynchronous Transfer Mode) switch fabric. Finally, Section 7 concludes the paper.

2 Related Work

The idea of first-order temporal logic model checking is not new. For instance, Bohn *et. al* [3] presented an algorithm for checking a first-order CTL specification on first-order Kripke structure, an extension of "ordinary" Kripke structures by transitions with conditional assignments. The algorithm separates the control and data parts of the design and generates the first-order verification conditions on data. The control part can be verified with Boolean model checking, while the data part of the design has to be verified using interactive theorem proving. Compared to this work, our logic is less expressive since $\mathcal{L}_{\mathcal{MDG}}{}^*$ cannot accept existential quantification. However, in our approach the property is checked on the whole model automatically, while in [3] a theorem prover is needed to validate the first-order verification conditions. Besides, our method can be applied on any finite state models, while their application is limited to designs that can be separated into data and control parts.

Hojati *et.al* [13] proposed an integer combinational/sequential (ICS) concurrency model to describe hardware systems with datapath abstraction. They used symbols such as finite relations, interpreted and uninterpreted integer functions and predicates, and proceeded the verification of ICS models using language containment. For a subclass of "control-intensive" ICS models, integer variables in the model can be replaced by enumerated variables, hence enabling a verification at the Boolean level without sacrificing accuracy. Compared to ICS, our ASM models are more general in the sense that the abstract sort variables in our system can be assigned any value in their domain, instead of a particular constant or function of constants as in ICS models. For the class of ICS models where finite instantiations cannot be used, our verification system can still compute all the reachable states and check properties.

Cyrluk and Narendran [6] defined a first-order temporal logic—ground temporal logic (GTL), which for universally quantified computation paths, falls in between first-order and propositional temporal logics. GTL models consist of first-order language interpretation models and infinite sequences of states. The validity problem is the same as checking of an LTL formula. The authors further identified a decidable fragment of GTL, which consists of $\Box P$(always P) formulas, where P is a GTL formula only containing an arbitrary number of "Next" operators. For this decidable fragment, they did not show how to build the decision procedure, though. Compared to [6], our $\mathcal{L_{MDG}}^*$ is more expressive (cf. Section 4).

In [4], Burch and Dill also presented a subset of first-order logic, specifically, quantifier-free logic of equality with uninterpreted functions, to specify properties for verifying microprocessor control circuitry . Their method is appropriate for verification of microprocessor control because it allows abstraction of datapath values and operations. However, their approach cannot verify liveness properties.

Based on MDGs, Xu *et. al* [21] developed an abstract CTL model checking tool, which verifies an ASM with respect to a first-order temporal logic ($\mathcal{L_{MDG}}$). $\mathcal{L_{MDG}}$ consists of limited set of templates including: $\mathsf{A}(P)$, $\mathsf{AG}(P)$, $\mathsf{AF}(P)$, $\mathsf{A}(P)\mathsf{U}(Q)$, $\mathsf{AG}(P \rightarrow (\mathsf{F}(Q))$, and $\mathsf{AG}((P) \rightarrow ((Q)\mathsf{U}(R))$, where P, Q, and R are Next_let_formulas.[1] This MDG tool does not allow temporal operator nesting and cannot deal with properties beyond its templates. For example, a property like $\mathsf{G}(a = 1 \rightarrow \mathsf{F}(b = 1) \wedge \mathsf{F}(c = 1))$ cannot be expressed in $\mathcal{L_{MDG}}$.

3 Multiway Decision Graphs

The underlying logic of MDGs is a many-sorted first-order logic with a distinction between concrete and abstract sorts. Concrete sorts have an enumeration, while abstract sorts do not. This distinction leads to the definition of concrete variables, abstract variables, individual constants appearing in the enumeration, generic constants of abstract sorts, abstract function symbols and cross-operators [5]. Let f denote a function symbol of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$. Then, if α_{n+1} is an abstract sort, then f is an abstract function symbol; if α_{n+1} is a concrete sort, and at least one of the sorts $\alpha_1, \ldots, \alpha_n$ is abstract, f is a cross-operator.

An interpretation is a mapping ψ that assigns a denotation to each sort, constant and function symbol and satisfies the following conditions: 1) The denotation $\psi(\alpha)$ of an abstract sort α is a non-empty set; 2) If α is a concrete sort with enumeration $\{a_1, \cdots, a_n\}$ 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 α, 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) A variable assignment

[1] Since our $\mathcal{L_{MDG}}^*$ is an extension of $\mathcal{L_{MDG}}$, Next_let_formula will be explained in Section 4.

with the domain X compatible with an interpretation ψ is a function ϕ that maps every variable $x \in X$ of sort α to an element $\phi(x)$ of $\psi(\alpha)$, Φ_X^{ψ} is used to represent the set of ψ-compatible assignments of the variables in X; 6) $\psi, \phi \models P$ means that the formula P is true under an interpretation ψ and $\psi-$compatible variable assignment ϕ, $\psi \models P$ represents $\psi, \phi \models P$ for every $\psi-$compatible variable assignment ϕ, and $\models P$ means $\psi \models P$ for all ψ; Two formulas P and Q are logically equivalent iff $\models P \Leftrightarrow Q$.

