# Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs

Y. Xu[1], E. Cerny[1], X. Song[1], F. Corella[2], O. Aït Mohamed[1]

[1] D'IRO, Université de Montréal,
C.P. 6128, Succ. Centre-Ville, Montréal, H3C 3J7, Canada
[2] Hewlett-Packard Company, USA

**Abstract.** We study model checking for a first-order linear-time temporal logic. The computation model is based on *Abstract State Machines* (ASMs) in which data and data operations are described using abstract sorts and uninterpreted function symbols. ASMs are suitable for describing Register-Transfer level designs. We then define a first-order linear-time temporal logic called $\mathcal{L}_{\mathcal{MDG}}$ which supports the abstract data representations. Both safety and liveness properties can be expressed in $\mathcal{L}_{\mathcal{MDG}}$, 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.

## 1 Introduction

Symbolic model checking has proven to be a very practical technique for the automatic verification of hardware designs [8, 9, 12]. However, these methods require the description of the design to be at the Boolean logic level, and thus they are in general 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 for a first-order linear-time temporal logic. Our approach is based on a computation model called an *abstract state machine* (ASM) where a data value can be 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 [5]. ASMs can be used to describe designs at the Register Transfer Level (RTL). We first define the first-order linear-time temporal logic $\mathcal{L}_{\mathcal{MDG}}$ and then develop the corresponding property checking algorithms.

To check a property $p$ in $\mathcal{L}_{\mathcal{MDG}}$ on an ASM $M$, we first build additional ASMs $M_j$ automatically for basic subformulas of $p$ in which only the temporal operator X is allowed (called *Next_let_formulas*), then we compose the additional ASMs with $M$, and finally we check a simplified property on the composite machine. The property checking algorithms are based on implicit state enumeration as supported by Multiway Decision Graphs (MDGs) [5] whose complexity is independent of the width of the datapath. However the algorithms do not always terminate[1]. Decidability of model checking for

[1] Hence, strictly speaking, they should be called procedures rather than algorithms.

$\mathcal{L}_{\mathcal{MDG}}$, just like decidability of reachability analysis for our ASMs, is left as an open question.

While our formalization of ASMs was introduced in [4], this is the first time that we address the model checking problem for ASMs.

To our knowledge, three previous developments reported in the literature are related to ours. Hungar, Grumberg and Damm [6] proposed a "true symbolic model checking" technique. They represented data and data operations by first-order formulas, and used FO-ACTL (first-order ACTL), a branching-time first-order temporal logic with universal path quantifier to specify properties. The method is based on the assumption that all data loops terminate, and on the separation of control and data path in typical circuits. If a property only contains control signals, then Boolean model checking is applied. When a property contains data, they replace all first-order components in the property formula with the Boolean constant true resulting in a propositional CTL formula. If this propositional CTL formula is not verified by a Boolean model checker, then the original property fails. Otherwise (the propositional CTL formula is verified), the tableau method is used to generate a pure first-order verification condition from the system and the property to be proven. They then verify the property by proving the validation of the verification condition using a theorem prover.

Cyrluk and Narendran defined a first-order temporal logic - Ground Temporal Logic (GTL) [1], which falls in between first-order and propositional temporal logics. The validity problem in GTL is the same as checking a linear time temporal logic formula for all computation paths. In [1], the authors showed that the full GTL is undecidable. They then identified a decidable fragment of GTL, consisting of $\blacksquare p$ (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 for this decidable fragment.

Hojati, Brayton et al. [13, 14] proposed an integer combinational/sequential (ICS) concurrency model which uses finite relations, interpreted and uninterpreted integer functions and predicates, and interpreted memory functions to describe hardware systems with datapath abstraction. 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 instantiations) and then the property verification can be carried out at the Boolean level without sacrificing accuracy. They gave a linear time algorithm for recognizing such a subset. For verifying properties of circuits containing data transformations modeled using interpreted and uninterpreted functions, finite instantiation cannot be used. In that case, they compute the set of states reachable in $n$ steps using BDDs, and check that no error exists in these $n$ steps.

Burch and Dill also used a subset of first-order logic, specifically, the quantifier-free logic of equality with uninterpreted functions for verifying microprocessor control circuity [7]. Their logic is appropriate for verification of microprocessor control because it allows abstraction of datapath values and operations. However, their method, unlike ours, cannot verify properties involving temporal operators, especially, liveness properties.

