The Computer Journal, 47(1), © The British Computer Society; all rights reserved

Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs (MDGs)

YING XU[1], XIAOYU SONG[2], EDUARD CERNY[3] AND OTMANE AIT MOHAMED[4]

[1] *Nortel Networks, Ottawa, Canada*
[2] *Department of ECE, Portland State University, Portland, OR, USA*
[3] *D'IRO, Université de Montreal, Montreal, Canada*
[4] *Department of ECE, University of Concordia, Montreal, Canada*
Email: xuying@nortelnetworks.com, song@ee.pdx.edu, cerny@iro.umontreal.ca, ait@ece.concordia.ca

We study model checking for a first-order linear-time temporal logic. We present the computation model: abstract description of state machines (ASMs), in which data and data operations are described using abstract sort and uninterpreted function symbols. ASMs are suitable for describing Register Transfer level designs. We define a first-order linear-time temporal logic called $L_{\mathbf{MDG}}$ which supports the abstract data representations. Both safety and liveness properties can be expressed in $L_{\mathbf{MDG}}$, however, only universal path quantification is possible. Fairness constraints can also be imposed. The property checking algorithms are based on implicit state enumeration of an ASM and implemented using Multiway Decision Graphs.

Received 26 March 2002; revised 20 June 2003

1. INTRODUCTION

Symbolic model checking has proven to be an important technique for the automatic verification of hardware designs [1, 2, 3, 4]. However, these methods require the description of the design to be at the Boolean logic level, and thus in general they are not adequate for verifying circuits with large datapath because of the state explosion problem.

Being motivated by a desire to combine the automation feature of model checking and the abstract representation of data in theorem proving, we developed model checking algorithms for a few first-order linear-time temporal logic patterns. Our approach is based on a computation model called an *abstract description of state machine* (ASM) where a data value is 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 [5, 6]. ASMs can be used to describe designs at Register Transfer Level (RTL). An ASM is encoded using Multiway Decision Graphs (MDGs) [6, 7] of which Reduced Ordered Binary Decision Diagrams (ROBDDs) [8] are a special case. The verification of ASMs is based on state enumeration whose complexity is independent of the width of the datapath. Thus, the state explosion problem caused by descriptions of large datapaths at the Boolean logic level is alleviated.

While our formalization of ASMs was introduced in [6, 7], and the invariant checking on ASMs was presented in [9], in this paper we address the model checking problem for ASMs. We define the first-order linear-time temporal logic L_{MDG} and present the property checking algorithms for L_{MDG}. As part of the property checking algorithms, we also show a special handling of the next operators, i.e. using ASMs to represent the basic L_{MDG} sub-formulas in which only the next operator X is allowed (called Next_let_formulas).

To check a property p in L_{MDG} on an ASM M, we first build additional ASMs automatically for Next_let_formulas, then we compose the additional ASMs with M, and finally check a relatively simpler property on the composite machine. The property checking algorithms are based on implicit state enumeration as supported by MDGs. However, the algorithms do not always terminate. Decidability of model checking for L_{MDG}, just like decidability of reachability analysis for our ASMs, is left as an open question.

There are previous developments reported in the literature that are directly related to ours. Hungar, Grumberg, Damm *et al.* proposed a 'true symbolic model checking' technique in [10], and later improved in [11] by checking first-order-CTL (FO-CTL) properties on systems with clear separation between control and datapath. In their method, the BDD-based symbolic model checker is used to check the properties on the control part, the data part is handled by annotating each control-expanded state by a first-order formula, and BDDs are used to represent and manipulate the first-order annotations symbolically. The FO-CTL is more expressive than L_{MDG}, since only a limited nesting of temporal operators is allowed in L_{MDG}. However, their method differs from our method in the way the property checking is achieved. Basically, they showed how to cast first-order model checking into

BDD-based model checking, while our property checking algorithms are carried out directly on the first-order logic level.

Cyrluk and Narendran [12] defined a first-order temporal logic–Ground Temporal Logic (GTL), which falls in between the first-order and the propositional temporal logics. The validity problem in GTL is the same as checking a linear-time temporal logic formula for all computation paths. In [12], the authors showed that the full GTL is undecidable. They then identified a decidable fragment of GTL, consisting of $\Box p$ (always p) formulas where p is a GTL formula containing an arbitrary number of 'Next' operators, but no other temporal operators. However, they did not show how to build the decision procedure. We shall see in the following sections that the decidable fragment of GTL is actually a subset of the class of properties that we can verify.

Hojati, Brayton *et al.* proposed a concurrency model called integer combinational/sequential (ICS), which uses finite relations, interpreted and uninterpreted integer functions and predicates, and interpreted memory functions to describe hardware systems with datapath abstraction [13, 14, 15]. Verification of ICS models is performed using language containment. They showed that for a subclass of 'control-intensive' ICS models, integer variables in the model can be replaced by enumerated variables (i.e. finite instantiation) and then the property verification can be carried out at the Boolean logic level without sacrificing accuracy. They gave a linear-time algorithm for recognizing those subsets. For verifying properties of circuits with complex datapaths, i.e. the circuit contains interpreted and uninterpreted functions, finite instantiation cannot be used. Instead, they compute the set of states reachable in n steps using BDDs and check that no error exists in these n steps. Their algorithm, if it terminates, computes the set of reachable states. Thus, they can check only safety properties, while our methods can check safety properties as well as certain liveness properties.

Burch and Dill [16] used a subset of first-order logic, specifically, the quantifier-free logic of uninterpreted functions and predicates with equality and propositional connectives, for verifying microprocessor control circuitry. Velev and Bryant [17] presented a Equality Validity Checker (EVC) for a logic called Equality with Uninterpreted Functions and Memories (EUFM). The EVC translates a formula in EUFM to a propositional formula, and then evaluates the propositional formula using a Boolean satisfiability procedure. Those methods are appropriate for verification of microprocessor control because they allow abstraction of datapath values and operations. However, those methods, unlike ours, cannot verify properties involving temporal operators, in particular, liveness properties. Another relevant but differently focused approach was [18], in which Namjoshi and Kurshan showed an algorithm that constructs a finite state 'abstract' program from a given, possibly infinite state, 'concrete' program by means of a syntactic program transformation.

This paper is organized as follows: in Section 2, we first describe the formal logic used in our ASM approach, and then define the computation model, i.e. the definition of ASMs. In Section 3, we define the syntax and the semantics of L_{MDG}, as a property specification language for which we have been able to develop property checking procedures. In Section 4, we present in detail the property checking procedures. In Section 5, we show how to impose fairness constraints in our verification system and the algorithms for checking liveness properties under fairness constraints. In Section 6, we verify some properties regarding the Island Tunnel Control bench mark using our model checker and also using VIS from University of California at Berkley. We also verify several properties regarding an abstract counter in which the value of the counter is described using a variable of abstract type. We conclude the paper in Section 7.

2. ABSTRACT DESCRIPTION OF STATE MACHINES

Abstract description of state machine is a model used for describing hardware designs at the RTL. Using ASMs, 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. The model checking method based on a first-order linear-time temporal logic as developed in this paper allows to verify properties on designs represented by ASMs. Thus, it is necessary to review first the terminology related to ASMs.

2.1. A many-sorted first-order logic

As in an ordinary many-sorted first-order logic, the vocabulary consists of *sorts*, *constants*, *variables* and *function symbols* (or *operators*). Constants and variables have sorts. We deviate from standard many-sorted first-order logic by introducing a distinction between *concrete* (or *enumerated*) *sorts* and *abstract sorts*; the difference is that concrete sorts have *enumerations*, while abstract sorts do not. The enumeration of a concrete sort α is a set of distinct constants of sort α. We refer to constants occurring in enumerations as *individual constants* and to other constants as *generic constants*.

The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let f be a function symbol of type $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \to \alpha_{n+1}$. If α_{n+1} is an abstract sort then f is an *abstract function symbol*. If all the $\alpha_1, \ldots, \alpha_{n+1}$ are concrete, f is a *concrete function symbol*. If α_{n+1} is concrete while at least one of $\alpha_1, \ldots, \alpha_n$ is abstract, then f is referred to as a *cross-operator*. Both abstract function symbols and cross-operators may be *uninterpreted*, or *partially interpreted* by conditional rewrite rules. The *terms* and their *types* (*sorts*) are defined inductively as follows: a constant or a variable of sort α is a term of type α; and if f is a function symbol of type $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \to \alpha_{n+1}$, $n \geq 1$, and $A_1, \ldots, A_n$ are terms of types $\alpha_1, \ldots, \alpha_n$, then $f(A_1, \ldots, A_n)$ is a term of type α_{n+1}. 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* if it only contains: (i) the individual constants; (ii) the abstract generic constants; (iii) the abstract variables; and (iv) the terms of the form $f(A_1, \ldots, A_n)$ where f is a function symbol and $A_1, \ldots, A_n$ are concretely reduced terms. Thus, the concretely reduced terms are those that have no concrete subterms other than individual constants. A term of the form $f(A_1, \ldots, A_n)$ where f is a cross-operator and $A_1, \ldots, A_n$ are concretely reduced terms is called a *cross-term*. An *equation* is an expression $A_1 = A_2$ where A_1 and A_2 are terms of same type α. *Atomic formulas* are the equations, plus **T** (truth) and **F** (falsity). Formulas are built from the atomic formulas in the usual way using logical connectives and quantifiers.