An MDG is a finite directed acyclic graph. An internal node of an MDG can be labeled with a concrete variable with its edge labels being the individual constants in the enumeration of the sort; Or it can be an abstract variable and its edges are labeled by abstract terms of the same sort; Or it can be a cross-operator with its edges labels being the individual constants. An MDG may have only one leaf node denoted as **T**, which means all paths in the MDG correspond to true formula. MDGs are used to represent relations as well as sets in abstract state machines (ASM). An ASM is defined as a tuple $D = (X, Y, W, F_I, F_T, F_O)$, where X, Y and W are disjoint finite sets of input, state, and output symbols, respectively. F_I is an MDG representing the initial states, F_T is an MDG for the state transition relation, and F_O is an MDG for the output relation.

The MDG package provides a set of basic operators, including conjunction (Conj) of a set of MDGs with different nodes of abstract variables; Disjunction (Disj) of a set of MDGs; Relational Product (RelP), which computes conjunction, existentially quantification, and renaming substitution in one pass; Pruning by Subsumption (PbyS) produces an MDG, representing the difference of two abstract sets, given say by MDGs P and Q, by pruning the edges from P contained by Q. Finally, a procedure ReAn(G, C) computes the set of 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, with any interpretation ψ, while the invariant condition C holds in all reachable states. In M, $S_I = Set^{\psi}(F_I) = \{\phi \in \Phi_Y^{\psi} \mid \psi, \phi \models (\exists U) F_I\}$, $R_O = Set^{\psi}(F_O) = \{(\phi, \phi', \phi'') \in \Phi_X^{\psi} \times \Phi_Y^{\psi} \times \Phi_Z^{\psi} \mid \psi, \phi \cup \phi' \cup \phi'' \models F_O\}$, and $R_T = Set^{\psi}(F_T) = \{(\phi, \phi', \phi'') \in \Phi_X^{\psi} \times \Phi_Y^{\psi} \times \Phi_Z^{\psi} \mid \psi, \phi \cup \phi' \cup \phi'' \models F_T\}$.

4 MDG Language Emptiness Checking Approach

4.1 MDG LEC Tool Structure

The structure of the MDG LEC is shown in Figure 1. It takes as inputs a property φ in a first-order temporal logic ${\mathcal{L}_{\mathcal{MDG}}}^*$ (to be defined later) and a system design M modeled as an ASM. The tool first transforms the ${\mathcal{L}_{\mathcal{MDG}}}^*$ formula φ into a set of atomic propositions (AP_{φ}) augmented by a circuit C_{φ}, which constructs all basic logic operators as well as equality conditions. The details on the construction will be given in Section 4.3. The tool then builds the equivalent PLTL formula ϕ. C_{φ} is further composed with M to produce a new ASM M' by the composer, which connects ASM_variables (input, state and output signals) in C_{φ} with the system model. Using AP_{φ}, we reconstruct a syntactically equivalent PLTL property formula ϕ, which we feed into an automata generator producing a GBA $B_{\neg\phi}$. The latter is then composed with M' to produce an ASM M'' and a set of fairness conditions $f_{B_{\neg\phi}}$. Finally, the tool checks if the language of the composed machine is empty using an adapted forward generalized Strongly Connected Component(SCC)-hull (GSH) algorithm [17]. In the following, we give a definition for $\mathcal{L_{MDG}}^*$ and detail the transformation procedure. The checking algorithm will be described in Section 4.

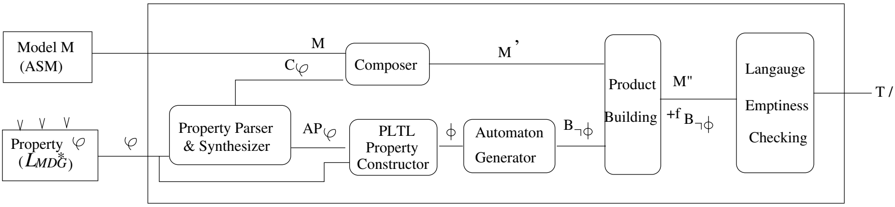

Fig. 1. Structure of MDG LEC

Given a PLTL formula ϕ, there exist many procedures to generate a GBA $B_{\neg\phi}$ for ϕ such that GPVW [11], GPVW+ [11], LTL2AUT [7], and Wring [10]. Although these procedures are based on the same tableau construction method, Wring improves on others by applying rewriting on the PLTL formula and simplifying the result GBA. We have chosen the Wring procedure for the automaton generation in MDG LEC. The constructed GBA is an ω-automaton with several sets of accepting states defined as the fairness condition. A run is accepted if it contains at least one state in every accepting set infinitely often. As a result, the language of the automaton is nonempty iff the automaton contains a fair cycle – a (reachable) cycle that contains at least one state from every accepting set, or equivalently, a (reachable) nontrivial SCC that intersects each accepting set [19].