Compared to [1], we shall see in the following sections that the decidable fragment of GTL is actually a subset of $\mathcal{L}_{\mathcal{MDG}}$. Compared to ICS [13, 14], our ASM models are more general in the sense that the abstract sort variables in our system (corresponding to the integer variables in ICS models) can be assigned any value in their domain, rather a particular constant or function of constants as in the ICS model. For the class of ICS models where finite instantiations cannot be used, our verification system can

still compute all the reachable states and check safety properties as well as certain liveness properties. For example, the abstract counter presented in Section 6 cannot be handled by the ICS model, but it can be described using the ASM model. Compared to [6], our first-order linear-time temporal logic is less expressive than FO-ACTL, since we only allow limited nesting of temporal operators. However, in our approach the property is checked on the whole model automatically, while in [6] a theorem prover is eventually needed to validate the pure first-order verification condition.

This paper is organized as follows: In Sections 2, we give a definition of abstract state machines (ASMs). In Section 3, we define the syntax and the semantics of $\mathcal{L}_{\mathcal{MDG}}$. In Sections 4 and 5, we present the property checking algorithms, and show how to impose fairness constraints. In Section 6, we discuss implementation issues and present experimental results: property verification of an abstract counter. We conclude the paper in Section 7.

# 2 Abstract State Machines (ASMs)

Abstract State Machines (ASMs) are used in our approach to model the designs to be verified. We strongly recommend that interested readers refer to [5] for the definitions of a many-sorted first-order logic and Directed Formulas (DFs) before reading the definition of the ASMs. We cannot include them in this paper due to lack of space.

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 $\eta$ be an 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 $\psi$, an input vector is a $\psi$-compatible assignment to the set of input variables $X$; thus the set of input vectors (input alphabet) is $\Phi_X^\psi$. Similarly, $\Phi_Z^\psi$ is the set of output vectors. A state is a $\psi$-compatible assignment to the set of state variables $Y$, hence, the state space is $\Phi_Y^\psi$. A state $\phi$ can also be described by an assignment $\phi' = \phi \circ \eta^{-1} \in \Phi_{Y'}^\psi$, to $Y'$.
2. $F_I$ is a 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 $\psi$, a state $\phi \in \Phi_Y^\psi$ is an initial state iff $\psi, \phi \models (\exists U) F_I$. Thus the set of initial states is $S_I = Set^\psi(F_I) = \{\phi \in \Phi_Y^\psi \mid \psi, \phi \models (\exists U) F_I\}$.
3. $F_T$ is a DF of type $(X \cup Y) \rightarrow Y'$ representing the transition relation. Given an interpretation $\psi$, an input vector $\phi \in \Phi_X^\psi$ and a state $\phi' \in \Phi_Y^\psi$, a state $\phi'' \in \Phi_Y^\psi$ is a possible next state iff $\psi, \phi \cup \phi' \cup (\phi'' \circ \eta^{-1}) \models F_T$. Thus the transition relation is $R_T = \{(\phi, \phi', \phi'') \in \Phi_X^\psi \times \Phi_Y^\psi \times \Phi_Y^\psi \mid \psi, \phi \cup \phi' \cup (\phi'' \circ \eta^{-1}) \models F_T\}$.
4. $F_O$ is a DF of type $(X \cup Y) \rightarrow Z$ representing the output relation. Given an interpretation $\psi$, the output relation is $R_O = \{(\phi, \phi', \phi'') \in \Phi_X^\psi \times \Phi_Y^\psi \times \Phi_Z^\psi \mid \psi, \phi \cup \phi' \cup \phi'' \models F_O\}$.

# 3 A First-Order Linear-Time Temporal Logic: $\mathcal{L}_{\mathcal{MDG}}$

Given a description of an ASM, and a set of ordinary variables, the *atomic formulas* of $\mathcal{L}_{\mathcal{MDG}}$ 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:

1. Each atomic formula is a *Next_let_formula*.
2. If $p, q$ are *Next_let_formulas*, then so are:
   $!p$ $(not\ p), p\&q$ $(p\ and\ q), p \mid q$ $(p\ or\ q), p \rightarrow q$ $(p\ implies\ q), Xp$, and
   $\mathsf{LET}\,(v = t)\ \mathsf{IN}\,p$, where $t$ is an *ASM_variable* and $v$ an ordinary variable[2].

