IPSJ Transactions on System LSI Design Methodology Vol. 3 105–117 (Feb. 2010)

Regular Paper

Approximate Invariant Property Checking Using Term-Height Reduction for a Subset of First-Order Logic [⋆1]

Hiroaki Shimizu,[†1] Kiyoharu Hamaguchi[†2] and Toshinobu Kashiwabara[†2]

The use of a subset of first-order logic, called EUF, in model checking can be an effective abstraction technique for verifying larger and more complicated systems. The EUF model checking problem is, however, undecidable. In this paper, in order to guarantee the termination of state enumeration in the EUF-based model checking, we introduce a technique called term-height reduction. This technique is used to generate a finitely represented over-approximate set of states including all the reachable states. By checking a specified invariant property for this over-approximate set of states, we can safely assure that the invariant property always holds for the design, when verification succeeds. We also show some experimental results for a simple C program and a DSP design.

1. Introduction

Model checking [1] is a technique to verify whether hardware or software designs satisfy some designated properties. The technique checks all the reachable states of the design exhaustively, and it has advantages of being able to verify the design with no test pattern and detect all the errors with regard to the designated properties. Some model checking tools, such as SMV [12], SPIN [13], CBMC [14], and UCLID [15], have been put to practical use.

Model checking has still difficulties in handling large and complicated designs. To tackle this problem, abstraction techniques are necessary to use. In this paper, for this purpose, we adopt abstraction by a quantifier-free first-order logic with equality and uninterpreted functions (EUF) [2,3]. In the EUF-based model checking, for example, arithmetic operations are abstracted as function symbols, and the algorithm treats them simply as symbols without considering their semantics.

Model checking using EUF is, however, known to be undecidable [4]. In fact, straightforward state exploration for transition functions defined with EUF terms does not terminate, because the number of terms which possibly occur in state variables can be infinite. Bounded model checking [11], which handles transitions up to a given number of cycles, is an approach for this difficulty, and some bounded model checkers have been developed such as UCLID [15], EUREKA [7] or SAL [6] which utilize SAT solvers for EUF and its extension, or SMT solvers.

This paper addresses *unbounded* invariant property checking using the EUF. We introduce a technique called *term-height reduction* to restrict the number of terms occurring in state variables. When the height of some term which occurs in a state variable exceeds a given limit, its innermost sub-term is replaced by a new variable so that its height is lower than or equal to the limit. Intuitively, this manipulation discards the least recently performed operation to the corresponding term.

This height reduction technique, together with the state reduction technique similar to that by Isles, et al. [5], generates a finitely represented over-approximate set of states including all the reachable states, and guarantees termination of state enumeration The degree of approximation is controlled by the term-height parameter. Although our algorithm is based on explicit state enumeration, in the experiments we performed, state explosion is effectively curtailed.

We applied our technique to a simple C program for Bisection Method and a DSP design, that is, ADPCM encoder. Since both of the systems has an indeterminate number of iteration in their parts, state enumeration does not terminate if we apply a straightforward procedure without term-height reduction. Furthermore, since both of them contain arithmetic operations such as multiplication or division, formal verification without an abstraction technique, or that at Boolean level, is significantly difficult. Our verification algorithm was able to verify them successfully.

The remainder of this paper is organized as follows. We describe related works first, and give the definition of EUF, its state machine, and invariant checking

†1 Presently with IIM Corp.
†2 Osaka University
⋆1 This paper is based on the publication by the authors at International Symposium on Automated Technology for Verification and Analysis 2008 (ATVA2008)

problem. Next, we present the procedure of state traversal with EUF, followed by the detail of our algorithm. We also show the experimental results.

2. Related Works

Unbounded model checking using EUF or its extension has been studied in some literature[4),5),8)]. Isles, et al.[5)] show a state enumeration procedure for transition systems using EUF terms extended with memories. They use some state reduction techniques[4)], which utilize replaceability of sub-terms that comprise two states. Corella, et al.[8)] show a procedure using Multiway Decision Graphs, which can represent characteristic functions for state sets. They also use state reduction techniques similar to that by Isels, et al.[5)]. In both of these works, termination is not guaranteed.