4.2 $\mathcal{L_{MDG}}^*$: A First-Order Temporal Logic

Let $\mathcal{F}$ be a set of function symbols and $\mathcal{V}$ a set of variables. We denote the set of terms freely generated from $\mathcal{F}$ and $\mathcal{V}$ by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The syntax of an $\mathcal{L_{MDG}}^*$ formula is given by the following grammar:

$$\begin{array}{lll}
\textit{Sort} & \mathcal{S} & ::= S \mid \underline{S} \\
\textit{Abstract sort} & S & ::= \alpha \mid \beta \mid \gamma \mid \ldots \\
\textit{Concrete sort} & \underline{S} & ::= \underline{\alpha} \mid \underline{\beta} \mid \underline{\gamma} \mid \ldots \\
\textit{Generic constant} & C & ::= a \mid b \mid c \mid \ldots \\
\textit{Concrete constant} & \underline{C} & ::= \underline{a} \mid \underline{b} \mid \underline{c} \mid \ldots \mid \underline{0} \mid \underline{1} \mid \ldots \\
\textit{Variable} & \mathcal{V} & ::= V \mid \underline{V} \mid \underline{\underline{V}} \\
\textit{Abstract variable} & V & ::= x \mid y \mid z \mid \ldots \\
\textit{Concrete variable} & \underline{V} & ::= \underline{x} \mid \underline{y} \mid \underline{z} \mid \ldots
\end{array}$$

$$
\begin{array}{lll}
\textit{Ordinary variable} & \underline{\underline{V}} & ::= \underline{\underline{x}} \mid \underline{\underline{y}} \mid \underline{\underline{z}} \mid \ldots \\
\textit{Atomic formula} & A & ::= \textit{true} \mid \textit{false} \mid Eq \\
 & Eq & ::= V_1 = V_2 \mid V = C \mid \underline{V_1} = \underline{C} \mid \underline{C_1} = \underline{C_2} \mid V = \mathcal{T}(\mathcal{F}, \mathcal{V}) \\
\textit{Next\_let\_formula} & N & ::= A \mid !N \mid N \;\&\; N \mid N \parallel N \mid N \;\rightarrow\; N \mid \mathsf{X}\; N \\
 & & \quad \mid \mathsf{LET}\;\; (\underline{\underline{V}} = V)\; \mathsf{IN}\, N \\
\textit{Property} & P & ::= N \mid !\; P \mid P_1 \;\&\; P_2 \mid P_1 \parallel P_2 \mid P_1 \;\mathsf{U}\; P_2 \mid P_1 \;\mathsf{R}\; P_2 \mid \mathsf{G}\; P \\
 & & \quad \mid \mathsf{F}\; P
\end{array}
$$

Semantics. An infinite path π in an ASM M is an infinite sequence of states. We denote by π^i the suffix of π beginning with the state π_i, which is the i^{th} state in π. We use $Val_{\phi \cup \sigma}(t)$ to denote the value of term t under a ψ-compatible assignment ϕ (cf. Section 3) to state variables, input variables, and output variables and a ψ-compatible assignment σ to the ordinary variables v. The satisfaction of an $\mathcal{L_{MDG}}^*$ formula along a path π under the ψ-compatible assignment σ to the ordinary variable v is defined inductively as follows.

$\pi, \sigma \models t_1 = t_2$ iff $Val_{\pi_0 \cup \sigma}(t_1) = Val_{\pi_0 \cup \sigma}(t_2)$.

$\pi, \sigma \models \mathsf{LET}\ (v = t)\mathsf{IN}\ p$ iff $\pi, \sigma' \models p$
where $\sigma' = \sigma \setminus \{(v, \sigma(v))\} \cup \{(v, Val_{\pi_0 \cup \sigma}(t))\}$

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

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

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

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

$\pi, \sigma \models \mathsf{X}p$ iff $\pi_1, \sigma \models p$.

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

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

$\pi, \sigma \models p\mathsf{U}q$ iff for some $k \geq 0, \pi_k, \sigma \models q$, and $\pi_j, \sigma \models p$ for all $j(0 \leq j < k)$.

$\pi, \sigma \models p\mathsf{R}q$ iff for some $k \geq 0, \pi_k, \sigma \models q$, or there exists j,
$\pi_j, \sigma \models p$ for all $j(0 \leq j < k)$.