The properties allowed in $\mathcal{L}_{\mathcal{MDG}}$ can have the following forms:

$$\begin{aligned} Property ::= &\ Next_let_formula \\ &\mid \mathsf{G}(Next_let_formula) \\ &\mid \mathsf{F}(Next_let_formula) \\ &\mid Next_let_formula \mathsf{U} Next_let_formula \\ &\mid \mathsf{G}(Next_let_formula \rightarrow \mathsf{F} Next_let_formula) \\ &\mid \mathsf{G}(Next_let_formula \rightarrow (Next_let_formula \mathsf{U} Next_let_formula)) \end{aligned}$$

### Semantics of $\mathcal{L}_{\mathcal{MDG}}$

A path $\pi$ is a sequence of states. We use $\pi^i$ to denote a path starting from $\pi_i$ where $\pi_i$ denote the $i^{th}$ state in $\pi$. All the formulas in $\mathcal{L}_{\mathcal{MDG}}$ are path formulas. We write $\pi, \sigma \models p$ to mean that a path formula $p$ is true at path $\pi$ under a $\psi$-compatible assignment $\sigma$ to the ordinary variables. We use $Val_{s \cup \sigma}(t)$ to denote the value of term $t$ under a $\psi$-compatible assignment $s$ to state variables, input variables, and output variables and a $\psi$-compatible assignment $\sigma$ to the ordinary variables. We define $\models$ inductively as follows:

$\pi, \sigma \models t_1 = t_2$ iff $Val_{\pi_0 \cup \sigma}(t_1) = Val_{\pi_0 \cup \sigma}(t_2)$.
$\pi, \sigma \models \mathsf{LET}\ (v = t) \mathsf{IN}\ p$ iff $\pi, \sigma' \models p$ where $\sigma' = \sigma \setminus \{(v, \sigma(v))\} \cup \{(v, Val_{\pi_0 \cup \sigma}(t))\}$
$\pi, \sigma \models\ !p$ iff it is not the case that $\pi, \sigma \models p$.
$\pi, \sigma \models p\&q$ iff $\pi, \sigma \models p$ and $\pi, \sigma \models q$.
$\pi, \sigma \models p \mid q$ iff $\pi, \sigma \models p$ or $\pi, \sigma \models q$.
$\pi, \sigma \models p \rightarrow q$ iff $\pi, \sigma \models !p$ or $\pi, \sigma \models q$.
$\pi, \sigma \models \mathsf{X}p$ iff $\pi^1, \sigma \models p$.
$\pi, \sigma \models \mathsf{G}p$ iff $\pi^j, \sigma \models p$ for all $j \geq 0$.
$\pi, \sigma \models \mathsf{F}p$ iff $\pi^j, \sigma \models p$ for some $j \geq 0$.
$\pi, \sigma \models p\mathsf{U}q$ iff for some $k \geq 0, \pi^k, \sigma \models q$, and $\pi^j, \sigma \models p$ for all $j (0 \leq j < k)$.

Given a property in $\mathcal{L}_{\mathcal{MDG}}$ regarding an ASM under a given interpretation $\psi$, the property holds on the ASM iff the property is true for every path $\pi$ such that $\pi_0$ is an initial state and, for every $i$, there is a transition from $\pi_i$ to $\pi_{i+1}$ for some $\psi$-compatible assignment to the input variables.

## 4 Model Checking for Properties in $\mathcal{L}_{\mathcal{MDG}}$

Our approach to property checking consists of constructing additional ASMs that represent the *Next_let_formulas* appearing in the property, composing these additional

[2] We allow the formula $\mathsf{LET}\,(v_1 = t_1)\&\ldots\&(v_n = t_n)\ \mathsf{IN}\,p$ as a shorthand for $\mathsf{LET}\,(v_1 = t_1)\ \mathsf{IN}(\ \mathsf{LET}\,(v_1 = t_1)\ \mathsf{IN}(\ldots\ \mathsf{LET}\,(v_n = t_n)\ \mathsf{IN}\,p\ )\ldots)$; and we call $(v_1 = t_1)\&\ldots\&(v_n = t_n)$ a Let_equation.

ASMs with the original one, and then applying the appropriate algorithms to verify a simplified property on the composite machine. Given a *Next_let_formula* $P$ regarding an ASM $D = (X, Y, Z, F_I, F_T, F_O)$, an ASM $D_P = (X_P, Y_P, Z_P, F_{IP}, F_{TP}, F_{OP})$ can be constructed to represent the *Next_let_formula*. The input variables of $D_P$ are the *ASM_variables* of $D$ which are referred to in the property, i.e., $X_P \subseteq X \cup Y \cup Z$. They represent the values at the "current" clock cycle. The set of the state variables $Y_P$ and the transition relation $F_{TP}$ are constructed so as to "remember" the values of the input variables of $D_P$ or the results of comparisons of variables in the past $n$ (or less than $n$) cycles, where $n$ is the maximum nesting number of the X operators in the property. The set of state variables of $D_P$ contains a special state variable *Flag* of boolean sort which indicates the truth of the *Next_let_formula* one cycle earlier. There is no output from $D_P$, i.e., $Z_P$ is empty, hence there is no output relation either. The details of an algorithm for constructing an ASM representing a *Next_let_formula* are given in [16].

