TOWARDS LAMGUAGE EMPTINESS MODEL CHECKING FOR MDG

Fang Wang and Sofiène Tahar
Concordia University, ECE Dept., Montreal, Quebec, H3G 1M8, Canada
{f_wang, tahar}@ece.concordia.ca

Abstract

Automata-based model checking uses ω-automata as the unifying models of both the system and its specification, and tests the languages of automata for the system are contained in the languages of automata for the specification. MDG is a hardware verification package that uses multiway decision graphs which accepts abstract variables and uninterpreted functions. This paper, first, surveys the ω-automata based model checking approaches and multiway decision graphs functionalities, then addresses the feasibility and approaches of developing automata-based model checking in MDGs.

1. INTRODUCTION

Formal verification is being considered by the industry community as a potential complement to traditional simulation which may help combating the verification crisis of designed circuits. One of such formal verification methods is automata-based model checking [12].

Automata-based model checking uses ω-automata as the unifying models for the system and its specification, and checks whether the languages of system are contained in the languages of specification. The verification proceeds as follows: both the system and the negation of the specification are transformed into ω-automata. The former recognizes all the system execution sequences, while the latter accepts all models that violate the specification. Verification amounts to checking whether the language recognized by the synchronous product of the above two automata is empty. To find one language means the specification can be violated by the system, and this language supplies an error trace for the system designer. The implementation of this methodology uses Reduced Ordered Binary Decision Diagrams (ROBDDs), one of its problems is state explosion since it is based on Boolean logic [7].

To overcome the traditional state explosion problem of ROBDDs, Multiway Decision Graphs (MDGs) [4] have been presented as a new class of decision graphs that comprises, but is much broader than, the class of ROBDDs. The underlying logic of MDGs is a subset of many-sorted first-order logic with a distinction between concrete and abstract sorts. A concrete sort has an enumeration that is the set of individual constants while an abstract sort does not. Hence, a data signal can be represented by a signal variable of abstract sort, rather than by a vector of Boolean variables, and a data operation can be viewed as a black-box and represented by an uninterpreted function symbol. MDGs are thus much more compatible than ROBDDs for designs containing a datapath [4].

Motivated by a desire to combine the automation feature of model checking and the abstract representation of data in theorem proving, our objective is to propose an ω-automata based model checking approach using MDGs to deal with abstract variables and uninterpreted operation symbols. In this paper, we first investigate automata-based model checking methods by surveying the different approaches available in the literature. We then describe the most important features of MDG. Finally, we discuss the feasibility of embedding these approaches in the MDG tools. The rest of this paper is organized as follows: Section 2 describes the types of automata used in automata-based model checking; Section 3 surveys the various methodologies of automata-based model checking using ROBDDs; Section 4 reviews some preliminary about MDGs; Section 5 proposes two methods to embed automata-based model checking in MDG tools; Finally, Section 6 concludes the paper.

2. ω-AUTOMATA

In the field of automata-based model checking, there are three most commonly used automata: Büchi automaton, generalized Büchi automata, and L-automata. This section gives a brief review of these concepts.

Proceedings of the 2002 IEEE Canadian Conference on Electrical & Computer Engineering
0-7802-xxxx-x/02/$10 © 2002 IEEE

2.1 Büchi Automata

A Büchi Automata (BW) [12] is a tuple $\Lambda =< \Sigma, S, I, \delta, F >$, where Σ is a finite nonempty alphabet, S is a finite set of sates, $I \subseteq S$ is the set of initial states, $\delta : S \rightarrow 2^S$ is the transition function, $F \subseteq S$ is a set of acceptance states. A run r of a BW is accepted if there are some accepting states repeated infinitely often, that is, $inf(r) \cap F \neq \emptyset$, where $inf(r)$ is the set of states occurring infinitely often.