An *interpretation* is a mapping ψ that assigns a denotation to each sort, constant and function symbol such that:

(i) The denotation $\psi(\alpha)$ of an abstract sort α is a non-empty set.
(ii) If α is a concrete sort with enumeration $\{a_1, a_2, \ldots, a_n\}$ then $\psi(\alpha) = \{\psi(a_1), \psi(a_2), \ldots, \psi(a_n)\}$ and $\psi(a_i) \neq \psi(a_j)$ for $1 \leq i < j \leq n$.
(iii) If c is a generic constant of sort α, then $\psi(c) \in \psi(\alpha)$. If f is a function symbol of type $\alpha_1 \times \cdots \times \alpha_n \to \alpha_{n+1}$, then $\psi(f)$ is a function from the Cartesian product $\psi(\alpha_1) \times \cdots \times \psi(\alpha_n)$ into the set $\psi(\alpha_{n+1})$.

Let X be a set of variables, a *variable assignment* with domain X compatible with an interpretation ψ is a function φ that maps every variable $x \in X$ of sort α to an element $\varphi(x)$ of $\psi(\alpha)$. We write Φ^{ψ}_{X} for the set of ψ-compatible assignments to the variables in X, $\psi, \varphi \models P$ if a formula P denotes truth under an interpretation ψ and a ψ-compatible variable assignment φ to the variables that occur free in P, $\models P$ if a formula P denotes truth under every interpretation ψ and every ψ-compatible variable assignment to the variables that occur free in P.

2.2. Directed formulas

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

(i) Each disjunct is a conjunction of equations of the form
$A = a$, where A is a term of concrete sort α of the form '$f(B_1, \ldots, B_n)$' (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 α, or
$w = a$, where $w \in (U \cup V)$ is a variable of concrete sort α and a is an individual constant in the enumeration of α, or
$v = A$, where $v \in V$ is a variable of abstract sort α and A is a term of type α containing no variables other than elements of U.
(ii) In each disjunct, the left-hand sides (LHSs) of the equations are pairwise distinct.
(iii) Every abstract variable $v \in V$ appears as the LHS of an equation $v = A$ in each of the disjuncts. (Note that there need not be an equation $v = a$ for every concrete variable $v \in V$.)

Intuitively, in a DF of type $U \to V$, 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 the other equations specify the values of (some of the) V variables in that case. The cases need not be mutually exclusive, nor exhaustive.

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

Let P be a DF of type $U \to V$. For a given interpretation ψ, P can be used to represent the set of vectors $\mathit{Set}^{\psi}_{V}(P) = \{\phi \in \Phi^{\psi}_{V} | \psi, \phi \models (\exists U) P\}$.

In the following sections, DFs are used for two distinct purposes: to represent sets (*viz.* sets of states as well as sets of input vectors and output vectors) and to represent relations (*viz.* the transition and output relations).

2.3. Abstract description of state machines

An ASM M is a tuple $D = (X, Y, Z, F_I, F_T, F_O)$, where

(i) X, Y and Z are sets of variables, *viz.* the input, state and output variables, respectively. 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. X, Y and Z must be disjointed from Y'.
Given an interpretation ψ, an input vector of the state machine M represented by D is a ψ-compatible assignment to the set of input variables X; thus the set of input vectors, or input alphabet, is Φ^{ψ}_{X}. Similarly, Φ^{ψ}_{Z} 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^{ψ}_{Y}. A state ϕ can also be described by an assignment $\phi' = \phi \circ \eta^{-1} \in \Phi^{\psi}_{Y}$ to the next state variables.
A variable in $X \cup Y \cup Z$ is called an *ASM_variable.*
(ii) F_I is a DF representing the set of initial states, of type $U \to Y$, where U is a set of abstract variables disjoint from $X \cup Y \cup Y' \cup Z$. Typically, F_I is a one-disjunct DF representing the set of initial states.
Given an interpretation ψ, a state $\phi \in \Phi^{\psi}_{Y}$ is an initial state iff $\psi, \phi \models (\exists U) F_I$. Thus the set of *initial states* is $S_I = \mathrm{Set}^{\psi}(F_I) = \{\phi \in \Phi^{\psi}_{Y} | \psi, \phi \models (\exists U) F_I\}$.
(iii) F_T is a DF of type $(X \cup Y) \to Y'$ representing the transition relation.
Given an interpretation ψ, an input vector $\phi \in \Phi^{\psi}_{X}$ and a state $\phi' \in \Phi^{\psi}_{Y}$, a state $\phi'' \in \Phi^{\psi}_{Y}$ is a possible next state iff $\psi, \phi \cup \phi' \cup \phi'' \circ \eta^{-1} \models F_T$. Thus the transition relation of the state machine M represented by D is

$$R_T = \{(\phi, \phi', \phi'') \in \Phi^{\psi}_{X} \times \Phi^{\psi}_{Y} \times \Phi^{\psi}_{Y} | \psi, \phi \cup \phi' \cup (\phi'' \circ \eta^{-1}) \models F_T\}.$$

(iv) F_O is a DF of type $(X \cup Y) \to Z$ representing the output relation.

Given an interpretation ψ, the output relation of the state machine M represented by D is $R_O = \{(\phi, \phi', \phi'') \in \Phi_X^\psi \times \Phi_Y^\psi \times \Phi_Z^\psi | \psi, \phi \cup \phi' \cup \phi'' | \models F_O\}$.

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

2.4. Basic algorithms of DFs

We recall the basic algorithms used in [5, 6], but here we give their definitions in terms of DFs, since these algorithms will be needed later in the model checking procedures.

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

This algorithm is mainly used to compute the union of sets of states.

Conjunction. The conjunction algorithm takes as inputs a set of DFs $P_i, 1 \leq i \leq n$, of types $U_i \to V_i$ and produces a DF $R = \mathrm{Conj}(\{P_i\}_{1\leq i\leq n})$ of type $((\bigcup_{1\leq i\leq n} U_i)\backslash(\bigcup_{1\leq i\leq n} V_i)) \to (\bigcup_{1\leq i\leq n} V_i)$ such that $\models R \Leftrightarrow (\bigwedge_{1\leq i\leq n} P_i)$. A precondition of this operation is that V_i and V_j $(1 \leq i < j \leq n)$ must have disjoint sets of abstract variables.

This algorithm is used to extract a common subset from sets of states.

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

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

The algorithm computes the conjunction of the P_i, existentially quantifies the variables in E, and applies the renaming substitution η. The type of the result R is then

$$\left(\left(\bigcup_{1\leq i\leq n} U_i\right)\Big\backslash\left(\bigcup_{1\leq i\leq n} V_i\right)\right) \to \left(\left(\left(\bigcup_{i\leq i\leq n} V_i\right)\Big\backslash E\right) \cdot \eta\right).$$

In our property checking procedures, this algorithm is used to compute the set of states reachable in one transition from one set of states.

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

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

The disjuncts that are removed from P are *subsumed* by Q, hence the name of the algorithm. If R is **F**, then it follows tautologically that $\models P \Rightarrow (\exists U)Q$.

This algorithm is used to check whether a set of states is a subset of another set of states. Let P_1, P_2 be two DFs of type $U \to Y$. Then for a given interpretation ψ, the two sets of states represented by P_1, P_2 are respectively $S_1 = \mathrm{Set}^\psi(P_1) = \{\phi \in \Phi_Y^\psi | \psi, \phi \models (\exists U)P_1\}$ and $S_2 = \mathrm{Set}^\psi(P_2) = \{\phi \in \Phi_Y^\psi | \psi, \phi \models (\exists U)P_2\}$. We say that P_1 and P_2 are *equivalent* DFs (in that case, for any ψ, S_1 and S_2 are equivalent sets) if $\textbf{PbyS}(P_1, P_2) = \textbf{F}$ and $\textbf{PbyS}(P_2, P_1) = \textbf{F}$.

3. A FIRST-ORDER LINEAR-TIME TEMPORAL LOGIC: L_{MDG}

Given a description of an ASM, and a set of *ordinary variables* which are available for use in the specification of a property to be verified, the *atomic formulas* of L_{MDG} are Boolean constant **T**, **F**, or equations $t_1 = t_2$, where t_1 is an ASM_variable, t_2 is an ASM_variable, or a constant, or an ordinary variable, or a function of ordinary variables. The *Next_let_formulas* are defined as follows:

(i) each atomic formula is a Next_let_formula;
(ii) if p, q are Next_let_formulas, then so are: $!p$ (not p), $p\&q$ (p and q), $p|q$ (p or q), $p \to q$ (p implies q), $\mathbf{X}p$ (next-time p) and **LET** $(v = t)$ **IN** p, where t is an ASM_variable, v an ordinary variable. (Note: the **LET** construct allows us to use an ordinary variable v to remember the value of an ASM_variable t at the current state.)

We allow the formula **LET** $(v_1 = t_1)\&\cdots\&(v_n = t_n)$ **IN** p as a shorthand for **LET** $(v_1 = t_1)$ **IN** ((**LET** $(v_1 = t_1)$ **IN** (...... **LET** $(v_n = t_n)$ **IN** p))). We call $(v_1 = t_1)\,\&\cdots\&\,(v_n = t_n)$ a *Let_equation*.