In the following subsections, we describe algorithms for verifying the various forms of the formulas in $\mathcal{L}_{\mathcal{MDG}}$. 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, we can obtain 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.

Recall that Disj computes disjunction and RelP computes relational product; both can be applied to any number of MDGs at once. Recall that PbyS($P$, $Q$) removes from the MDG $P$ any MDG paths (i.e. disjuncts) that are subsumed by the MDG $Q$.

### 4.1 Verification of G(Next_let_formula)

In this case we perform reachability analysis on the composite machine $M = (X_M, Y_M, Z_M, G_I, G_T, G_O)$, where:

- $X_M = X$ is the set of the input variables,
- $Y_M = Y \cup Y_P$ is the set of state variables, containing the variables in $Y$ and in $Y_P$,
- $Z_M = Z$ is the set of output variables,
- $G_I$ is a DF representing the set of initial states of $M$,
- $G_T$ is a DF representing the transition relation of $M$,
- $G_O$ is a DF representing the output relation of $M$.
  and in each state we check that $Flag = 1$.

The algorithm to verify G($Flag = 1$) is as follows:

1. Check_G($M, C$)
   /* $C$ is the DF $Flag = 1$ */
2. $R := G_I; S := G_I; K := 0;$
3. loop
4. $S_{notC}$ := PbyS($S, C$);
5. if $S_{notC} \neq$ F then return failure;
   /* if the property is not satisfied in $Set(S)$, then report failure */
6. $K := K + 1;$
7. $I$ := Fresh($X_M, K$);/*generate input values */

```
8.   N := RelP({I, S, G_T}, X_M ∪ Y_M, η');/* compute next states*/
9.   S := PbyS(N, R); /*compute frontier set of states*/
10.  if S = F then return success; /* if fixpoint reached, report success*/
11.  R := PbyS(R, S); /* simplify R by removing states subsumed by S */
12.  R := Disj(R, S); /* compute all states reached so far */
13. end loop;
14. end Check_G;
```

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 both *Next_let_formulas* and then transform the problem to the verification of $(FlagP = 1)\mathsf{U}(FlagQ = 1)$ for all the initial states of the composite machine.

```
1.    check_U(M, C_p, C_q)
      /*M is the composite machine, G_I is the set of initial states
      /* G_T is the transition relation */
      /* C_p is the DF of FlagP = 1. C_q is the DF of FlagQ = 1 */
2.    Σ := ∅; /* Σ is a set of DFs, each DF represents a set of visited states */
3.    S := G_I;
4.    K := 0;
5.    loop
6.       S_notq := PbyS(S, C_q); /*remove from S states with FlagQ = 1*/
7.       if S_notq = F then return success;
8.       if ∃T ∈ Σ, PbyS(T, S_notq) = F then return failure;
         /*This step verifies if DF S_notq covers any DF in Σ, i.e.,
         for each DF T in Σ, PbyS(T, S_notq) = F is checked to
         detect a cycle. If there is a cycle, then failure is reported*/
9.       R = PbyS(S_notq, C_p);/* remove from S_notq states with FlagP = 1*/
10.      if R ≠ F then return failure;
11.      Σ := Σ ∪ {S_notq}; /* add DF S_notq as an element to Σ */
12.      K := K + 1;
.13.     I := Fresh(X_M, K); /* generate input values */
14.      S := RelP({I, S_notq, G_T}, X_M ∪ Y_M, η');/* compute next states */
15.   end loop;
16.   end Check_U
```

The algorithm removes from the set of reached states those states satisfying $FlagQ = 1$. If the leftover $Set(S_{notq})$ becomes empty, then the algorithm stops by reporting success. Otherwise, if there is at least one cycle along which all states satisfy $FlagP = 1$,

then there is at least one path starting from the initial state where $p\mathsf{U}q$ does not hold, the algorithm stops and reports failure. Otherwise, it checks whether all the states in $Set(S_{notq})$ satisfy $FlagP = 1$. If there are some states where $FlagP = 1$ does not hold, meaning that there is a path on which $FlagP = 1$ does not hold in every state before a state satisfying $FlagQ = 1$ is reached, then the algorithm also stops and reports failure. Otherwise, it computes the next states reachable from $Set(S_{notq})$ and repeats the process.