2.2 Generalized Büchi Automata

A Generalized Büchi Automaton (GBW) [12] is an extended BW with the multiple acceptance sets instead of one acceptance set of BW. A labeling GBW is a tuple $\Lambda =< S, I, \delta, F, D, L >$, where S is a finite set of sates, $I \subseteq S$ is the set of initial states, $\delta : S \rightarrow 2^S$ is the transition function, $F \subseteq 2^S$ is a set of sets of acceptance states, $F = \{F_1, F_2, \cdots, F_n\}$, D is a finite domain, and $L : S \rightarrow 2^D$ is the labeling function. A run r of a GBW is accepted if for each $F_i \in F$, $inf(r) \cap F_i \neq \emptyset$.

2.3 L-Automata

An L-automaton [7] is a finite state transition structure with acceptance condition consists of recur edges and cycle sets. An L-automaton is a tuple $\Lambda =< \Sigma, S, \delta, I, R, C >$, where Σ is a finite set of input symbols, S is a finite set of sates, $I \subseteq S$ is the set of initial states, $\delta : S \times \Sigma \rightarrow 2^S$ is the transition function, $C = (C_1, \cdots, C_k)$ is a set of K cycle sets, which define node-based acceptance conditions, and R is a set of state transition graph edges called recur edges, which impose edge-based acceptance conditions. A run r is accepted if and only if either the run r has an infinite number of edges $e \in R$, or there exists $C_i \in C$ such that $inf(r) \subseteq C_i$.

The dual of an L-automaton, called L-process, is mainly used in language containment based verification. An L-process has the same structure as an L-automaton with the dual interpreted acceptance condition. A run r is accepted by an L-process if and only if the run has no recurring edges $e \in R$ and for all $C_i \in C, inf(r) \cap \overline{C_i} = \emptyset$.

3. ω-AUTOMATA BASED MODEL CHECKING

Automata based model checking can be classified in two groups. Namely, those based on GBW and those based on L-automata.

3.1 GBW based Model Checking

GBW based model checking uses Propositional Linear Temporal Logic (PLTL) formulas as properties. The PLTL formulas first are translated into GBWs, then a language emptiness procedure is used to check if the language accepted by the product automaton is empty.

3.1.1 Constructing GBW from PLTL In [9], an efficient algorithm based on tableau construction was proposed for the construction of GBW on-the-fly. An improvement of this algorithm, based on syntactic simplification, is discussed in [13]. An extended algorithm of [9, 13], based on rewriting of *PLTL* formula and simplifying the resulting Büchi automata, was presented in [11]. Another algorithm using the classical construction [12], instead of the tableau construction of [9, 13, 11] is presented in [10]. Based on the above algorithms, there are many construction tools, such as, LTL2AUT [13], EQLTL [6], Wring [11], and LTL2BA [10].

In this paper, we focus on the tableau construction algorithm [11]. To translate a *PLTL* formula ϕ into a GBW, we use expansion rules also known as tableau rules

$$\psi_1 \ \cup \ \psi_2 = \psi_2 \vee (\psi_1 \wedge X(\psi_1 \ \cup \ \psi_2)),$$

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

where ψ_1, ψ_2 are *PLTL* formulas, $\cup$ is the *until* temporal operator, and $\mathbf{R}$ is the dual of temporal operator $\cup$. These are applied to expand ϕ until the resulting expression is a propositional formula in terms of elementary subformula of ϕ. An elementary formula is a constant, an atomic proposition, or a formula starting with X. The expanded formula, put in disjunctive normal form, is an elementary *cover* of ϕ. 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 form the next part of the term; they are *PLTL* formulas that identify the obligations that must be fulfilled to obtain an accepting run [9]. 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 [13]. The automaton is obtained by connecting each state to the states in the *cover* for its next part. The states in the elementary *cover* of ϕ 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 ψ_2.

3.1.2 Language Emptiness Checking Algorithm. Checking nonemptiness of the language of a *GBW* is equivalent to finding a *Strongly Connected Component (SCC)* that is reachable from the initial states and contains an accepting state [5]. Touati *et al.* [14] presented the first symbolic algorithm for SCC decomposition. An improvement of Touati's algorithm has been presented by Xie and Beerel [15], which takes $\Theta(n^2)$ steps in the worst case. Bloem *et. al.* improved the algorithm of [15] to $\Theta(nlogn)$ in worst case [2]. Wang *et al.* [16] used abstractions to compute an SCC decomposition of the system by refinement.