Bryant, et al.[9)] show a criterion for convergence test, which checks if newly added states are all included in the previously enumerated state set. This criterion is formulated as a quantified second-order formula, for which they show a semi-decidable procedure. This criterion gives a precise definition of convergence, but they have also reported that their approach leads to high computational complexity.

The contribution of this paper is an invariant checking method of unbounded state traversal for EUF-based state machines. A novel idea we propose is term-height reduction technique, which is the key for termination of state traversal. In addition to this term-height reduction, our algorithm uses state space reduction techniques named state merging, which was introduced by Isels, et al.[5)]. Using the term-height reduction and state merging together, our algorithm produces a finitely represented over-approximate set of states in the state traversal procedure. Unlike our approach, unbounded model checking, or invariant checking methods in the other literatures[4),5),8)] cannot guarantee termination of the state traversal procedure.

3. EUF and State Machine

We abstract designs by a subset of the first-order logic, called EUF. In this section, we define the syntax and the semantics of EUF and its state machine, and then, invariant checking problem.

```
term := variable | function-symbol(term, ... , term) |
ITE(formula,term,term)
formula := true | false | Boolean-variable |
(term=term) | predicate-symbol(term, ..., term) |
formula ∨ formula | formula ∧ formula | ¬ formula
```

Fig. 1 The syntax of EUF.

3.1 EUF Syntax

EUF is a subset of first-order logic. The logic does not have any quantifier, but has the equal sign as a predefined predicate. It is constructed from *terms* and *formulas*. **Figure 1** shows its syntax. The numbers of arguments, called *arities*, for function symbols and predicate symbols are finite.

Term t has nested structure of function symbols. The term-height of t, denoted by *term-height*(t), is defined as follows:

$$\textit{term-height}(\mathrm{t}) = \begin{cases} \textit{MAX}(\textit{term-height}(t_1), \ldots, \textit{term-height}(t_n)) + 1, & \text{if } t = f(t_1, \ldots, t_n). \\ 0, \text{ if } t \text{ is a variable.} \end{cases},$$

where MAX is the function which returns the maximum from its arguments and f is a function symbol. For example, let `c1` and `c2` be terms, `f` and `g` be function symbols. Then the term-heights of `c1`, `f(c1)` and `g(g(c1,f(c2)),f(c1))` are 0, 1 and 3, respectively.

In this paper, *atomic formulas* are an equation, a predicate and a Boolean variable. An atomic formula and negation of an atomic formula are *literals*. A *product term* is a literal or conjunction of more than one literals. A *disjunction normal form* (DNF) is a product term or disjunction of more than one product terms.

We can assume all of the ITE terms have been removed. This can be done by recursively replacing $t = \mathrm{ITE}(\alpha, t_1, t_2)$ with $(\alpha \wedge (t = t_1)) \vee (\neg\alpha \wedge (t = t_2))$.

3.2 EUF Semantics

For a nonempty domain $\mathcal{D}$ and an interpretation σ, the truth of a formula is defined. The interpretation σ maps a function symbol and predicate symbol of arity k to a function $\mathcal{D}^k \rightarrow \mathcal{D}$ and $\mathcal{D}^k \rightarrow \{\textit{true}, \textit{false}\}$, respectively. Also, σ maps each variable to an element in $\mathcal{D}$. Boolean variables are mapped to $\{true, false\}$.