To verify $\mathsf{G}(c \rightarrow p\mathsf{U}q)$ on machine $D$, we build a composite machine $M$ from $D$, an ASM representing $c$, an ASM representing $p$, and an ASM representing $q$, and then verify $\mathsf{G}((FlagC = 1) \rightarrow ((FlagP = 1)\mathsf{U}(FlagQ = 1)))$ on the composite machine. This is achieved by first computing all the reachable states of $M$ (represented by $W$), by collecting from $W$ those states that satisfy "$FlagC = 1$" ($V := \mathsf{Conj}(W, C_c)$ where $C_c$ is the DF of $FlagC = 1$), and finally by applying the Check_U algorithm with the set $V$ as the initial set of states.

A property in the form of $\mathsf{F}(Next_let_formula)$ can be verified by checking $\mathsf{TU}Next_let_formula$ 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 computation path is a path along which the states satisfy each fairness condition infinitely often.

In the literature, various methods for specifying fairness constraints have been developed for CTL model checking [2] and language containment using L-automata [15].

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 $H_i, i = 1, \ldots, n$, be $n$ "exception" conditions, and $S_\omega$ the set of infinitely repeating states along a computation path. If at least one $H_i$ holds on all $s$ in $S_\omega$, then the path is not fair and need not satisfy the property under investigation. That is, only those computation paths along which the states satisfy each $!H_i$ infinitely often are considered. We call the formula representing the exception condition $H_i$ an *H_formula*. The syntax of an *H_formula* is as follows:

1. The equation $t_1 = t_2$ is an *H_formula*, where $t_1$ is an *ASM_variable* and $t_2$ is an *ASM_variable* or a constant.
2. If $p, q$ are *H_formulas*, then so are $!p, p\&q, p|q, p \rightarrow q, \mathsf{X}p$.

### 5.2 Verification of $p\mathsf{U}q$ with fairness constraints

To verify that $p\mathsf{U}q$ (where $p$ and $q$ are *Next_let_formulas*) holds for the initial states of an ASM $D$ under fairness constraints $!H_1, !H_2, \ldots, !H_n$, we build additional ASMs to represent $p, q$, and $H_i$ $(1 \le i \le n)$, and then transform the problem to checking $(FlagP = 1)\mathsf{U}(FlagQ = 1)$ on the initial states of the composite machine derived from $D$ and the additional ASMs with fairness constraints $!(FlagH_i = 1)$ $(1 \le i \le n)$.

Let (1) $M$ be the composite machine, (2) $G_I$ be the set of initial states, and $G_T$ the transition relation, (3) $C_p$ be the DF of $FlagP = 1$, and $C_q$ be the DF of $FlagQ = 1$,

(4) $H_i$ $(1 \leq i \leq n)$ be the DF of $FlagH_i = 1$. The algorithm for verifying $(FlagP = 1)\mathsf{U}(FlagQ = 1)$ under fairness constraints $!(FlagH_i = 1)$ $(1 \leq i \leq n)$ is as follows:

1. Check_U_fair$(M, C_p, C_q, H_1, \ldots, H_n)$
2. $\Sigma := \emptyset$;
3. $S := G_I; K := 0$;
4. loop1
5. $S_{notq} := \mathsf{PbyS}(S, C_q)$;
6. if $S_{notq} = \mathsf{F}$ then return success;
7. if $\exists T \in \Sigma, \mathsf{PbyS}(T, S_{notq}) = \mathsf{F}$ then return failure;
8. $R = \mathsf{PbyS}(S_{notq}, C_p)$;
9. if $R \neq \mathsf{F}$ return failure;
10. $\Sigma := \Sigma \cup \{S_{notq}\}$;
11. $S_1 := S_{notq}$;
12. for $i = 1$ to $n$ do
13. $S_{notH} := \mathsf{PbyS}(S_1, H_i)$;/*remove from $S_1$ the states with $FlagH_i = 1$ */
14. $S_2 := \mathsf{Conj}(S_1, H_i)$; /*$S_2$ represents the states in $S_1$ with $FlagH_i = 1$ */
15. if $S_2 = \mathsf{F}$ then $S_{4notq} = \mathsf{F}$;
16. if $S_2 \neq \mathsf{F}$ then begin
17. $S_3 := S_2; S_f := S_2; L := 0$;
18. loop2 /* to compute all the states reachable from $S_2$ with $FlagH_i = 1$ */
19. $L := L + 1$;
20. $I_2 := \mathsf{Fresh}(X_M, L)$; /* generate new input values */
21. $N_1 := \mathsf{RelP}(\{I_2, S_f, G_T\}, X_M \cup Y_M, \eta')$; /* compute next states */
22. $N_2 := \mathsf{PbyS}(N_1, C_q)$; /*remove from $N_1$ the states with $FlagQ = 1$ */
23. $N_3 := \mathsf{Conj}(N_2, H_i)$; /*pick from $N_2$ the states with $FlagH_i = 1$ */
24. if $\mathsf{PbyS}(N_3, C_p) \neq \mathsf{F}$ then return failure;
    /* if the states in $N_3$ do not satisfy $FlagP = 1$, report failure */
25. $S_f := \mathsf{PbyS}(N_3, S_3)$; /* compute the frontier states */
26. if $S_f = \mathsf{F}$ then exit loop2;
    /* if all the states reachable from $S_2$ have been visited, exit loop 2 */
27. $S_3 := \mathsf{PbyS}(S_3, S_f)$;
28. $S_3 := \mathsf{Disj}(S_3, S_f)$;/* add the states of $S_f$ to $S_3$ */
29. end loop2;
30. $S_{41} := \mathsf{RelP}(\{I_2, S_3, G_T\}, X_M \cup Y_M, \eta')$;
    /* compute the next states of $S_3$ */
31. $S_4 := \mathsf{PbyS}(S_{41}, H_i)$; /* remove from $S_{41}$ the states with $FlagH_i = 1$ */
32. $S_{4notq} := \mathsf{PbyS}(S_4, C_q)$;
33. if $\mathsf{PbyS}(S_{4notq}, C_p) \neq \mathsf{F}$ then return failure;
34. end_if
35. $S_1 = \mathsf{Disj}(S_{4notq}, S_{notH})$;
36. end_for
37. if $S_1 \neq \mathsf{F}$ then begin
38. $K := K + 1$;
39. $I_1 := \mathsf{Fresh}(X_M, K)$;/* generate input values */
40. $S := \mathsf{RelP}(\{I_1, S_1, G_T\}, X_M \cup Y_M, \eta')$; /* compute the next states of S1 */
41. end_if
42. end loop1
43. end