An $\mathcal{L_{MDG}}^*$ formula is said to be *satisfied* in the machine D if it is satisfied along a path of M; a formula is said to be *valid* in D if it is satisfied along all paths of M.

4.3 $\mathcal{L_{MDG}}^*$ Transformation

As shown in Figure 1, the first step of the MDG LEC is to transform the formula $\mathcal{L_{MDG}}^*$ into a PLTL formula. The transformation of formula in $\mathcal{L_{MDG}}^*$ to PLTL is obtained by generating a circuit description for each subformula (Next_let_formula) in the property. The generated circuit provides a single atomic proposition output, which will replace the entire Next_let_formula in the original property. Applying the same procedure to each subformula results in a simpler formula. The rules which govern this construction are given below:

- $< V_1 = V_2 >= absComp(V_1, V_2)$, where *absComp* is a cross-operator, which denotes the truth of $V_1 = V_2$ in the current state of the circuit. The cross-operator is partially interpreted by the rewriting rule: $absComp(X, X) = 1$, which can be interpreted as “the value of the two abstract terms are equal if the two abstract terms are syntactically the same”.

- $< V_1 = C >= absComp(V_1, C)$.
- $< \underline{V_1} = \underline{C} >= comp(\underline{V_1}, \underline{C})$, where *comp* is a concrete function.
- $< \overline{\mathsf{LET}}\ (\underline{\underline{V}} = V)\ \mathsf{IN}\, \overline{N} >$. Here, search the referred ASM variable V in N, count the nesting depth n of the X operator between the *Let* operator and the atomic formula, and add a sequence of n "registers". The input of the sequence is V, and its output is the ordinary variable $\underline{\underline{V}}$.
- $N_1 \rightarrow N_2$ is handled as an abbreviation for $\neg N_1 \parallel N_2$.
- &, $\|$, and ! are built using the logic gates "and", "or" and "not" gates, respectively.
- $< V = T >= buildterm(V, T)$, where *buildterm* is a function that implements a term $T \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. It builds the function symbols for each element in $\mathcal{F}$ and connects them together according to the appearance order in the term. The inputs are the variables of $\mathcal{V}$, while the output is the term T. The cross-operator *absComp* is used to denote the truth of $V = T$.

The function $< ., . >$ is generalized to a property definition as follows:

1. For the atomic formula *false* or *true*, we generate constant signal 0 or 1, respectively.
2. $< T_{op}\ N >$, where $T_{op} = \mathsf{U}, \mathsf{R}, \mathsf{G}, \mathsf{F}$. According to temporal operators before the Next_let_formula N, we rewrite the N to $(true\ \&\ N)$ or $(false \mid N)$. If the immediate operator is F, U, or R, we use $(true\ \&\ N)$, or $(false \mid N)$ otherwise. This is done to make sure that the property is checked after n cycles (using registers) from the initial states, where n is the maximum nesting depth of the X operators in the property.
3. $< P_1\ op\ P_2 >=< P_1 > OP < P_2 >$, where OP is an implementation of op.
4. $< T_{op}\ P >= T_{op} < P >$, where $T_{op} = \mathsf{U}, \mathsf{R}, \mathsf{G}, \mathsf{F}$.

4.4 Illustrative Example

To illustrate the $\mathcal{L}_{\mathcal{MDG}}{}^{*}$ transformation approach, we use the property

$$\mathsf{G}((state = fetch\_st\ \&\ input = inc2) \rightarrow$$

$$\mathsf{F}(LET(v = pc)IN(XXX(state = fetch\_st\ \&\ pc = inc(inc(v))))))$$

on an abstract counter introduced in [6]. Figure 2 shows the transformation procedure for the above property. The property contains two Next_let_formulas

$$N1 \equiv (state = fetch\_st\ \&\ input = inc2)$$

and

$$N2 \equiv (LET(v = pc)IN(XXX(state = fetch\_st\ \&\ pc = inc(inc(v)))))).$$

The circuit descriptions for $N1$ and $N2$, shown in the middle of the figure, are derived by applying the rules described in the previous section. Thereafter, the property $\mathsf{G}(< N1 > \rightarrow \mathsf{F} < N2 >)$ is transformed into $\mathsf{G}(p = 1 \rightarrow \mathsf{F}(q = 1))$, which is translated by Wring into the GBA, shown on the right side of the figure.

The generated GBA consists of a state transition graph (ASM) and a set of acceptance (fairness) conditions, which will be used by the language emptiness checking algorithm described in the next section.