Valuation of a term t and a formula α, denoted by $\sigma(t)$ and $\sigma(\alpha)$ respectively, are defined as follows. Here, f is a function symbol and p is a predicate symbol. 1) For term $t = f(t_1, t_2, \ldots, t_n)$, $\sigma(t) = \sigma(f)(\sigma(t_1), \sigma(t_2), \ldots, \sigma(t_n))$. 2) For term $t = ITE(\alpha, t_1, t_2)$, $\sigma(t) = \sigma(t_1)$ if $\sigma(\alpha) = true$, otherwise $\sigma(t) = \sigma(t_2)$. 3) For formula $\alpha = p(t_1, t_2, \ldots, t_n)$, $\sigma(\alpha) = \sigma(p)(\sigma(t_1), \sigma(t_2), \ldots, \sigma(t_n))$. 4) For formula $\alpha = (t_1 = t_2)$, $\sigma(\alpha) = true$ if and only if $\sigma(t_1) = \sigma(t_2)$. 5) For formula $\alpha = \neg\alpha_1$, $\sigma(\alpha) = \neg\sigma(\alpha_1)$. 6) For formula $\alpha = \alpha_1 \circ \alpha_2$, where $\circ$ is $\vee$ or $\wedge$, $\sigma(\alpha) = \sigma(\alpha_1) \circ \sigma(\alpha_2)$.

A formula α is *valid* if and only if $\sigma(\alpha) = true$ for any interpretation σ and any domain $\mathcal{D}$.

For simplicity, we introduce a new special constant *TRUE*, and treat $p(t_1, \ldots, t_n)$ and $\neg p(t_1, \ldots, t_n)$ as $p(t_1, \ldots, t_n) = TRUE$ and $\neg(p(t_1, \ldots, t_n) = TRUE)$, respectively. Then each literal can be treated as an equation, a Boolean variable or negative forms of them.

3.3 EUF State Machine

We start with an example. An EUF state machine is given as a set of transition functions.

Example 1 We consider a program code using EUF in the following:

```
0:  while (t1!=t2){
        t1 := f(t1,t2);
    }
1:  t2 := g(t2);
```

The transition functions for this code are as follows:

$$\mathtt{b1}' := \mathtt{ITE}(\mathtt{b1} = 0, \mathtt{ITE}(\mathtt{t1} = \mathtt{t2}, 1, 0), 1)$$
$$\mathtt{t1}' := \mathtt{ITE}(\mathtt{b1} = 0, \mathtt{ITE}(\mathtt{t1} = \mathtt{t2}, \mathtt{t1}, \mathtt{f}(\mathtt{t1}, \mathtt{t2})), \mathtt{t1})$$
$$\mathtt{t2}' := \mathtt{ITE}(\mathtt{b1} = 1, \mathtt{ITE}(\mathtt{t1} = \mathtt{t2}, \mathtt{g}(\mathtt{t2}), \mathtt{t2}), \mathtt{t2})$$

Here, $b1$ is a Boolean state variable. $t1$ and $t2$ are state variables interpreted in the EUF logic, called term state variables. The variables with $'$ are next state variables. f and g are function symbols. The transition functions are a natural extension of usual transition functions for sequential state machines. For example, the second transition function means that the state variable $t1$ at the next cycle is $t1$ of the present cycle if $b1 = 0$ and $t1 = t2$, and $f(t1, t2)$ of the present cycle if $b1 = 0$ and $t1! = t2$, and so on. $\square$

More precisely, an EUF state machine is defined as follows: To describe transition functions, we assume four types of variables as follows: 1) Boolean state variables: $b_1, \ldots, b_m$, 2) term state variables: $t_1, \ldots, t_n$, 3) Boolean variables: $a_1, \ldots, a_p$, 4) term variables: $c_1, \ldots, c_q$.

The variables of 1) and 3) are Boolean variables, and those of 2) and 4) are variables of the EUF. We introduce next state variables $b'_1, \ldots, b'_m$ and $t'_1, \ldots, t'_n$ corresponding to $b_1, \ldots, b_m$ and $t_1, \ldots, t_n$ respectively. Then, transition functions are described by $b'_i := F_i$ $(1 \le i \le m)$ and $t'_j := T_j$ $(1 \le j \le n)$, where F_i is a formula and T_j is a term. F_i and T_j do not contain any next state variables. Some of Boolean variables and term variables are specified as inputs. This means that each of these input variables at each step is treated as distinct, and an interpretation for such a variable at the i-th step can be different from that at the j-th step $(i \ne j)$.