The properties allowed in L_{MDG} can have the following forms:

Property ::=
Next_let_formula
| **G** (Next_let_formula)
| **F** (Next_let_formula)
| (Next_let_formula) **U** (Next_let_formula)
| **G** ((Next_let_formula) ⇒ (**F** (Next_let_formula)))
| **G** ((Next_let_formula) ⇒ ((Next_let_formula) **U** (Next_let_formula)))

3.1. Semantics of L_{MDG}

A path π is a sequence of states. We use π^i to denote a path starting from π_i where π_i denotes the ith state in π. All the formulas in L_{MDG} are path formulas. We write $\pi, \sigma \models p$ to mean that a path formula p is true at path π under a ψ-compatible assignment σ to the ordinary variables. We use $Val_{\phi\cup\sigma}(t)$ to denote the value of term t under a ψ-compatible assignment ϕ to the state, input and output variables, and a ψ-compatible assignment σ to the ordinary variables. We then define $\models$ inductively as follows:

(i) $\pi, \sigma \models t_1 = t_2$ iff $Val_{\pi 0 \cup \sigma}(t_1) = Val_{\pi 0 \cup \sigma}(t_2)$.
(ii) $\pi, \sigma \models \mathbf{LET}(v = t)$ **IN** p iff $\pi, \sigma' \models p$ where $\sigma' = (\sigma \backslash \{(v, \sigma(v))\}) \cup \{(v, Val_{\pi 0 \cup \sigma}(t))\}$.
(iii) $\pi, \sigma \models !p$ iff it is not the case that $\pi, \sigma \models p$.
(iv) $\pi, \sigma \models p \,\&\, q$ iff $\pi, \sigma \models p$ and $\pi, \sigma \models q$.
(v) $\pi, \sigma \models p|q$ iff $\pi, \sigma \models p$ or $\pi, \sigma \models q$.
(vi) $\pi, \sigma \models p \rightarrow q$ iff $\pi, \sigma \models !p$ or $\pi, \sigma \models q$.
(vii) $\pi, \sigma \models \mathbf{G}p$ iff $\pi^1, \sigma \models p$.
(viii) $\pi, \sigma \models \mathbf{G}p$ iff $\pi^j, \sigma \models p$ for all $j \geq 0$.
(ix) $\pi, \sigma \models \mathbf{F}p$ iff $\pi^j, \sigma \models p$ for some $j \geq 0$.
(x) $\pi, \sigma \models p\mathbf{U}q$ iff for some $k \geq 0, \pi^k, \sigma \models q$, and $\pi^j, \sigma \models p$ for all j $(0 \leq j < k)$.

Given a property in L_{MDG} regarding an ASM under a given interpretation ψ, the property holds on the ASM iff the property is true for every path π such that π_0 is an initial state and, for every i, there is a transition from π_i to π_{i+1} from some ψ-compatible assignment to the input variables.

4. MODEL CHECKING FOR PROPERTIES IN L_{MDG}

Our approach to model checking is to build automatically additional ASMs that represent the Next_let_formulas appearing in the property to be verified, connect these additional ASMs to the original one, and then check a simpler property on the composite machine [19].

Given a Next_let_formula P regarding an ASM $D = (X, Y, Z, F_I, F_T, F_O)$, an ASM $D_p = (Xp, Yp, Zp, F_{Ip}, F_{Tp}, F_{Op})$ can be constructed to represent the Next_let_formula. The input variables of D_p are the ASM_variables of D which appear in the property, i.e. $Xp \subseteq X \cup Y \cup Z$. They represent the values at the ‘current’ cycle. Let n be the maximum nesting number of **X** operators in the property. The set of state variables *Yp* and the transition relation F_{Tp} are constructed so as to ‘remember’ the values of input variables of D_p or the results of comparison of the variables in the past n (or less than n) cycles. The set of the state variables of D_p contains a special state variable of Boolean type, *Flag*, which indicates the truth of the Next_let_formula one cycle earlier. The initial set of states F_{Ip} are assigned differently depending on which property template the Next_let_formula P corresponds to. The general idea is that the initial states of D_p should not affect the result of verifying P on the original ASM D. There is no output from D_p, i.e. *Zp* is empty. Hence, there is no output relation either. The details of an algorithm for constructing an ASM representing a Next_let_formula can be found in [20].

In the following subsections, we describe algorithms for verifying the various forms of the formulas in L_{MDG}. When our property checking algorithms report success to a query, then the property holds for an ASM under any interpretation. It is possible that a property holds for the ASM under the intended interpretation of the abstract function symbols and constants, but not under every interpretation. In that case, the algorithm, if it terminates, will return a false negative answer with respect to the original, non-abstracted problem. However, if all the data operations are viewed as black boxes, a property is expected to hold for every interpretation; it is in this sense that we say that our algorithms are applicable to designs where data operations are viewed as black boxes.

4.1. Verification of G(Next_let_formula)

To verify a property in the form of G(Next_let_formula) on an ASM D, we first build an additional ASM D_p with the special state variable *Flag* to represent the Next_let_formula, and then construct a composite machine $M = (X_m, Y_m, Z_m, G_I, G_T, G_O)$, where

(i) $X_m = X$ is the set of the input variables of D;
(ii) $Y_m = Y \cup Yp$ is a set of the state variables, containing both the variables in Y and *Yp*; however, since $M$is a composite machine (the states of D_p are derived from D) rather than the product machine of D and D_p, under each interpretation ψ, the state space of M is actually a subset of $\Phi_Y^\psi \times \Phi_{Y_P}^\psi$;
(iii) $Z_m = Z$ is the set of the output variables of D;
(iv) $G_I = F_I \wedge F_{Ip}$ is a DF of type $U \rightarrow Y_m$ representing the set of initial states of M;
(v) $G_T = F_T \wedge F_{Tp}$ is the abstract description of the transition relation of M;
(vi) $G_O = F_O$ is the abstract description of the output relation of M.

We then transform the problem to verifying $Flag = 1$ on each reachable state.

For example, to check the property $\mathbf{G}(req = 1 \rightarrow \mathbf{LET}(v = Din)\ \mathbf{IN}\ (\mathbf{X}\ (Dout = v)))$ on an ASM D, we build a composite ASM as shown in Figure 1, and then perform reachability analysis and in each state check that $Flag = 1$.

The algorithm to check a property in the form of $\mathbf{G}(Flag = 1)$ is as follows:

```
Check_G($M, C$)
/* $C$ is the DF $Flag = 1$. */
/*$\eta'$ is the function that maps each state variable of $M$ to
the corresponding next-state variables. */
begin
  $R := G_I$; $Q := G_I$; $K := 0$;
  loop
    $P :=$ PbyS($Q, C$);
    if $P \neq$ F then return failure; /* if the property is
        not satisfied, report failure */
    $K := K + 1$;
    $I :=$ Fresh($X_m, K$); /*generate input values */
    $N :=$ RelP($\{I, Q, G_T\}, X_m \cup Y_m, \eta'$);   /*
        compute next states */
    $Q :=$ PbyS($N, R$);   /* compute frontier set of
        states */
    if $Q =$ F then return success;   /* if fixpoint
        reached, report success */
    $R :=$ Disj($R, Q$);   /* compute all states reached
        so far */
  end loop;
end;
```

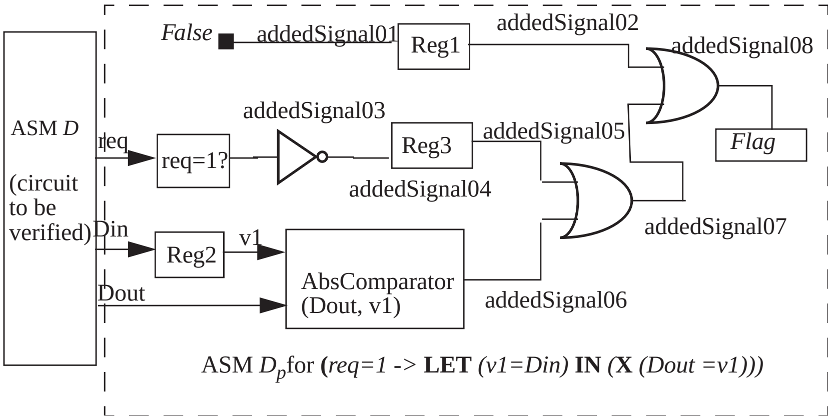

FIGURE 1. The composite machine of ASM D and ASM D_p.

If the set of initial states represented by G_I does not satisfy the property we report failure. Otherwise, we compute the next new states and add them to those already visited until a fixpoint is reached. At each iteration, we verify the property on the newly generated states.

To check a property in the form of Next_let_formula, we construct a composite ASM in the same way as in the case of G(Next_let_formula), and then we verify that $Flag = 1$ on the states reached in $n + 1$ transitions from the initial states, where n is the maximum nesting depth of the X operators in the property, and the 1 cycle delay is caused by the register associated with *Flag*.