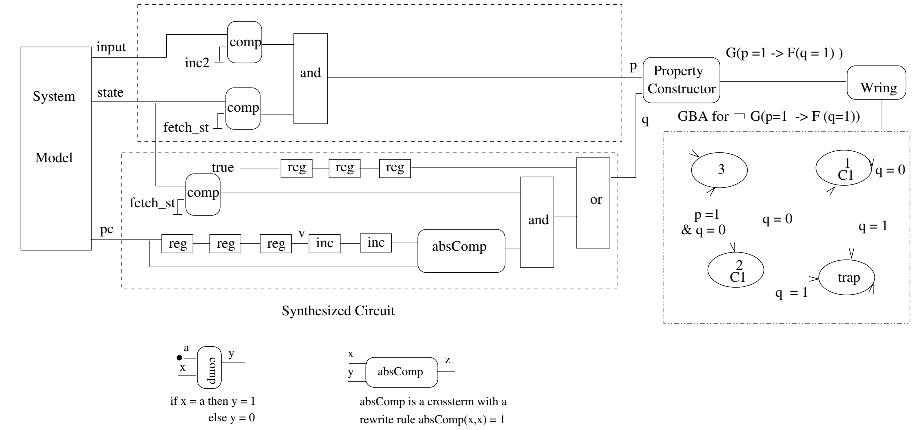

Fig. 2. Circuit synthesis and automaton generation for $\mathsf{G}((state = fetch\_st \ \& \ input = inc2) \rightarrow \mathsf{F}(LET(v = pc)IN(XXX(state = fetch\_st \ \& \ pc = inc(inc(v)))))$

5 Language Emptiness Checking Algorithm

5.1 Generic SCC Hull Algorithm

The GSH algorithm is based on the following definitions described by the $\mu-$calculus [16,17].

$$E\ p\ \mathsf{U}\ q = \mu Z.\ q\ \vee\ (p\ \wedge\ EXZ), \qquad E\ p\ S\ q = \mu Z.\ q \vee (\ p \wedge EY\ Z)$$
$$EG\ p = \nu Z.\ p \wedge EX\ Z, \qquad EH\ p = \nu Z.\ p \wedge EY\ Z$$
$$EF\ p = E\ \text{true}\ \ \mathsf{U}p, \qquad EP\ p = E\ \ \text{true}\ \ Sp,$$

where $\mu Z.\tau$ (τ stands for the $\mu-$calculus formula) denotes the least fixpoint of τ, $\nu Z.\ \tau$ is its greatest fixpoint, $EX\ Z$ denotes the direct predecessors (images) of states in the set Z and $EY\ Z$ denotes the direct successors of Z.

Let $G = (V, E)$ be a graph and $CL = \{C_1, \cdots, C_m\} \subseteq V$ a set of Büchi fairness conditions, and $T_F = \{ES_1, \cdots, ES_m, EY\}$ be a set of forward operators over V, where ES_i is defined as $\lambda Z.\ E\ Z\ S(Z\ \wedge\ c_i)$ and EY as $\lambda Z.\ Z\ \wedge\ EY\ Z$. Similarly, let $T_B = \{EU_1, \cdots, EU_m, EX\}$ be a set of backward operators over V, where EU_i is defined as $\lambda Z.E\ Z\ U\ (Z\ \wedge\ c_i)$ and EX as $\lambda Z.\ Z\ \wedge\ EX\ Z$. The GSH algorithm first computes the set of reachable states from the initial states, and then recursively removes the states that cannot be reached by fair SCCs as well as those that cannot reach fair SCCs until it reaches a fixpoint.

The GSH algorithm can be summarized as follows:

Step 1) Calculate the set of states Z reachable from the initial states.
Step 2) Fairly pick an operator τ from $T_F \cup T_B$. Apply τ to Z, and let $Z = \tau(Z)$.
Step 3) Check if Z is a fixpoint of all the operators in $T_F \cup T_B$. If yes, stop; otherwise, go to Step 2.

Remark 1. In Step 2, "fairly pick" means the operator τ is selected under two restrictions: The operator ES (or EU) cannot be selected again unless other operators in $T_F(T_B)$ make changes to Z, and operator EX (or EY) cannot be selected again if it makes no changes to Z unless other operators in the set T_F (or T_B) make changes to Z.

Remark 2. The forward operators in T_F remove the states that cannot be reached by the fair cycles, 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 T_B remove the states that cannot reach any fair cycles, 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.

5.2 EL and EL2 Algorithms

The EL [8], EL2[9], and HH [12] algorithms have been proposed as the instances of the GSH algorithm by specifying a particular order to pick operators, namely:

$$\begin{aligned} ES_1, ES_2, \cdots, ES_m, EY, \cdots, EY, ES_1, ES_2, \cdots (EL2) \\ ES_1, EY, ES_2, EY, \cdots, ES_m, EY, ES_1, EY, \cdots (EL) \\ EU_1, ES_1, \cdots, EU_m, ES_m, EX, EY, EX, EY, \cdots, EU_1, ES_1, \cdots (HH). \end{aligned}$$

In the following, we will focus on EL and EL2 algorithms, which can be implemented with only forward operators.