We call the following formula *transition relation*:

$$\bigwedge_{1 \le i \le m} (b'_i = F_i) \wedge \bigwedge_{1 \le j \le n} (t'_j = T_j) \qquad (1)$$

The behavior of an EUF state machine depends on an initial state and a sequence of interpretations for each step. At an initial state, each Boolean variable is mapped to *true* or *false*, and each term variable is mapped to an element in $\mathcal{D}$. We define an *interpretation sequence* as $\tilde{\sigma} = (\sigma^0, \sigma^1, \ldots)$, where σ^i is an interpretation at the i-th step. Note that the length of an interpretation sequence is infinite. Any interpretation sequence is required to satisfy the following conditions.

- The interpretations of term variables, Boolean variables, function symbols and predicate symbols are the same at every step, except for input variables. In other words, for each variable c_j, each Boolean variable a_j, each function symbol f_j and each predicate symbol p_j, $\sigma^0(c_j) = \sigma^i(c_j)$, $\sigma^0(a_j) = \sigma^i(a_j)$, $\sigma^0(f_j) = \sigma^i(f_j)$ and $\sigma^0(p_j) = \sigma^i(p_j)$ for all i.

- As for each input term variable c_j and for each input Boolean variable a_j at the i-th step, we introduce new variable names c_j^i and a_j^i for distinction. $\sigma^i(c_j^i)$ and $\sigma^i(a_j^i)$ are given arbitrary at the i-th step. $\sigma^k(c_j^i) = \sigma^i(c_j^i)$ $(k = i, i+1, \ldots)$, and $\sigma^k(a_j^i) = \sigma^i(a_j^i)$ $(k = i, i+1, \ldots)$. The interpretations of these variables for $k = 0, 1, \ldots, i-1$ are not defined.
- For transition functions $b'_j := F_j$ and $t'_j := T_j$, $\sigma^{i+1}(b_j) = \sigma^i(F_j)$, $\sigma^{i+1}(t_j) = \sigma^i(T_j)$ for all i.
- For an initial state in which each term state variable t_j is mapped to variable c_j, $\sigma^0(t_j) = \sigma^0(c_j)$.

We call an interpretation sequence which satisfies the above conditions a *normal interpretation sequence*. Note that σ^i and σ^{i+1} are identical except for the state variables b_j and t_j, which depend on each step, and the input variables c_j^{i+1} and a_j^{i+1}, which are not defined for the i-th step.

In this paper, we assume that interpretation sequences are normal.

3.4 Invariant Checking Problem Using EUF

The inputs of our problem are an EUF state machine M, which models a design we want to verify, and a property P, which is composed from Boolean state variables, term state variables, Boolean variables and term variables. The invariant checking problem is to check whether M always satisfies P.

Since the contents of the Boolean or term state variables used in P depends on each state, the formula to be checked at each state also depends on each state. The details are given in Section 4.3.

4. State Traversal and Invariant Checking

Invariant checking requires a procedure for enumerating the reachable states from the given initial states. We show a basic state traversal procedure for an EUF state machine. The resulting state transition graph has generally infinite number of states.

4.1 State Traversal Procedure

Each state s is composed of the following two elements:

- State vector $\overrightarrow{v} = (\overrightarrow{b}, \overrightarrow{t})$
- Condition set $C \subseteq T \times T \times Rel$

where T is the set of terms which do not contain ITE terms, and $Rel = \{=, \neq\}$.

We suppose $\overrightarrow{b} = (b_1, \ldots, b_m)$ and $\overrightarrow{t} = (t_1, \ldots, t_n)$, where for $1 \leq i \leq m$, b_i is *true* or *false*, and for $1 \leq j \leq n$, t_j is a term. C is the set of all conditions which must be satisfied in order to reach the state. The conditions in C can also be regarded as constraints to the terms assigned to state variables.