4.2. Verification of (Next_let_formula)U(Next_let_formula)

We use additional ASMs to represent the Next_let_formulas and then transfer the problem to checking ($FlagP = 1$) $\mathbf{U}(FlagQ = 1)$ on the composite machine.

```
Check_U$(M, C_p, C_q)$
/*M is the composite machine */
/*$G_I$ is the set of initial states of $M$ */
/* $G_T$ is the transition relation of $M$ */
/* $C_p$ is the DF containing $FlagP = 1$.   $C_q$ is the DF
containing $FlagQ = 1$ */
begin
$\Sigma := \Phi$; /* $\Sigma$ is a set containing DFs with each DF
    representing the set of states satisfying $FlagP = 1$ but
    not $FlagQ = 1$ at each transition step */
$P := G_I$;
$K := 0$;
loop
  $Q :=$ PbyS $(P, C_q)$;   /*remove from $P$ states with
    $FlagQ = 1$*/
  if $Q :=$ F return success;
  if $\exists T \in \Sigma$, PbyS$(T, Q) =$ F return failure; /*This
    step checks if DF $Q$ covers any one of the DFs in $\Sigma$,
    i.e. for each DF $T$ in $\Sigma$, PbyS$(T, Q) =$ F is checked
    to detect a cycle in which $FlagP = 1$ is true but
    $FlagQ = 1$ never becomes true. If there is a cycle,
    then failure is reported*/
  $R =$ PbyS$(Q, C_p)$;   /*remove from $Q$ states with
    $FlagP = 1$*/
  if $R \neq$ F return failure;
  $\Sigma := \Sigma \cup \{Q\}$;  /*add DF $Q$ as an element into $\Sigma$ */
  $K := K + 1$;
  $I :=$ Fresh$(X_m, K)$; /* generate input values */
  $P :=$ RelP$(\{I, Q, G_T\}, X_m \cup Y_m, \eta')$;  /* compute
    next states */
end loop;
end;
```

The above algorithm removes from the reached set of states those states satisfying $FlagQ = 1$. If the leftover Set(Q) is empty, then the algorithm stops by reporting success. Otherwise, if there is at least one cycle where states keep satisfying $FlagP = 1$, i.e. $FlagQ = 1$ never becomes true, then there is at least one path starting from the initial state where $p\mathbf{U}q$ does not hold, it stops and reports failure. Otherwise, it checks whether all the states in Set(Q) satisfy $FlagP = 1$. If there are some states where $FlagP = 1$ does not hold, which means that there are some path(s) on which $FlagP = 1$ does not hold in every state before a state satisfying $FlagQ = 1$ is reached, then it also stops and reports failure. Otherwise, it computes the next states reachable from Set(Q) and repeats the process.

To check a property in the form $\mathbf{G}(c \Rightarrow p\mathbf{U}q)$ where c, p and q are Next_let_formulas on machine D, we need to build a composite machine M from D, an ASM representing c, an ASM representing p and an ASM representing q, and transfer to checking the property $\mathbf{G}((FlagC = 1) \Rightarrow ((FlagP = 1)\mathbf{U}(FlagQ = 1)))$ on M. We then do reachability analysis to obtain all the reachable states of M (represented by W), collect from W the states satisfying '$FlagC = 1$' ($V :=$ **Conj**(W, C_c) where C_c is a DF containing $FlagC = 1$), and finally apply the algorithm **Check_U** with the set V as the set of initial states.

A property in the form of **F**(Next_let_formula) can be verified by checking **TU**(Next_let_formula) using the **Check_U** algorithm.

5. VERIFICATION OF LIVENESS PROPERTIES WITH FAIRNESS CONSTRAINTS

5.1. Fairness constraints

When verifying liveness properties, one is usually interested only in the so-called fair infinite computation paths. A *fair path* is a computation path along which the states satisfy each fairness condition infinitely often.

In the literature, different methods for specifying fairness constraints have been developed for CTL model checking [21] and for language containment using L-automata [22, 23].

In our method, we impose fairness constraints using a subset of the criteria employed in the method based on language containment, namely, by specifying cycle sets. Let $H_i, i = 1, \dots, n$, be n 'exception' conditions, and S_ω the set of infinitely repeating states along a computation path. If at least one H_i holds on all states in S_ω, then the computation path is not fair and need not satisfy the property under investigation. That is, only those computation paths along which the states satisfy every $!(H_i)$ infinitely often are considered. Therefore, $!(H_i)$ $(1 \le i \le n)$ can be viewed as the fairness constraints. We call the formula representing the exception condition H_i an *H_formula*. The syntax of an H_Formula is as follows:

(i) the equation $x = y$ is an H_Formula, where x is an ASM_variable and y either an ASM_variable or a constant.

(ii) if p, q are H_Formulas, then so are: $!p$ (not p), $p \& q$ (p and q), $p|q$ (p or q), $p \rightarrow q$ (p implies q), $\mathbf{X}p$ (next-time p).

5.2. Verification of $p\mathbf{U}q$ with fairness constraints

To verify that $p\mathbf{U}q$ (where p and q are Next_let_formulas) holds for the initial states of an ASM D under the fairness constraints $!H_1, !H_2, \dots, !H_n$, we build additional ASMs to represent p, q and $H_i (1 \le i \le n)$, and then transfer the problem to checking $(\mathit{FlagP} = 1)\mathbf{U}(\mathit{FlagQ} = 1)$ on the initial states of the composite machine derived from D and the additional ASMs. The algorithm for verifying $(\mathit{FlagP} = 1)\mathbf{U}(\mathit{FlagQ} = 1)$ under fairness constraints $!(\mathit{FlagH}_i = 1)$ $(1 \le i \le n)$ is as follows:

```
(1) Check_U_fair($M, C_p, C_q, H_1, \dots, H_n$)
/* $M$ is the composite machine, */
/* $G_I$ is the set of initial states of $M$, */
/* $G_T$ is the transition relation of $M$,   */
/* $C_p$ is the DF containing $FlagP = 1$, */
/* $C_q$ is the DF containing $FlagQ = 1$, */
/* $H_i$ $(1 \le i \le n)$ is the DF representing formula
$FlagH_i = 1$. */
(2) begin
(3) $\Sigma := \Phi$; /* $\Sigma$ is a set containing DFs with each DF
    representing the set of states satisfying $FlagP = 1$ but
    not $FlagQ = 1$ at each transition step */
(4) $P := G_I$;
(5) $K := 0$;
(6) loop1
(7)   $S_{notq}$ := PbyS $(P, C_q)$;   /* remove from $P$ states
      with $FlagQ = 1$ */
(8)   if $S_{notq} = \mathbf{F}$ then return success;
(9)   if $\exists T \in \Sigma$, PbyS$(T, S_{notq}) = \mathbf{F}$ then return failure;
      /* This step checks if DF, $S_{notq}$ covers any one of the
      DFs in $\Sigma$ i.e. for each DF $T$ in $\Sigma$, PbyS$(T, S_{notq}) = \mathbf{F}$
      is checked to detect a cycle. If there is a cycle, then
      failure is reported */
(10)  $R$ = PbyS $(S_{notq}, C_p)$;  /* remove from $S_{notq}$ states
      with $FlagP = 1$ */
(11)  if $R \ne \mathbf{F}$ then return failure;
(12)  $\Sigma := \Sigma \cup \{S_{notq}\}$;  /* add DF $S_{notq}$ as an element
      into $\Sigma$ */
(13)  $S_1 := S_{notq}$;
(14)  for $i = 1$ to $n$ do
(15)    $S_{notH}$ := PbyS$(S_1, H_i)$; /* remove from $S_1$ states
        with $FlagH_i = 1$*/
(16)    $S_2$ := Conj$(S_1, H_i)$; /* $S_2$ represents the states in $S_1$
        with $FlagH_i = 1$ */
(17)    if $S_2 = \mathbf{F}$ then $S_{4notq} = \mathbf{F}$;
(18)    if $S_2 \ne \mathbf{F}$ then begin
(19)      $S_3 := S_2$; $S_f := S_2$; $L := 0$;
(20)      loop2 /* to compute all the states reachable from $S_2$
          with $FlagH_i = 1$ */
(21)        $L := L + 1$;
(22)        $I_2$ := Fresh$(X_m, L)$; /* generate new input
            values */
(23)        $N_1$ := RelP$(\{I_2, S_f, G_T\}, X_m \cup Y_m, \eta')$; /* com-
            pute next states */
(24)        $N_2$ := PbyS$(N_1, C_q)$; /* remove from $N_1$ the
            states with $FlagQ = 1$ */
(25)        $N_3$ := Conj$(N_2, H_i)$; /* pick from $N_2$ the states
            with $FlagH_i = 1$ */
(26)        if PbyS$(N_3, C_p) \ne \mathbf{F}$ then return failure; /* if the
            states in $N_3$ do not satisfy $FlagP = 1$, report failure
            */
(27)        $S_f$ := PbyS$(N_3, S_3)$; /* compute the frontier set
            of states */
(28)        if $S_f = \mathbf{F}$ then exit loop2; /* if all the states
            reachable from $S_2$ have been visited, exit loop 2 */
(29)        $S_3$ := PbyS$(S_3, S_f)$;
(30)        $S_3$ := Disj$(S_3, S_f)$; /* add the states of $S_f$ to $S_3$
            */
(31)      end loop2;
(32)      $I_3$ := Fresh$(X_m, L)$; /* generate new input
          values */
(33)      $S_{41}$ := RelP$(\{I_3, S_3, G_T\}, X_m \cup Y_m, \eta')$; /*
          compute the next states of $S_3$ */
(34)      $S_4$ := PbyS$(S_{41}, H_i)$; /* remove from $S_{41}$ the states
          with $FlagH_i = 1$ */
(35)      $S_{4notq}$ := PbyS$(S_4, Cq)$;
(36)      if PbyS$(S_{4notq}, Cp) \ne \mathbf{F}$ then return failure;
(37)    end_if
(38)    $S_1$ := Disj$(S_{4notq}, S_{notH})$;
(39)  end_for
(40)  $K := K + 1$;
(41)  $I_1$ := Fresh$(X_m, K)$; /* generate input values */
```

```
(42)  $S := \mathbf{RelP}(\{I_1, S_1, G_T\}, X_m \cup Y_m, \eta')$; /* compute
      next states */
(43)  end loop1
(44) end
```

In this algorithm, Σ is a set containing the DFs representing each a set of states not satisfying $FlagQ = 1$ on the fair computation paths after a transition step, P represents the frontier set of states to be explored further, and n is the number of fairness constraints.