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

In loop1, in steps (5) – (10), $S_{notq}$ represents the set of states in $S$ not satisfying $FlagQ = 1$. If $S_{notq}$ is empty, then the computation stops by reporting success. Otherwise, if $S_{notq}$ covers any set in $S$, 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 $\Sigma$ and the computation continues (lines $8 - 9 - 10$).

Lines (11) to (36) form a loop that 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 exception 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 $18 - 29$). This set represents all the states that are reachable from $S_2$ by any number of transition steps and each satisfies $FlagH_i = 1$ and $FlagP = 1$, but does 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, and it may not terminate). 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 33). $S_1$ is the union of the set of states represented by $S_{4notq}$ and $S_{notH}$ at each iteration of the loop.

If $S_1$ is not empty, then $S$ is computed to represent the states reachable in one transition step from the states in $S_1$. The computation continues in loop1 with $S$ as the new frontier set of states to be checked.

In Fig. 1, we show an example that illustrates how this algorithm works. Suppose we wish to verify $(FlagP = 1)\mathsf{U}(FlagQ = 1)$ under the fairness constraint $!(FlagH_1 = 1)$ on the state transition graph given in Fig. 1. 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 $3^{rd}$ iteration in loop1. However, checking $(FlagP = 1)\mathsf{U}(FlagQ = 1)$ without the fairness constraint would fail on the path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_2 \rightarrow s_3 \rightarrow s_2 \rightarrow s_3 \ldots$.

To check $\mathsf{G}(c \rightarrow p\mathsf{U}q)$ 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 transform the problem to checking $\mathsf{G}((FlagC = 1) \rightarrow ((FlagP = 1)\mathsf{U}(FlagQ = 1)))$ 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 "$FlagC = 1$" ($V := \mathsf{Conj}(W, C_c)$ 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.

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 path, if the states repeating forever are covered by a cycle set and there is no other state reached from those states as shown in Fig. 2, Check_U_fair will report failure. However, it is not necessary that $FlagP = 1$ holds on those states, since this path should not be considered. Thus Check_U_fair may give a false negative answer. In real system, this situation happens rarely, however.