Examples of states are shown in **Fig. 2**. Each box represents a state. This example is explained later.

For a state s, we consider a specific interpretation which makes true all the formulas in C. Then, each state variable has a specific value. In this sense, state s is truly a specific state. We can also consider arbitrary interpretations satisfying C. Then, we can consider many states corresponding to s, that is, we can regard this symbolically expressed state s as representing *a set of states* in actuality. In the following, we mainly view each state in the latter sense.

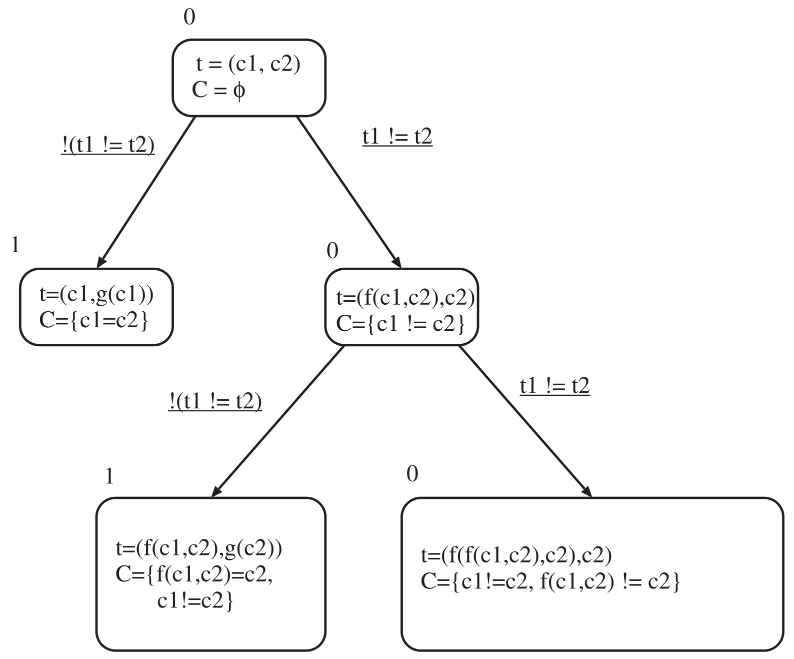

Fig. 2 Simple State Traversal.

As for state traversal, based on the transition relation, we enumerate all the reachable states from initial states one by one.

Example 2 Figure 2 shows a part of state traversal for the transition functions in Example 1. Each box corresponds to a state. The binary value at the left upper side of each state represents the value of $b1$ at the state. The initial values for state term variables $t1$ and $t2$ are $c1$ and $c2$ respectively, where $c1$ and $c2$ are EUF variables in actuality. C is a condition set at each state. State traversal for this example does not terminate, because the number of the occurrences of function symbol f can be arbitrarily large, when we follow branches labeled by $t1! = t2$ repeatedly. $\square$

In the following, we explain precise procedure of state traversal. Firstly, we perform a preprocessing to convert the transition relation into DNF (disjunctive normal form).

Then, the next state $s' = (\overrightarrow{v}', C')$ is constructed from the current state $s = (\overrightarrow{v}, C)$ based on the following rule. First, we replace all the current state variables occurring in the transition relation in DNF, with the corresponding values or terms in $\overrightarrow{v}$. Then, we can get the formula $\alpha \triangleq \alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_p$, where each α_i $(1 \leq i \leq p)$ is a product term whose variables are next state variables b'_k $(1 \leq k \leq m)$ and t'_l $(1 \leq l \leq n)$, Boolean variables and term variables. Note that α_i does not contain any current state variable, because it has been replaced by a value or a term in $\overrightarrow{v}$.

For each $\alpha_i \triangleq \beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_q$, where β_j $(1 \leq j \leq q)$ are literals, the next state s' is generated as follows. The contents of $\overrightarrow{v}'$ and C' are initialized with the ones of $\overrightarrow{v}$ and C, respectively. Next, for each β_j, the appropriate step shown in the below is performed.