In loop1, Lines (7–12), S_{notq} represents the set of states in P not satisfying $FlagQ = 1$. If S_{notq} is empty, then the computation stops by reporting success; otherwise, if S_{notq} covers any set in P, which means there is at least one cycle that is not one of the cycle sets, and the states in the cycle do not satisfy $FlagQ = 1$, then the algorithm stops and reports failure. If no cycle is detected, then we check whether the states in S_{notq} satisfy $FlagP = 1$. If not then report failure; if yes, then S_{notq} is added to Σ and the computation continues (Lines 10–12).

Lines (13–39) form a loop which is executed n times. This loop deals with each exception condition. At every i-th ($1 \leq i \leq n$) iteration, S_2 represents the set of states in S_1 that satisfy the excepting condition $FlagH_i = 1$, and S_{notH} represents the set of states in S_1 that does not satisfy $FlagH_i = 1$. If S_2 is not empty, the algorithm computes S_3 (loop 20–31). This set represents all states that are reachable from S_2 by any number of transition steps and that all states satisfy $FlagH_i = 1$ and $FlagP = 1$, but do not satisfy $FlagQ = 1$. In other words, S_3 could contain cycles which are formed by the states satisfying $FlagH_i = 1$ and $FlagP = 1$ but not $FlagQ = 1$. (The way to compute S_3 is the same as the reachability analysis.) Then, one more transition is done to compute the set of states reachable by one transition step from the states of S_3, but not satisfying $FlagH_i = 1$, and these states are stored in S_4. S_{4notq} represents the set of states in S_4 that do not satisfy $FlagQ = 1$. If this set contains at least one state that does not satisfy $FlagP = 1$, then report failure (Line 36). S_1 is the union of the set of states represented by S_{4notq} and S_{notH} at each iteration of the loop.

In Lines (40–42), P is computed to represent the states reachable in one transition step from the states in S_1. The computation continues in loop 1 with P being the new frontier set of states to be checked.

In Figure 2, we show an example that illustrates how this algorithm works. Suppose we wish to verify $(FlagP = 1)\mathbf{U}$ $(FlagQ = 1)$ under the fairness constraint $!(FlagH_1 = 1)$ on the state transition graph given in Figure 2. We also indicate the values of $FlagP$, $FlagQ$ and $FlagH_1$ in each state. We shall see that the algorithm stops and reports success at the 3rd iteration in loop1. However, checking $(FlagP = 1)$ $\mathbf{U}(FlagQ = 1)$ without the fairness constraint would fail on the path s1 → s2 → s3 → s2 → s3 → s2 → s3 $\cdots$.

The **Check_U_fair** algorithm is conservative, i.e. it requires that for every path, $FlagP = 1$ is satisfied on all the states along the path before a state satisfying $FlagQ = 1$ is reached. Along some paths, if the states repeating forever are covered by a cycle set and there is no other state reached by those states as shown in Figure 3, **Check_U_fair** will report failure. However, it is not necessary that $FlagP = 1$ holds on these states, since this path should not even be considered. Thus **Check_U_fair** may give a false negative answer. In real systems, this situation happens rarely.

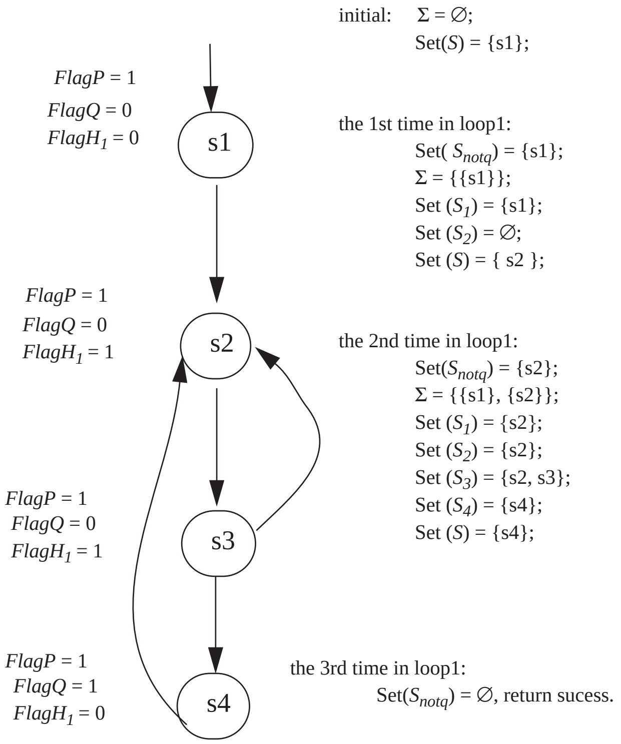

FIGURE 2. Example of checking $(FlagP = 1)\mathbf{U}(FlagQ = 1)$ under fairness constraint $!(FlagH_1 = 1)$.

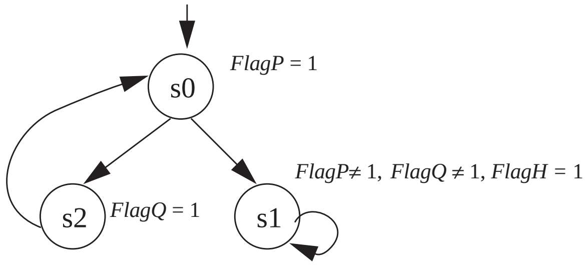

FIGURE 3. Example of a false negative answer when verifying $(FlagP = 1)\mathbf{U}(FlagQ = 1)$ under the fairness constraint $!\,(FlagH = 1)$.

To check $\mathbf{G}(c \Rightarrow p\mathbf{U}q)$ where c, p, q are Next_let_ formulas under the fairness constraints $!H_1, !H_2, \ldots, !H_n$ on an ASM D, we build a composite machine M from D and ASMs representing $c, p, q, H_i (1 \leq i \leq n)$, and then transfer the problem to checking $\mathbf{G}((FlagC = 1) \Rightarrow ((FlagP = 1)\mathbf{U}(FlagQ = 1)))$ on M under the fairness constraints $!(FlagH_i = 1)\,(1 \leq i \leq n)$. We then do reachability analysis to get all the reachable states of M (represented by W), collect from W the states satisfying ‘$FlagC = 1$’ ($V := \mathbf{Conj}(W, C_c)$ where C_c is a DF containing $FlagC = 1$), and finally apply the algorithm **Check_U_fair** with the set V as the set of initial states.

To verify that $\mathbf{F}p$ (where p is a Next_let_formula) under fairness constraints, we verify ($\mathbf{TU}q$). The method will not produce any false negative answer since $\mathbf{T}$ is satisfied by any state in this case.

6. EXPERIMENTAL RESULTS

To show how to express properties in L_{MDG}, and how to use our model checker, we use the Island Tunnel Controller (ITC) [24] and the Abstract Counter [12] as examples. Although the two examples are small and do not represent the scale of designs that the MDG-based model checker can verify, they are ideal for illustration purposes. From the two examples, we can see how the ASMs are used to describe design models, and how the properties can be stated using L_{MDG}. We also carried out the same verification using the ROBDD-based verification tool VIS [25]. Both tools showed the same verification result. However, using the MDG-based method, we were able to use abstract variables that describe the datapath and the first-order temporal logic to state properties, hence, the performance of the MDG-based model checker is much better than that of VIS.

6.1. Checking properties of the ITC

The ITC was originally introduced by Fisler and Johnson [24] to illustrate the notation of a heterogeneous logic system supporting diagrams as logic entities, however, no verification experiments were performed.