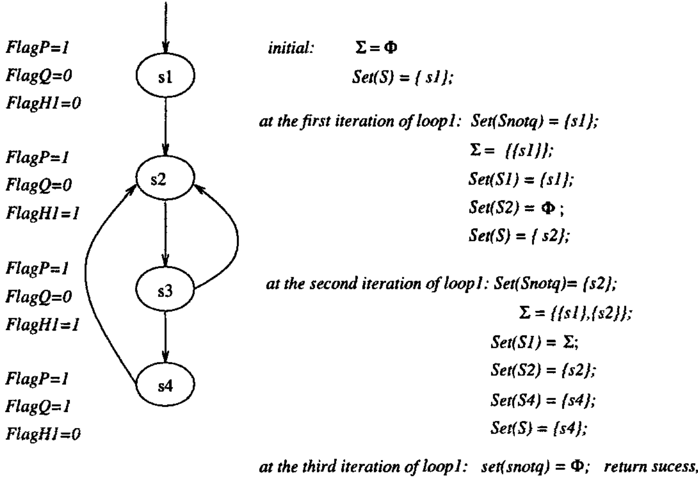


**Fig. 1.** An example of verifying $(FlagP = 1)\mathsf{U}(FlagQ = 1)$ under the fairness constraint $!(FlagH_1 = 1)$

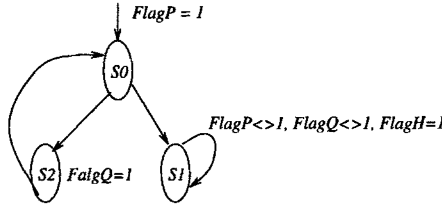


**Fig. 2.** Example of a false negative answer for verifying $(FlagP = 1)\mathsf{U}(FlagQ = 1)$ under the fairness constraint $!(FlagH = 1)$

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

## 6 Implementation Issues and Experimental Results

To automatically verify properties expressed in $\mathcal{L}_{\mathcal{MDG}}$, we developed programs to:

- check if the signals in a property are declared in the original circuit description;
- check the syntax of the property;
- build the additional circuits to represent the *Next_let_formulas* in the property and the exception conditions if fairness constraints are imposed;
- merge the description of the additional circuits with the description of the circuit to be verified;

The above programs are implemented in C with Yacc and Lex. The property checking algorithms are developed based on the current MDG package and they are implemented in Quintus Prolog V3.2.

To show how to express properties in our logic, and how to use our model checker, we use the abstract counter of [1] as an example. Fig. 3 shows the control state transition graph of the counter. There are four control states: $c_Fetch, c_Load, c_Inc1, andc_Inc2$. Depending on the input, the counter $pc$ will get a new value, or increase by one, or keep the previous value.

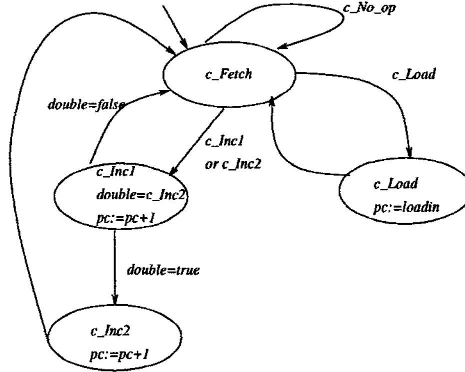


**Fig. 3.** An abstract counter

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). It takes three transition steps to compute all the reachable states. We verified the following three properties:

**Property 1:** From state $c_Fetch$, if the input is $c_Inc2$, then the machine always reaches state $c_Inc2$ in two transition steps. This property can be expressed in $\mathcal{L}_{\mathcal{MDG}}$ as follows:

$$\mathsf{G}((state = c_Fetch \& input = c_Inc2) \rightarrow (\mathsf{XX}(state = c_Inc2)));$$

**Property 2:** From state $c_Fetch$, if the input is $c_Inc2$, then the machine reaches the state $c_Fetch$ in three transition steps and the counter $pc$ is increased by two. This property can be expressed in $\mathcal{L}_{\mathcal{MDG}}$ as follows:

$$\mathsf{G}((state = c_Fetch \& input = c_Inc2) \rightarrow \mathsf{LET}\ v_1 = pc\ \mathsf{IN}\ \mathsf{XXX}(state = c_Fetch \,\&\, pc = inc(inc(v_1))));$$

**Property 3:** From the state $c_Fetch$, the machine will eventually reach the state $c_Load$ if the input is not $c_No_op$ or $c_Inc1$ or $c_Inc2$ forever. The property can be expressed in $\mathcal{L}_{\mathcal{MDG}}$ as:

$$\mathsf{G}((state = c_Fetch) \rightarrow (\mathsf{F}(state = c_Load)));$$

under the following fairness constraint:

$$!((state = c_Fetch) \rightarrow ((input = c_Inc1) \mid (input = c_No_op) \mid (input = c_Inc2)));$$

These properties are verified by our model checker in less than one second. Table 1 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 is carried out on a SPARC Station 20 with 128 MB of memory.

| | Building the composite machine | | Checking the simplified property | |
|---|---|---|---|---|
| | CPU time (sec) | Memory (MB) | CPU time (sec) | Memory (MB) |
| Property 1 | 0.21 | 0.89 | 0.04 | 0.15 |
| Property 2 | 0.31 | 0.90 | 0.12 | 1.75 |
| property 3 | 0.37 | 1.65 | 0.06 | 0.51 |