3.2 L-automata based Model Checking

In [7], Kurshan defined a modified version of finite-state automata and finite-state machines that accept sequences, called L-automata and L-process, to represent specifications and implementation, respectively. A specification is typically represented by deterministic L-automaton T and an implementation by a nondeterministic L-process P. Verification is cast in term of testing for a language containment, i.e., testing if $L(P) \in L(T)$. One of the greatest strengths of this method is its use of reduction both to control the complexity of state-space analysis and to provide a basis for hierarchical verification.

3.2.1 Language Emptiness Checking Algorithm. The algorithm for verifying $L(S) \in L(T)$ proceeds by constructing an L-process P from S and T whose language satisfies $L(P) = L(S) \text{Ł} L(T)$, that is accepting all languages of L-process that does not satisfy the task L-automata T. The language emptiness checking for L-process P. is implemented by finding a set of SCCs of the transition graph of P with the recurring edges removed and checking that each SCC is contained in some cycle sets. If there is no SCC reachable from the initial state contained in the cycle sets, then the checking succeeds; otherwise, the checking fails.

4. MULTIWAY DECISION GRAPHS (MDG)

An MDG is a finite, directed acyclic graph [3]. An internal node of an MDG can be a variable of concrete sort with its edge labels being the individual constants in the enumeration of the sort; or it can be a variable of abstract sort and its edges are labeled abstract terms of the same sort; or it can be a cross-term (whose function symbols is a cross-operator). An MDG may only have one leaf node denoted as **T**, which means all paths in an MDG are true formula. Thus, MDGs essentially represent relations rather than functions. MDGs can also represent sets of states. Like, ROBDDs, MDGs must be reduced and ordered. They obey a set of well-formed conditions, which turns MDGs into a canonical representation [4]. Algorithms for computing *disjunction, relational product (conjunction* followed by existential quantification), *pruning-by-subsumption (PbyS,* for test of set inclusion) have been implemented in the MDG software package [18]. MDG provided applications for hardware verification such as combinational circuits verification, MDG model checking and equivalence checking for two state machines.

Register-Transfer Level (RTL) hardware designs can be suitably represented by MDGs. The hardware description language that the MDG tools accept is a Prolog-style HDL, MDG-HDL, which allows the use of abstract variables for representing data signals. The MDG-HDL description is then compiled into internal MDG data structures. 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 [18].

Besides circuit description, 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 MDG applications. All of these are organized into four kinds of input files: the algebraic specification file, the circuit description file including partitioning for transition/output relations, the invariant property specification file, and the states encoding file [18].

4.1 Abstract State Machines

An abstract state machine (ASM) is a finite state machine obtained by letting some data input, state or output variables be of abstract sort. The behavior of

a state machine is defined by transition and output relations, together with its set of initial states. More details can be found in [3].

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

1. X, Y and Z are sets of the input, state, and output variables, respectively. A variable in $X \cup Y \cup Z$ is called an *ASM_variable*. Let η 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 as the next-state variables disjoint from X, Y, and Z. Given an interpretation ψ, an input vector is a ψ-compatible assignment to the set of input variables X; thus the set of input vectors (input alphabet) is Φ_X^ψ. Similarly, is the set of output vectors. A state is a ψ-compatible assignment to the set of state variables Y; hence, the state space is Φ_Y^ψ. A state ϕ can also be described by an assignment $\phi' = \phi \circ \eta^{-1} \in \Phi_Y^\psi$ to Y'.

2. F_I is a Direct Formula (*DF*) of type $U \rightarrow Y$ representing the set of initial states, where U is a set of abstract variables disjoint from $X \cup Y \cup Y' \cup Z$. Given an interpretation ψ, a state $\phi \in \Phi_Y^\psi$ is an initial state iff $\psi, \phi \models (\exists U) F_I$. Thus the set of initial states is $S_I = Set_Y^\psi(F_I) = \{\phi \in \Phi_Y^\psi \mid \psi, \phi \models (\exists U) F_I\}$.