6.1.1. The ITC

Generally speaking, the ITC controls the traffic lights at both ends of a tunnel based on the information collected by sensors installed at both ends of the tunnel: there is one lane tunnel connecting the mainland to an island, as shown in Figure 4. At each end of the tunnel, there is a traffic light. There are four sensors for detecting the presence of vehicles: one at the tunnel entrance (*ie*) and one at the tunnel exit on the island side (*ix*), and one at the tunnel entrance (*me*) and one at the tunnel exit on the mainland side (*mx*). It is assumed that all cars are finite in length, that no car gets stuck in the tunnel, that cars do not exit the tunnel before entering the tunnel, that cars do not leave the tunnel entrance without travelling through the tunnel, and that there is sufficient distance between two cars such that the sensors can distinguish the cars.

In [24], one more constraint is imposed: 'at most 16 cars may be on the island at any time'. The number '16' 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'. In our ASM approach, we have the luxury to model an abstract datapath, hence, we used an abstract variable to describe the counter n. For ROBDD-based verification methods, like VIS, a particular instance of n has to be given.

Fisler and Johnson proposed a specification of ITC using three communicating controllers and two counters as shown in Figure 5. The island light controller (ILC) has four states: *green*, *entering*, *red* and *exiting*. The outputs *igl* and *irl* control the green and red lights on the island side, respectively; *iu* indicates that the cars from the island side are currently occupying the tunnel, and *ir* indicates that ILC is requesting the tunnel. The input *iy* requests the ILC to release control of the tunnel, and *ig* grants control of the tunnel from the island side. A similar set of signals is defined for the mainland light controller (MLC). The tunnel controller (TC) processes the requests for access issued by the ILC and MLC. The island counter and the tunnel counter keep track of the numbers of cars currently on the island and in the tunnel, respectively. For the TC, at each clock cycle, the counter *tc* is increased by 1 depending on $tc+$ or decremented by 1 depending on $tc-$ 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.

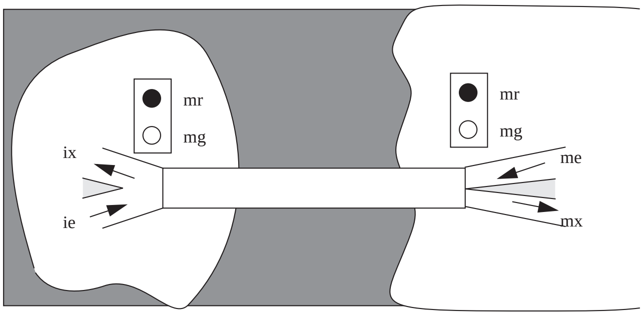

FIGURE 4. The ITC.

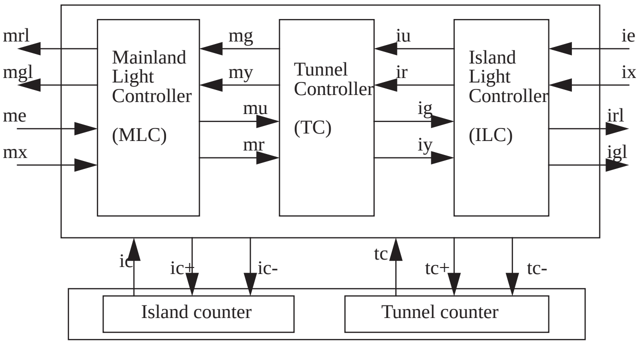

FIGURE 5. The specification of the ITC.

In [24], Fisler and Johnson proposed a set of properties that the ITC design should satisfy. In the next section, we will show how those properties are specified in L_{MDG}, and the CPU time and memory used for verifying the properties using the MDG package.

6.1.2. Property checking using the MDG package

We first create an ASM model representing the ITC design which could be read by the MDG verification system. We created modules representing ILC, MLC, TC and the counters as specified. All the signals are described using concrete variables, except that two state variables of abstract sort WORDN for n-bit word are used to describe the island counter (*ic*) and the tunnel counter (*tc*). The uninterpreted function *inc* of type $WORDN \rightarrow WORDN$ is used to describe the operation of incrementation by 1, and *dec* of the same type to describe the decrementation by 1. The environment (ENV) is built in such a way that it allows a non-deterministic choice of values on the primary inputs *ie*, *me*, *ix* and *mx*.

The following properties were verified on the ITC design:

PROPERTY 1. *The lights at both entrances of the tunnel do not show green at the same time*.

This is a typical safety property that a traffic light controller should satisfy. This property is described in the specification language L_{MDG} as follows:

$$\text{G}(!\,((\text{igl} = 1)\ \&\ (\text{mgl} = 1)));$$

PROPERTY 2. *The island counter is never ordered to increment and decrement simultaneously:*

$$\text{G}(!\,((\text{ic}- = 1)\ \&\ (\text{ic}+ = 1)));$$

PROPERTY 3. *The tunnel counter behaves properly if ordered to increment and decrement simultaneously*.

$$\begin{aligned}&\text{G}(\,((\text{tc}+ = 1)\ \&\ (\text{tc}- = 1))\\&\quad\rightarrow (\text{LET}\ (\text{v} = \text{tc})\ \text{IN X}\ (\text{tc} = \text{v})));\end{aligned}$$

We used an ordinary variable v to remember the value of *tc* at the current state, and compare the value of *tc* at the next state with v. The property states that if both the signals *tc*+ and *tc*− are set, then the value of *tc* should not change from the current state to the next state. Rewrite rules which interpret *dec*(*inc*(v)) and *inc*(*dec*(v)) as v are used in the verification of this property.

PROPERTY 4. *The tunnel counter is never ordered to increment simultaneously by both the ILC and the MLC.*

$$\text{G}\ (\,!((\text{itc}+ = 1)\ \&\ (\text{mtc}- = 1)));$$

Table 1 shows the CPU time and the memory used in building the composite machine and checking the simplified property regarding the signal *Flag* on the composite machine. The experiment was carried out on a SPARC Station 20 with 128 MB of memory.

6.1.3. *Property checking using VIS*

Besides the ASM-based verification experiments, we also verified the same set of properties using VIS [25]. The same ITC behaviour model was recoded in a subset of Verilog HDL, accepted by VIS. However, since VIS is based on finite state machines, the counters *tc* and *ic* are now assigned concrete values which indicate the maximum number of cars that are allowed in the tunnel and on the island. We developed models according to the number of register bits used for the counters. For example, if 4 bits are used to describe *ic* (tc), then the maximum of 16 cars are allowed on the island (in the tunnel). It takes 65 transition steps to compute all the reachable states when 4 bit counters are used. From Table 2, we can see that the number of transition steps increases when the counter width increases. The properties were described in CTL as follows:

TABLE 1. Statistics for the ITC property verification in MDG.

	Building the composite machine		Checking the simplified property	
Verification	CPU time (s)	Memory (MB)	CPU time (s)	Memory (MB)
Property 1	0.25	0.95	0.94	3.66
Property 2	0.32	0.98	0.61	3.53
Property 3	0.38	1.02	1.47	5.69
Property 4	0.27	1.03	0.68	4.04

PROPERTY 1. $\text{G}(!((\text{igl} = 1^*\text{mgl} = 1)));$

PROPERTY 2. $\text{G}(!((\text{ic\_minus} = 1^*\text{ic\_plus} = 1)));$

PROPERTY 3. *In CTL, this property could be expressed as the conjunction of the following formulas. We have to enumerate all the possible values that tc could take, i.e. from* 0 *to* 15.

$$\begin{gathered}\text{G}(\,((\text{tc}+ = 1)^*(\text{tc}- = 1)^*(\text{tc}<0> = 0^*\text{tc}<1> = 0^*\text{tc}<2> = 0^*\text{tc}<3> = 0))\\ \rightarrow (\text{AX}(\text{tc}<0> = 0^*\text{tc}<1> = 0^*\text{tc}<2> = 0^*\text{tc}<3> = 0)));\\ \text{G}(((\text{tc}+ = 1)^*(\text{tc}- = 1)^*(\text{tc}<0> = 1^*\text{tc}<1> = 0^*\text{tc}<2> = 0^*\text{tc}<3> = 0))\\ \rightarrow (\text{AX}(\text{tc}<0> = 1^*\text{tc}<1> = 0^*\text{tc}<2> = 0^*\text{tc}<3> = 0)));\\ \ldots\ldots\ldots..\\ \text{G}(((\text{tc}+ = 1)^*(\text{tc}- = 1)^*(\text{tc}<0> = 1^*\text{tc}<1> = 1^*\text{tc}<2> = 1^*\text{tc}<3> = 1))\\ \rightarrow (\text{AX}(\text{tc}<0> = 0^*\text{tc}<1> = 0^*\text{tc}<2> = 0^*\text{tc}<3> = 0)));\end{gathered}$$

PROPERTY 4. $\text{G}(\,!((\text{itc}+ = 1) * (\text{mtc}- = 1))\,);$

Table 2 shows the CPU time and the memory used for verifying all the four properties on models with different counter widths. We also indicate the number of transition steps needed for the state exploration and the number of reachable states for the different models. The experiment was also carried out on a SPARC Station 20 with 128 MB of memory.

6.1.4. *Discussion*

From the experimental results shown in Tables 1 and 2, we can see that the MDG-based model checking can verify a parameterized implementation having n bits, and it does so very efficiently and independently of the datapath width. That is exactly the purpose behind the development of the ASM-based model checking methods. On the other hand, using the ROBDD-based tool VIS, the number of transition steps needed for state exploration and the number of states get doubled, and the resource usage (CPU time and memory) for the property verification increases exponentially with the counter width.