5.3 MDG Implementation of EL and EL2 Algorithms

Two main operators are needed for the implementation of the EL and EL2 algorithms, namely, image computation and fixpoint computation. Image computation (forward operator) is sufficient since the GSH algorithm can be implemented by using only forward operators [17]. Furthermore, since in the MDG package there is no conjunction operation with the same abstract primary variables, the operators ES_i and $\lambda Z.Z \wedge EY\ Z$ cannot be applied either. A deeper observation reveals that under the assumption of Z is forward-closed the operators ES_i and $\lambda Z.Z\ \wedge\ EY\ Z$ can be replaced by EP_i and $\lambda Z.\ EY\ Z$, respectively, where EP_i is defined by[2]

$$EP_i\ Z\ = \lambda Z.\ (E\ true\ S\ (Z\ \wedge\ C_i))$$

Following the above argument, the operators in the GSH algorithm can be replaced with $T_F = \{EP_1, EP_2, \cdots, EP_m, \lambda Z.EYZ\}$ if only forward operators are used in the GSH algorithm. Below, we will focus on how to implement the above operators in the MDG package. Note that in the above definition EP_i

[2] A complete proof can be found in [20].

still contains a conjunction operation $Z \wedge C_i$. However, since Z and C_i do not share any abstract variables (in fact C_i does not use any abstract variables), this conjunction operation is feasible with MDGs.

Image Computation (EY) - Relational Product (Relp)

The operator EY which computes the direct successors of a set of state of Z can be implemented by the MDG operator Relp. The arguments of Relp are $\{I, Z, F_T\}, (X \cup Y)$, and $Y' \rightarrow Y$, where the MDG I represents the set of inputs, Z represents the current set, and the MDG F_T represents the transition relation.

Fixpoint Computation (EP) - Reachability Analysis (ReAn)

Given a state transition graph $G = \langle T, I \rangle$, the EP operator mainly implements the computation of the set of all reachable states from a given set. Therefore,

$$EPZ = E\text{true } S\ Z = \mu Y.Z \vee (\text{true} \wedge EYY) \equiv Z \vee EYZ \vee EY^2(Z) \vee \cdots.$$

In the MDG package, the procedure ReAn(G,C) has been developed for implicit state enumeration that tests if an invariant condition C is true at the output of every set. The operator EP is implemented by ReAn(G, true).

The procedure ReAn(G, true), takes a state transition graph $G = (X, Y, F_I, F_T)$, returns the set of reachable states from the initial states F_I for any interpretation ψ. In the rest of the paper, we will use ReAn(G) as a short form for ReAn(G, True).

Remark 3. The ReAn(G) may be non-terminating when a set of states cannot be represented by a finite MDG due mainly to the presence of abstract variables and uninterpreted functions. Several solutions do exist to alleviate this problem, for example, the initial state generalization [1].

MDG EL/EL2 Algorithms

The MDG based EL and EL2 algorithms take as its arguments a state transition graph $G = (X, Y, F_I, F_T)$, where X and Y are sets of input and state variables, respectively. F_I is the set of initial states, F_T is the transition relation, and $CL = \{C_1, ..., C_m\}$ is a set of acceptance Büchi acceptance conditions C_i, represented by MDGs consisting of concrete variables.

The algorithms work as follows: First compute the set of states Z reachable from the initial states F_I , and then iteratively apply the operators $EP_1, EP_2, \ldots, EP_m, \lambda Z.EYZ, \cdots, \lambda Z.EYZ$ (EL2) or the operators $EP_1, \lambda Z.EYZ, EP_2, \lambda Z.EYZ, \ldots, EP_m, \lambda Z.EYZ$ (EL) until no changes to Z can be made. If the fixpoint is empty, the algorithm returns “Succeed”; otherwise, it returns “Failed”. The EL algorithm can be described as follows, where ζ, C, Z, I are sets and K is an integer.

```
1   MDG EL Algorithm ($G = (X, Y, F_I, F_T), CL = \{C_1, C_2, \ldots, C_m\}$ )
2    $\zeta$:= $\emptyset$; K : = 0;
3    Z := ReAn(G);
4    Do {
5      $\zeta$ := Z;
6        For C ∈ CL {
7          $F_I$ = Conj(Z,C);       /* update the set of initial states */
8          Z := ReAn(G);
9          K := K+1;
10         I :=NewInputs(K);
11         Z :=RelP($\{I, Z, F_T\}, X \cup Y, Y' \rightarrow Y$)
12       }
13       If (Z = $\emptyset$) then return “Succeed”
14   } until (Pbys($\zeta$, Z) = F)
15   return “Failed”
```