**Table 1.** Experimental Results of Property Checking.

Using the decidable fragment of Ground Temporal Logic (GTL) [1], Properties 1 and 2 could be checked, but Property 3 could not be verified, since it is a liveness property. Using the "true symbolic model checking" [6], all the properties could be checked. But when verifying Property 2, as the abstract data *pc* appears in the property, we need to first strip the first-order parts in the formula to obtain a propositional formula $\mathsf{G}((state = c_Fetch \& input = c_Inc2) \rightarrow (\mathsf{XXX}(state = c_Fetch)))$. When the propositional formula is verified, a first-order verification condition need to be generated and verified. Using the ICS model [14, 13], it happens that the abstract counter falls into the class of circuits where finite instantiation cannot be applied and thus it is not possible to compute all the reachable states; therefore, it seems that none of the above properties could be verified.

# 7 Concluding Remarks

We defined a first-order linear-time temporal logic $\mathcal{L}_{\mathcal{MDG}}$ with only the universal path quantifier and developed property checking algorithms for $\mathcal{L}_{\mathcal{MDG}}$. To check a property of $\mathcal{L}_{\mathcal{MDG}}$ on an ASM $M$, we first build additional ASMs for the *Next_let_formulas* (which contain the temporal operator X) that appear in the property, then compose the additional ASMs with $M$, and finally check a simplified property on the composite machine. We use MDGs to encode sets of states and the transition relation. The property checking procedures are based on implicit state enumeration and are carried out fully automatically. We illustrated the application of our property checker on an abstract counter. We have also proven the soundness of our verification procedures in [16].

Since we use first-order logic, the reachability analysis may not terminate [3], thus the property checking may not terminate either. We are currently exploring techniques that migrate this problem [11, 10].

# References

1. Cyrluk D. and Narendran P. Ground temporal logic: A logic for hardware verification. In D. L. Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

2. Emerson E. A. and Lei C. L. Modalities for Model Checking: Branching Time Logic Strikes Back. *Science of Computer Programming*, 8:275–306, 1987.
3. Clarke E. M. and Emerson E., A. Design and Synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, number 131 in Lecture Notes in Computer Science, pages 52–71, New York, 1981. Springer-Verlag.
4. Corella F., Langevin M., Cerny E., Zhou Z., and Song X. State enumeration with abstract descriptions of state machines. In *Proc. IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (Charme'95)*, Frankfurt, Germany, October 1995.
5. Corella F., Zhou Z., Song X., Langevin M., and Cerny E. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, February 1997.
6. Hungar H., Grumberg O., and Damm W. 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.
7. Burch J. R. and Dill D. L. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Proc. Work. on Computer-Aided Verification*, number 818 in Lecture Notes in Computer Science. Springer Verlag, 1994.
8. Burch J. R., Clarke E. M., and McMillan K. L. Symbolic model checking: $10^{20}$ States and Beyond. In *LICS*, 1990.
9. McMillan K. L. *Symbolic model checking: An approach to the state explosion problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1992.
10. Aït Mohamed O., Cerny E., and Song X. MDG-based verification by retiming and combinational transformations. In Magdy A. Bayoumi and Graham Jullien, editors, *Proc. of the Great Lakes Symposium on VLSI (GLS-VLSI'98*, pages 356–361, Lafayette, Louisiana, USA, 1998. IEEE Computer Society Press.
11. Aït Mohamed O., Song X., and Cerny E. On the Non-termination of MDG-based Abstract State Enumeration. In *Proc. IFIP W 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (Charme'97)*, Montréal, October 1997. IFIP, Chapmann & Hall.
12. Coudert O. and Madre J. C. A unified framework for the formal verification of sequential circuits. In *International Conference on Computer-Aided Design*, 1990.
13. Hojati R., Dill D. L., and Brayton R. K. Verifying linear temporal properties of data insensitive controllers using finite instantiations. In *Conference on Hardware Description Languages (CHDL'97)*, April 1997.
14. Hojati R. and Brayton R. K. Automatic datapath Abstraction In Hardware Systems. In *Conference on Computer-Aided Verification*, June 1995.
15. Kurshan R. P. Reducibility in Analysis of Coordination. volume 103 of *Lecture Notes in Computer Science*, pages 19–39. Springer-Verlag, 1987.
16. Xu Y. *Model checking for a first-order branching time temporal logic based on abstract description of state machines.* PhD thesis, D'IRO, University of Montréal, 1998. Draft.