6.2. Verification of properties of an abstract counter

In this section, we use the MDG-based model checker to verify both safety and liveness properties on a small

TABLE 2. Statistics for the ITC property verification using VIS.

Counter width (bits)	CPU time (s)	Memory (MB)	Number of reachable states	Number of transition steps needed for state exploration
4	4	5.67	59808	65
5	15	6.01	234400	129
6	46	6.70	927648	257
7	205 (3:25)	8.35	3.69×10^{06}	513
8	875 (14:35)	11	1.47×10^{07}	1025
9	3097 (51:37)	22	5.88×10^{07}	2049
10	12697 (211:3 8)	50	2.35×10^{08}	4097

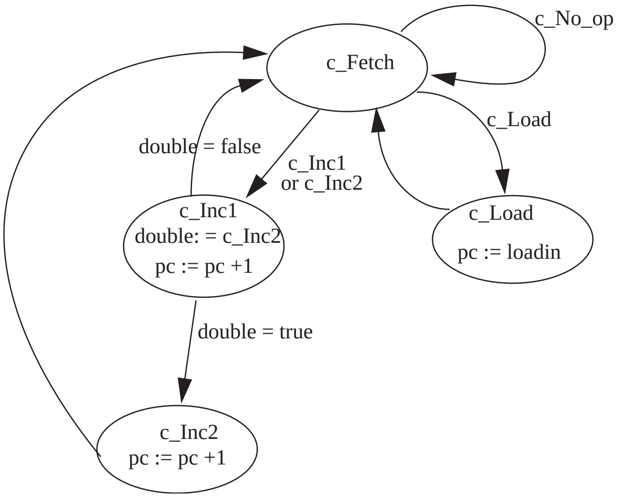

FIGURE 6. An abstract counter.

design: an abstract counter which was introduced in [12]. The abstract counter was used in [12] as an example to show how formulas in GTL can be used to describe state transitions and to specify design properties. Figure 6 shows the state transition graph of the counter. There are four control states: *c_Fetch*, *c_Load*, *c_Inc1* and *c_Inc2*. Depending on the input, the counter *pc* will get a new value, or increase by one, or keep the previous value.

6.2.1. Property checking using the MDG package

To use our model checker, we first describe the behaviour of the counter using the MDG–HDL language. The counter *pc* is of abstract sort. The control state is initialized to *c_Fetch*, the initial value of *pc* is a free variable called *init_pc* (i.e. the initial state is generalized to any value). As the counter variable *pc* is of abstract sort and implicit enumeration is applied, a set of states represented by DF $pc = init\_pc$ and its next states represented by DF $pc' = finc(init\_pc)$ are viewed as equivalent sets of states by the PbyS algorithm, all the reachable states are computed in three transition steps. The following properties were verified:

PROPERTY 1. *From* state c_Fetch, *if the input is* c_Inc2, *then the machine goes to the next* state c_Inc1. *This property*

TABLE 3. Statistics for the abstract counter verification in MDG.

	Building the composite machine		Checking the simplified property	
Verification	CPU time (s)	Memory (MB)	CPU time (s)	Memory (MB)
Property 1	0.17	0.80	0.04	0.14
Property 2	0.21	0.89	0.04	0.15
Property 3	0.31	0.90	0.12	1.75
Property 4	0.37	1.65	0.06	0.51

is expressed in L_{MDG} *as follows:*

$$\mathrm{G}(\,(\mathrm{state} = \mathrm{c\_Fetch} \,\&\, \mathrm{input} = \mathrm{c\_Inc2}) \\ \rightarrow (\mathrm{X}(\mathrm{state} = \mathrm{c\_Inc1}))\,);$$

PROPERTY 2. *From* state c_Fetch, *if the input is* c_Inc2, *then the machine always reaches* state c_Inc2 *in two transition steps. This property is expressed in* L_{MDG} *as follows:*

$$\mathrm{G}(\,(\mathrm{state} = \mathrm{c\_Fetch} \,\&\, \mathrm{input} = \mathrm{c\_Inc2}) \\ \rightarrow (\mathrm{XX}(\mathrm{state} = \mathrm{c\_Inc2}))\,);$$

PROPERTY 3. *From* state c_Fetch, *if the input is* c_Inc2, *then the machine reaches* state c_Fetch *in three transition steps and the counter pc has been increased by 2. This property is expressed in* L_{MDG} *as follows:*

$$\mathrm{G}(\,(\mathrm{state} = \mathrm{c\_Fetch} \,\&\, \mathrm{input} = \mathrm{c\_Inc2}) \\ \rightarrow (\mathrm{LET}(\mathrm{v1} = \mathrm{pc})\ \mathrm{IN}\ (\mathrm{XXX}(\mathrm{state} = \mathrm{c\_Fetch}\&\, \mathrm{pc} \\ = \mathrm{finc}(\mathrm{finc}(\mathrm{v1})))))\,);$$

PROPERTY 4. *From* state c_Fetch, *the machine will eventually reach* state c_Load *if the input is not* c_No_op *or* c_Inc1 or c_Inc2 *forever. The property is expressed in* L_{MDG} *as:*

$$\mathrm{G}(\,(\mathrm{state} = \mathrm{c\_Fetch}) \Rightarrow (\mathrm{F}(\mathrm{state} = \mathrm{c\_Load})));$$

under the following fairness constraint:

$$!(\,(\mathrm{state} = \mathrm{c\_Fetch}) \\ \rightarrow ((\mathrm{input} = \mathrm{c\_Inc1})|(\mathrm{input} = \mathrm{c\_No\_op}) \\ |(\mathrm{input} = \mathrm{c\_Inc2}))\,);$$

TABLE 4. Statistics for the abstract counter verification using VIS.

Counter width (bits)	CPU time (s)	Memory (MB)	Number of reachable states	Number of transition steps needed for state exploration
4	0.56	2.84	448	6
8	3	3.72	7168	6
16	7	4.80	1.83501×10^6	6
32	12	6.12	1.20259×10^{11}	6

These properties were verified by our model checker using less than one second. Table 3 shows the CPU time in seconds used in building the composite machine and checking the simplified property regarding *Flag* on the composite machine. The experiment was carried out on a SPARC Station 20 with 128 MB of memory.

6.2.2. *Property checking using VIS*

To compare the performance of the MDG-based model checker to that of an FSM-based verification tool, and to verify partially the verification results, we carried out the same property verification using VIS. Again, for the counter *pc*, we have to give its upper bound. We modelled the abstract counter in a subset of Verilog using registers with different width for the counter *pc*, i.e. registers consisting of 4 bits, 8 bits, 16 bits and 32 bits. On each model, we verified the same set of properties as in Section 6.2.1. The properties for the model with 4 bit *pc* register are stated in CTL as follows:

PROPERTY 1. G(((state = c_fetch) ∗ (input_instruction = c_inc2)) → (AX(state = c_inc1)));

PROPERTY 2. G(((state = c_fetch) ∗ (input_instruction = c_inc2)) → (AX(AX(state = c_inc2))));

PROPERTY 3. G(((state = c_fetch) ∗ (input_instruction = c_inc2) ∗ (pc<3> = 0 ∗ pc<2> = 0 ∗ pc<1> = 0 ∗ pc<0> = 0)) → (AX(AX(AX((state = c_fetch) ∗ (pc<3> = 0 ∗ pc<2> = 0 ∗ pc<1> = 1 ∗ pc<0> = 0)))))); with (pc<3>pc<2>pc<1>pc<0>) *ranging over from* 0000 to 1111;

PROPERTY 4. G((state = c_fetch) → (AF(state = c_load)));

under the following fairness constraint:

!((state = c_Fetch) → ((input = c_Inc1)|
(input = c_No_op)|(input = c_Inc2))).

Table 4 shows the number of transitions it takes for each model to compute all the reachable states, the number of the reachable states, the CPU time and the memory needed to verify Properties 1–4.

6.2.3. *Discussion*

The statistics shown in Tables 3 and 4 again demonstrate that the MDG-based model checking can verify both safety and liveness properties on a parameterized implementation independent of the datapath width very efficiently. However, from Table 4, we can see that with the counter width increasing, the number of reachable states increases exponentially, but the number of transition steps needed for state exploration stays the same and the usage of CPU time and memory only increases slightly, which was not the case in the ITC. The reason is that in this particular example, the counter *pc* is independent of the state transitions, i.e. the state transitions are not gated by the value of *pc*. Every time when loading in a new value of *pc* it can take any value within its range, hence, the node *pc* will not appear in the BDD expression of the sets of states. Therefore, no matter how large the width of *pc* is, the time and memory usage will not grow significantly. Nevertheless, the MDG-based model checking still outperforms the ROBDD-based model checker in the sense that one ASM model of the Abstract Counter and one set of properties automatically cover all the possible *pc* widths. Using VIS on the other hand, we have to build separate models and to develop separate sets of properties for *pc* instances of different widths.