3. F_T is a *DF* of type $(X \cup Y) \rightarrow Y'$ representing the transition relation. Given an interpretation ψ, an input vector $\phi \in \Phi_X^\psi$ and a state $\phi' \in \Phi_Y^\psi$, a state $\phi'' \in \Phi_Y^\psi$ is a possible next state iff $\psi, \phi \cup \phi' \cup (\phi'' \circ \eta^{-1}) \models F_T$. Thus the transition relation is $R_T = \{(\phi, \phi', \phi'') \in \Phi_X^\psi \times \Phi_Y^\psi \times \Phi_Y^\psi \models \psi, \phi \cup \phi' \cup (\phi'' \circ \eta^{-1}) \models F_T\}$.

4. F_O is a *DF* of type $(X \cup Y) \rightarrow Z$ representing the output relation. Given an interpretation ψ, the output relation is $R_O = \{(\phi, \phi', \phi'') \in \Phi_X^\psi \times \Phi_Y^\psi \times \Phi_Z^\psi \models \psi, \phi \cup \phi' \cup \phi'' \models F_O\}$.

4.2 Abstract First-Order Temporal Logic ($\mathcal{L}_{\mathcal{MDG}}$)

MDG Model Checking (MC) accepts properties described as an $\mathcal{L}_{\mathcal{MDG}}$ formula (*next_let_formula*) [17]. The $\mathcal{L}_{\mathcal{MDG}}$ defines an Abstract Computation Tree Logic (ACTL), a subset can be verified by using MDG MC.

The atomic formulas of $\mathcal{L}_{\mathcal{MDG}}$ are Boolean constants *True*, *False*, or equations $t_1 = t_2$, where t_1 is an *ASM_variable*, t_2 is either an *ASM_variable*, a constant, an ordinary variable, or a function of ordinary variables. The basic formulas in which only the temporal operator X is allowed, called *Next_let_formulas* are defined as follows: each atomic formula is a *Next_let_formula*, if p, q are *Next_let_formulas*, then so are: $!p$ (*not p*), $p \wedge q$ (*p and q*), $p \vee q$ (*p or q*), $p \rightarrow q$ (*p implies q*), Xp (*nexttime p*), and $\mathsf{LET}\ (v = t)\ \mathsf{IN}\ p$, where t is an *ASM_variable* and v an ordinary variable. The properties allowed in $\mathcal{L}_{\mathcal{MDG}}$ can have the following forms:

Property ::=
$\mathsf{A}(Next\_let\_formula) \mid \mathsf{AG}(Next\_let\_formula)$
$\mid \mathsf{AF}(Next\_let\_formula)$
$\mid \mathsf{A}(Next\_let\_formula)\mathsf{U}(Next\_let\_formula)$
$\mid \mathsf{AG}((Next\_let\_formula) \Rightarrow \mathsf{F}(Next\_let\_formula))$
$\mid \mathsf{AG}(Next\_let\_formula \Rightarrow (Next\_let\_formula$
$\mathsf{U}Next\_let\_formula))$

where $\mathbf{A}, \mathbf{AG}, \mathbf{AF}, \mathbf{F}, \mathbf{U}$ are temporal operators [5].

4.3 MDG Model Checking

MDG MC algorithms are presented in [17]. The properties to be verified, in MDG *MC*, are expressed by formulas in $\mathcal{L}_{\mathcal{MDG}}$ (*next_let_formula*) [17], while the systems are modeled as *ASMs* [3]. A property in $\mathcal{L}_{\mathcal{MDG}}$ holds on an ASM if and only if the property is true for all the paths starting from the initial states in the abstract computation tree [17].