In the above algorithm, line 3 computes the set of reachable states. In fact, in this step ReAn(G) performs the operation $EP(F_I)$. Lines 4 - 14 represent the main body of the algorithm. Lines 7 - 8 compute the set of states reached by $Z \wedge C_i, i \leq m$. Lines 9 - 11 compute the image of Z, where RelP essentially performs operation $EY(Z)$. The operations are iteratively applied to Z until no changes are made to it. Obviously, in each loop, $Z_1 \subseteq Z$ and $Z \subseteq \zeta$, since the operations ReAn, Conj and Relp remove some states from the set Z. Therefore, it is sufficient to test $\zeta \subseteq Z$ for the fixpoint $\zeta = Z$. Pbys is used for these tests at line 12.

```
1   MDG EL2 Algorithm ($G = (X, Y, F_I, F_T), CL = \{C_1, C_2, \ldots, C_m\}$)
2    $\zeta$:= $\emptyset$; K :=0;
3    Z := ReAn(G);
4    Do {
5        $\zeta$ := Z;
6        For C ∈ CL {
7          $F_I$ = Conj(Z,C);       /* update the set of initial states */
8          Z := ReAn(G);
9             }
10       K := K+1;
11       I :=NewInputs(K);
12       Z1 :=RelP($\{I, Z, F_T\}, X \cup Y, Y' \rightarrow Y$);
13       While (Pbys(Z, Z1) $\neq$ F) Do{
14         Z := Z1;
15         K:= K+1;
16         I :=NewInputs(K);
17        Z1 :=RelP($\{I, Z, F_T\}, X \cup Y, Y' \rightarrow Y$);
18         }
19      If (Z = $\emptyset$) then return “Succeed”
20  } until (Pbys($\zeta$, Z) = $\mathbf{F}$)
21  return "Failed";
```

In the above algorithm, line 3 computes the set of reachable states by using ReAn(G). Lines 4 - 20 compose the main body of the algorithm. Lines 6 - 9 compute the set of states reached by all $Z \wedge C_i, i \leq m$. Lines 10 - 18 compute the image of Z until the fixpoint is reached. It is obvious that $Z_1 \subseteq Z$ and $Z \subseteq \zeta$ in each iteration, since the operations ReAn, Conj and Relp remove some states from the set Z. Therefore, it is sufficient to test $Z \subseteq Z_1$ for $Z_1 = Z$ and $\zeta \subseteq Z$ for $\zeta = Z$. Pbys is used for these tests at the lines 13 and 20.

Remark 4. As pointed out in Remark 3, ReAn(G) is non-terminating, which may lead to the non-terminating of the **EL** and **EL2** algorithms. We can apply the same approaches to solve the problem.

6 Case Study

We have implemented the proposed MDG algorithm in Prolog and integrated it into the MDG package. We have conducted a number of experiments with small benchmark designs as well as with larger case studies to test the performance of our tool. In this section, we present the experimental results of the verification of an Asynchronous Transfer Mode (ATM) switch fabric as case study. The experiments were carried out on a Sun Ultra-2 workstation with 296MHZ CPU and 768MB of memory.

The ATM switch we consider is part of the Fairisle network designed and used at the Computer Laboratory of Cambridge University [15].The ATM switch consists of an input controller, an output controller and a switch fabric. In each cycle, the input port controller synchronizes incoming data cells, appends control information, and sends them to the fabric. The fabric strips off the headers from the input cells, arbitrates between cells destined to the same port, sends successful cells to the appropriate output port controller, and passes acknowledgments from the output port controller to the input port controller.

We use an RTL design of this ATM switch fabric with 4 inputs and 4 outputs defined as 8 variables of abstract sort (n-bit) modeled in MDG-HDL [18]. In the following we discuss five sample properties, P1-5. P1 and P2 are properties checking the acknowledgment procedure, involving no data signals, while P3,P4, and P5 are properties checking the data switching signals. P1, P2 and P5 are safety properties, while P3 and P4 are liveness properties. Details about the ATM switch fabric model as well as the specification of the above properties can be found in [20]

The experimental results of the verification of these properties with the LEC are summarized in Table 1, including CPU time, memory usage and number of MDG nodes generated. To compare our approach with BDD based language emptiness checking methods, we also conducted experiments on the same ATM

Table 1. Experimental Results with MDG LEC using EL and EL2

	MDG LEC (EL)			MDG LEC (EL2)		
	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

switch fabric using the ltl_model_check option of the VIS tool [2]. 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 2 and 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 LEC was able to verify both in a few minutes for *n*-bit (abstract) data. It is to be noted that VIS uses very powerful cone-of-influence [14] model reduction algorithms, while our MDG LEC does not perform any reduction on the model. When we turned off this model reduction VIS failed to verify any of the properties on 4-bit, 8-bit and 16-bit models.