6.3. Implementation issues

To check properties expressed in L_{MDG} automatically, we developed programs that

(i) check if the signals in a property (except the ordinary variables) are declared in the original circuit description; report any errors;
(ii) check the syntax of the property; report any errors;
(iii) build additional circuits to represent the Next_let_formulas in the property and the exception conditions if fairness constraints are imposed;
(iv) merge the description of the additional circuits with the description of the original circuit, which means adding declarations of components and signals of the additional circuits to the original circuit description file and the variable order file.

The above programs were implemented in C with Yacc & Lex. The model checking algorithms were developed upon the existing MDGs package implemented in Quintus Prolog V3.2.

7. CONCLUDING REMARKS

We studied model checking for a first-order linear-time temporal logic based on the ASMs. Since a data value is represented by a single variable of abstract type, rather by a vector of Boolean variables, and a data operation is represented by an uninterpreted function symbol, the width of a datapath of a design has no effect on the description model of the design. We can then alleviate the state explosion problem in symbolic model checking caused by a large datapath.

We defined L_{MDG} as the property specification language and developed property checking algorithms for L_{MDG}. Using L_{MDG}, both safety and liveness properties can be expressed with or without fairness constraints. To check a property of L_{MDG} on an ASM M, we first build additional ASMs for all the Next_let_formulas (which contain the temporal operator X) that appear in the property. We then compose the additional ASMs with M, and finally verify a simpler property on the composite machine. We use MDGs to encode sets of states and the transition relations. The property checking procedures are based on implicit state enumeration and are carried out fully automatically. We illustrated the application of our model checker on the ITC and the Abstract Counter benchmarks. The experimental results demonstrate that the MDG-based model checking can verify both safety and liveness properties on parameterized implementations independent of the datapath width very efficiently. Due to space limit, the proof of the correctness of the property checking procedures are not presented in this paper, while it can be found in [20].

Since we use first-order logic, the reachability analysis may not terminate [26], thus the property checking may not terminate either. We are currently exploring techniques that can mitigate this problem [27, 28]. We are also applying the MDG-based model checker to verify some industrial scale designs with large data path (most telecommunication circuits happen to fall into this category).

REFERENCES

[1] Burch, J. R., Clarke, E. M. and Long, D. E. (1990) Symbolic model checking with partitioned transition relations. In *Proc. Int. Conf. on Very Large Scale Integration (VLSI 91)*, Edinburgh, Scotland, August 1990, pp. 49–58. IFIP Transactions, North-Holland.

[2] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L. and Hwang, L. J. (1990) Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Annual IEEE Symp. on Logic in Computer Science*, Philadelphia, PA, June 1990, pp. 428–439. IEEE Computer Society Press.

[3] Clarke, E. M., Grumberg, O. and Long, D. E. (1992) Model checking and abstraction. In *Proc. Nineteenth Annual ACM Symp. on Principles of Programming Languages (POPL 92)*, New York, January 1992, pp. 343–354. ACM Press.

[4] McMillan, K. L. (1993) *Symbolic Model Checking, An Approach to the State Explosion Problem*. Kluwer Academic Publishers.

[5] Corella, F., Langevin, M., Cerny, E., Zhou, Z. and Song, X. (1995) State enumeration with abstract descriptions of state machines. In *Proc. IFIP WG10.5 Advanced Research Working Conf. on Correct Hardware Design and Verification Methods (Charme'95)*, Frankfurt, Germany, October 1995. Vol. 987 of Lecture Notes in Computer Science, pp. 146–160. Springer-Verlag.

[6] Corella, F., Zhou, Z., Song, X., Langevin, M. and Cerny, E. (1997) Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, **10**(1), pp. 7–46.

[7] Zhou, Z., Song, X., Corella, F., Cerny, E. and Langevin, M. (1995) Description and verification of RTL designs using multiway decision graphs. In *Proc. Conf. on Computer Hardware Description Languages and their Applications (CHDL'95)*, Chiba, Japan, August 1995, pp. 575–580. Elsevier Science Publishers.

[8] Bryant, R. E. (1986) Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Comp.*, **35**(8), 677–691.

[9] Zhou, Z., Song, X., Tahar, S., Cerny, E., Corella, F. and Langevin, M. (1996) Formal verification of the island tunnel controller using multiway decision graphs. In *Proc. Formal Methods in Computer-Aided Design (FMCAD'96)*, Palo Alto, California, USA, November 1996, volume 1166 of Lecture Notes in Computer Science, pp. 233–246. Springer-Verlag.

[10] Hungar, H., Grumberg, O. and Damm, W. (1995) What if model checking must be truly symbolic. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, Aarhus, Denmark, May 1995, volume 1019 of Lecture Notes in Computer Science, pp. 230–244. Springer-Verlag.

[11] Bohn, J., Damm, W., Grumberg, O., Hungar, H. and Laster, K. (1998) First-order-CTL model checking. In *Proc. Foundations of Software Technology and Theoretical Computer Science (FSTTCS'98)*, Chennai, India, December 17–19, 1998, volume 1530 of Lecture Notes in Computer Science, pp. 283–294. Springer-Verlag.

[12] Cyrluk, D. and Narendran, P. (1994) Ground temporal logic: a logic for hardware verification. In *Proc. Computer-Aided Verification (CAV'94)*, Stanford, CA, June 1994, volume 818 of Lecture Notes in Computer Science, pp. 247–259. Springer-Verlag.

[13] Hojati, R. and Brayton, R. K. (1995) Automatic datapath abstraction in hardware systems. In *Proc. 7th International Conf. On Computer Aided Verification (CAV'95)*, Liege, Belgium, July 1995, volume 939 of Lecture Notes in Computer Science, pp. 98–113. Springer-Verlag.

[14] Hojati, R., Dill, D. L. and Brayton, R. K. (1997) Verifying linear temporal properties of data insensitive controllers using finite instantiations. In *Proc. IFIP Conf. on Hardware Description Languages and their Applications (CHDL'97)*, Toledo, Spain, April 1997, pp. 60–73. Chapman & Hall.

[15] Isles, A. J., Hojati, R. and Brayton, R. K. (1998) Computing reachable control states of systems modeled with uninterpreted functions and infinite memory. In *Proc. Conf. on Computer-Aided Verification (CAV'98)*, Vancouver, Canada, July 1998, volume 1427 of Lecture Notes in Computer Science, pp. 256–267. Springer-Verlag.

[16] Burch, J. R. and Dill, D. L. (1994) Automatic verification of pipelined microprocessor control. In *Proc. 6th Int. Conf. on Computer Aided Verification (CAV'94)*, Stanford, CA, June 1994, volume 818 of Lecture Notes in Computer Science, pp. 68–80. Springer-Verlag.

[17] Velev, M. N. and Bryant, R. E. (2001) EVC: A Validity Checker for the Logic of Equality with Uninterpreted Functions and Memories, Exploiting Positive Equality and Conservative

Transformations, Computer-Aided Verification (CAV'01), July 2001, volume 2102 of Lecture Notes in Computer Science, pp. 235–240. Springer-Verlag.

[18] Namjoshi, K. S. and Kurshan, R. P. (2000) Syntactic program transformations for automatic abstraction. In *Proc. 12th Int. Conf. On Computer Aided Verification (CAV'2000)*, Chicago, USA, July 2000, volume 1855 of Lecture Notes in Computer Science, pp. 435–449. Springer-Verlag.

[19] Xu, Y., Cerny, E., Song, X., Corella, F. and Ait Mohamed, O. (1998) Model checking for a first-order temporal logic using multiway decision graphs. In *Proc. Conf. on Computer-Aided Verification (CAV'98)*, Vancouver, Canada, July 1998, volume 1427 of Lecture Notes in Computer Science, pp. 219–231. Springer-Verlag.

[20] Xu, Y. (1999) *Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs*. Ph.D. Thesis, D'IRO, University of Montreal, 1999. http://www.ece.pdx.edu/∼song/research/areas.html.

[21] Emerson, E. A. and Lei, C. L. (1987) Modalities for model checking: branching time logic strikes back. *Sci. Comp. Programm.*, **8**, 275–306.

[22] Kurshan, R. P. (1987) Reducibility in Analysis of Coordination. *Lecture Notes in Computer Science*, **103**, 19–39. Springer-Verlag.

[23] Kurshan, R. P. (1994) *Automata-Theoretic Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, NJ, USA.

[24] Fisler, K. and Johnson, S. (1995) Integrating design and verification environments through a logic supporting hardware diagrams. In *Proc. IFIP Conf. on Hardware Description Languages and their Applications (CHDL'95)*, Chiba, Japan, August 1995, pp. 669–674. Elsevier Science Publishers.

[25] Brayton, R. K. *et al*. (1995) *VIS: A System for Verification and Synthesis*. Technical Report UCB/ERL M95, Electronics Research Lab, University of California, Berkeley, CA.

[26] Clarke, E. M. and Emerson, E. A. (1981) Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logics of Programs*, New York, 1981, volume 131 of Lecture Notes in Computer Science, pp. 52–71. Springer-Verlag.

[27] Ait Mohamed, O., Song, X. and Cerny, E. (2003) On the Non-termination of MDG-based abstract state enumeration. *Theor. Comp. Sci.*, **300**, 161–179.

[28] Ait Mohamed, O., Song, X. and Cerny, E. (2004) MDG-based state enumeration by retiming and circuit transformation. *J. Circuits Syst. Comp.*, to be published. World Scientific Publishers.