Checking a property of $\mathcal{L}_{\mathcal{MDG}}$ on an ASM is carried out as follows: first build an ASM for the property to be checked, then compose it with the design ASM, finally check a "flag" on the composite machine. The implementation of MDG MC includes a parser to check property specification and automatically build ASMs for properties. Various MC algorithms are provided: *A*, *AG*, *AF*, *AU*, *AGAF*, *AGAU*, *AF* with fairness, *AGAF* with fairness, *AU* with fairness and *AGAU* with fairness. In MDG MC, the user manually chooses different algorithms for different property types. The counterexample generation feature is not provided by the current MDG MC tool.

5. AUTOMATA-BASED MODEL CHECKING USING MDG

Since MDGs accept abstract variables and uninterpreted functions, the MDG tools can deal with the state explosion problem in hardware formal verification more efficiently than ROBDDs based tools. This advantage has been declared in the MDG MC tool. Therefore, developing of automata-based model checking using MDGs is of great importance to complement the MDG tools package.

Automata-based model checking using MDGs should be able to accept Abstract first-ordered Temporal Logic (ATL), which accepts abstract variables and uninterpreted functions, as properties and ASMs as system designs. In the following, we briefly propose two automata-based model checking approaches. The first is based on GBW and the second is based on L-automata.

5.1 Lifting GBW based Model Checking to MDG

To integrate automata-based model checking based on GBW in MDG tools, we propose an approach as shown in Figure 1.

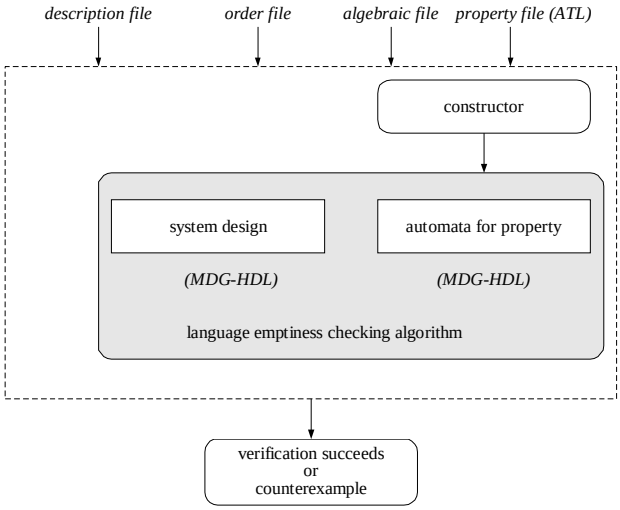

Fig. 1. Automata-based checking using MDGs.

The input files of this tool are circuit description file (MDG-HDL), custom symbol order file, algebraic file, and property file. We first define a concept of Abstract Description of GBW (*ABW*), which accepts of abstract variables and uninterpreted function systems. Then construct an ABW from the ALT. After that, we build the product ABW of the system model and the generated ABW. Finally, we should propose an algorithm to check the language emptiness of the product ABW, and to also provide a counterexample if the checking fails.

Abstract Description of GBW (ABW)

To lift the GBW from Finite State Machine (FSM) to ASM, we define an ABW as an abstract description of a GBW. An ABW is an abstract description of the GBW: $\Lambda = (X, Y, Z, F_I, F_T, F_O, F_C)$, where F_C is a set of *DFs* describing the sets of acceptance states, $F_C = \{F_C^1, F_C^2, \cdots, F_C^n\}$ given an interpretation ψ, a set of an acceptance state $\phi, \psi \models F_C^i$, i.e., $S_C^i = Set_Y^{\psi}(F_C^i)$.

5.2 Lifting L-Automata based Model Checking to MDG

The other way to check language emptiness with MDGs is to implement an L-process based model checking. For this purpose, we need first to lift the L-process from FSM to ASM, which we call Abstract Description of L-process (ADL). An ADL is obtained by restricting a set of states set as cycle sets and a set of edges as recur edges on an ASM. By the similar way to ABW, we generate an abstract description of an L-automaton for the property file and transform the system to an L-process (Figure 1). After computing the product of the system and the L-automaton generated by the constructor, we check if the language of the product is empty. If there exists a language accepted by the product, the checking fails and a counterexample is generated; otherwise the checking succeeds.