Table 2. Experimental Results with VIS using EL algorithm

	GSH 4-bit			GSH 8-bit			GSH 16-bit		
	Time (sec)	Memory (MB)	# BDD Nodes	Time sec)	Memory (MB)	# BDD Nodes	Time (sec)	Memory (MB)	# BDD Nodes
P1	27.2	52	3095941	31.2	52	3081297	36.8	52	3064133
P2	11.7	45	1670014	14.0	45	1659550	21.8	46	1699964
P3	12.4	40	1356704	899.8	629	98706620	*	*	*
P4	533.1	167	32770721	*	*	*	*	*	*
P5	14.7	44	1596773	321.8	137	35106376	*	*	*

7 Conclusion

In this paper, we introduced a new application of the MDG tool set, which implements a first-order LTL model checking algorithm. The tool, MDG LEC, accepts abstract state machines (ASM) as system models, and properties specified in a new defined first-order logic ($\mathcal{L}_{\mathcal{MDG}}{}^*$), which extends a previously developed language ($\mathcal{L}_{\mathcal{MDG}}$). We developed rules enabling the transformation of $\mathcal{L}_{\mathcal{MDG}}{}^*$ properties into generalized Büchi automata (GBA) making use of the Wring procedure. For the language emptiness checking, we adapted two instances (EL and

Table 3. Experimental Results with VIS using EL2 algorithm

	GSH 4-bit			GSH 8-bit			GSH 16-bit		
	Time (sec)	Memory (MB)	# BDD Nodes	Time sec)	Memory (MB)	# BDD Nodes	Time (sec)	Memory (MB)	# BDD Nodes
P1	27.1	52	3095941	29.5	52	3081297	36.8	52	3064191
P2	11.5	45	1670014	14.1	45	1659550	21.7	46	1699964
P3	12.5	40	1356704	899	628	98706620	*	*	*
P4	533	166	32770721	*	*	*	*	*	*
P5	13.8	44	1596773	317.6	137	35198884	*	*	*

EL2) of the generic SCC-hull (GSH) algorithm using MDG operators. Experimental results have shown that, thanks to the support of abstract data and uninterpreted functions, our MDG LEC tool outperforms existing BDD based LTL model checkers implementing EL and EL2 in VIS.

At present, we are investigating the generation of counter-examples, which, unlike BDD based approaches, is not straight forward since MDG does not support backward trace operators. We are also looking into ways to integrate model reduction algorithms (e.g., cone-of-influence) in order to improve the overall performance of the tool.

References

1. 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.
2. R. Bloem, K. Ravi, and F.Somenzi. Efficient decision procedures for model checking of linear time logic properties. In *Computer Aided Verification*, volume 1633 of *LNCS*, pages 222–235. Springer-Verlag, 1999.
3. J. Bohn, W. Damm, O. Grumberg, H. Hungar, and K. Laster. First-order-CTL model checking. In *Foundations of Software Technology and Theoretical Computer Science*, pages 283–294, 1998.
4. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.
5. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, 1997.
6. D. Cyrluk and P. Narendran. Ground temporal logic: A logic for hardware verification. In *International Conference on Compute Aided Verification*, volume 818 of *LNCS*, pages 247–259. Springer-Verlag, 1994.
7. M. Daniele, F. Giunchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. In *Computer Aided Verification*, volume 1633 of *LNCS*, pages 249–260. Springer-Verlag, 1999.
8. E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, 1986.

9. K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *7th Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 420–434. Springer-Verlag, 2001.
10. F.Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 247–263. Springer-Verlag, 2000.
11. 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, Warsaw, Poland, 1995. North-Holland.
12. 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.
13. R. Hojati and R. K. Brayton. Automatic datapath abstraction in hardware systems. In *Computer Aided Verification*, volume 939 of *LNCS*, pages 98–113. Springer-Verlag, 1995.
14. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes.* Princeton University Press, 1994.
15. I. Leslie and D. McAuley. Fairisle: an ATM network for the local area. *ACM communication review*, 19:327–336, 1991.
16. 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*, volume 1954 of *LNCS*, pages 143–160. Springer-Verlag, 2000.
17. F. Somenzi, K. Ravi, and R. Bloem. Analysis of symbolic scc hull algorithms. In *Conference on Formal Methods in Computer Aided Design*, volume 2517 of *LNCS*, pages 88–105. Springer-Verlag, 2002.
18. 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.
19. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *LNCS*, pages 238–266. Springer-Verlag, 1996.
20. F. Wang. *Automata based model checking using multiway decision graphs.* Technical Report, Concordia University, 2004. Available online at: http://www.ece.concordia.ca/∼f_wang/MDGLEC_report.ps.
21. Y. Xu, E. Cerny, X. Song, F. Corella, and O. Ait Mohamed. Model checking for a first-order temporal logic using multiway decision graphs. In *Computer Aided Verification*, volume 1427 of *LNCS*, pages 219–231. Springer-Verlag, 1998.
22. *http://hvg.ece.concordia.ca/mdg.*