6. CONCLUSIONS

The MDG tools is a package founded on the base of multiway decision graphs, which subsume ROBBDs and accommodate abstract sorts and uninterpreted function symbols. Therefore, the MDG tools have a great potential of overcoming the state explosion problem in hardware formal verification and dealing with larger circuits with complex datapaths. In this paper, we surveyed automata-based model checking methods, reviewed MDGs and analyzed the feasibility of embedding these methods into the MDG tools. We proposed a method to implement the embedding: First, lift the concepts of GBW and L-automaton from FSM to ASM, which accepts abstract variables and uninterpreted function symbols, then develop language emptiness checking algorithms for both ABW and abstract L-process. In future work, we will implement those methods as part of the MDG tools; then test them on benchmarks; and finally, prove their soundness.

References

[1] R. Bloem, K. Ravi, and F. Somenzi, "Efficient Decision Procedures for Model Checking of Linear Time Logic Properties", *Proc. of Conference on Computer Aided Verification*, Lecture Notes in Computer Science 1633, Springer-Verlag, 1999.

[2] R. Bloem, H. N. Gabow, and F. Somenzi, "An algorithm for strongly connected component analysis in n log n symbolic steps", In *Formal Methods in Computer Aided Design*, pp. 37-54, Lecture Notes in Computer Science 1855, Springer-Verlag, 2000.

[3] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou, "Verification with abstract state machines Using MDGs", *Formal Hardware Verification*, pp. 79- 114, Lecture Notes in Computer Science 1287, Springer-Verlag, 1997.

[4] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny, "Multiway Decision Graphs for automated hardware verification", *Formal Methods in System Design*, vol.10, no.1, pp. 7-46, 1997.

[5] E. M. Clarke, J. O. Grumberg, D. Peled, *Model checking*, Cambridge, MA: MIT Press, 2000.

[6] K. Etessami, G. Holzmann, "Optimizing Büchi Automata", In*Proceedings of 11th Int. Conf. on Concurrency Theory*, 2000.

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

[8] *Cadence Affirma FormalCheck*, Cadence, 1999.

[9] R. Gerth, D. Peled, M.Vardi, and P. Wolper, "Simple on-the-fly automatic verification of Linear temporal logic", In *Protocol Specification Testing and Verification,* pp. 3-18, 1995.

[10] P. Gastin, and D. Oddoux "Fast LTL to B'uchi Automata Translation", In *Proceedings of the 13th Conference on Computer Aided Verification",* pp. 53-65, Lecture Notes in Computer Science 2102, Springer-Verlag, 2001.

[11] F. Somenzi, and R. Bloem, "Efficient Büchi automata from LTL formulae", *Proc. of Conference on Computer Aided Verification*, pp. 247-263, Lecture Notes in Computer Science 1855, Springer-Verlag, 2000.

[12] Moshe Y. Vardi, "An automata-theoretic approach to linear temporal logic", In *Logics for Concurrency: Structure versus Automata*, pp. 238-266, Lecture Notes in Computer Science 1043, Springer-Verlag,1996

[13] M. Daniele, F. Giunchiglia, and M.Y. Vardi, "Improved Automata Generation for Linear Temporal Logic", *Proc. of Conference on Computer Aided Verification*, Lecture Notes in Computer Science 1633, Springer Verlag, 1999.

[14] H.J. Touati, R.K. Brayton, and R.P. Kurshan, "Testing language containment for $\omega-$automata using BDD's", *Information and Computation,* vol.118, no.1, pp.101-109, April 1995.

[15] A. Xie, and P.A. Beerel, "Implicit enumeration of strongly connected components and an application to formal verification", *IEEE Transactions on Computer-Aided Design,* vol.19, no.10, pp.1225-1230, 2000.

[16] C. Wang, R. Bloem, G.D. Hachtel, K. Ravi, and F. Somenzi, "Devide and Compose: SCC Refinement for Language Emptiness", *International Conference on Concurrency Theory*, 2001.

[17] Y. Xu, "Model Checking for a First-order Temporal Logic Using MDG", *Ph.D. Thesie D'IRO, University of Montreal*, April 1999.

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