Here, we assign an arbitrary value in { *true*, *false* } to each input Boolean variable in α_i, and we introduce a term variable with a new distinct name to each input term variable in α_i. This means that multiple next states can be generated for each α_i.

(1) If β_j is an equation $b'_k = b$, where b'_k is a next Boolean state variable and b is a propositional formula, the Boolean state variable b_k at the next state is assigned to the value of b.

(2) If β_j is an equation $t'_l = t$, where t'_l is a next term state variable and t is a term, the term state variable t_l at the next state is assigned to t.

(3) If β_j is an equation $t_1 = t_2$ or its negation $\neg(t_1 = t_2)$, where t_1 and t_2 are terms which do not contain any next state variable, the new condition $(t_1, t_2, =)$ or $(t_1, t_2, \neq)$ is added to C', respectively.

(4) If β_j is an Boolean variable or its negation, the Boolean variable is assigned to *true* or *false* so that β_j is true.

The above Case (3) means that C' contains the conditions which enable the transition from s to s', in addition to C. Thus, we say that state s' is reachable from state s in one step if s' can be composed from s by applying the above rule once, and the conjunction of the conditions in C' is satisfiable. For an EUF state machine M and an initial state s^0, $(s^0, s^1, \ldots)$ is a *state transition sequence* if s^{i+1} is reachable from state s^i in one step for any $i \geq 0$.

4.2 State Traversal and Interpretation Sequence

Given a state transition sequence $\tilde{s} = (s^0, s^1, \ldots)$, we can construct a normal interpretation sequence $\tilde{\sigma} = (\sigma^0, \sigma^1, \ldots)$. Note that, in the state traversal procedure, we have chosen α_j as well as values or term variables to be inserted to input variables, to generate s^{i+1} from s^i.

This can be done by 1) fixing the valuations of non-input term or Boolean variables, function or predicate symbols; 2) giving the valuations to state variables at the $i+1$-th step so that $\sigma^{i+1}(b_j) = \sigma^i(F_j)$ and $\sigma^{i+1}(t_j) = \sigma^i(T_j)$ for all $i = 0, 1, \ldots$; 3) giving to input Boolean variables at the i-th step, the same values we have chosen in the generation of s^{i+1}; 4) giving some valuations for term variables which were inserted to the input term variables at the i-th step.

When both equations $\sigma^{i+1}(b_j) = \sigma^i(F_j)$ and $\sigma^{i+1}(t_j) = \sigma^i(T_j)$ are satisfied, all the conditions for the transitions embedded in F_j and T_j must be true. Otherwise neither of the above equations can hold. Since all the transition conditions (reachable conditions) are recorded at C^i, we can show the following property.

Property 1 For a normal interpretation sequence, the conjunction of the conditions in C^i is true, under interpretation σ^i. $\square$

4.3 Invariant Checking

A property P, which is restricted to an invariant in this paper, is an EUF formula containing Boolean state variables, term state variables, Boolean variables and term variables.

For a state transition sequence $s = (s^0, s^1, \ldots)$, we denote by P_{v^i} the formula whose state variables are all replaced with corresponding contents of $\overrightarrow{v}^i$, where state $s^i = (\overrightarrow{v}^i, C^i)$. The property P holds at state s^i if the formula P^i defined below is valid.

$$P^i \triangleq \Big(\bigwedge_{(t_1, t_2, R_e) \in C^i} t_1 R_e t_2 \Big) \;\rightarrow\; P_{v^i} \tag{2}$$

Invariant checking is the problem of deciding whether P^i is valid at any reachable state s^i.

5. Over-approximation for Reachable States

In this section, we propose an over-approximate algorithm using term-height reduction. We introduce the maximum value of term-height *maxh* and restrict the heights of terms occurring in the state variables. The algorithm guarantees that a property P always holds if P holds for the approximated state set, otherwise the verification result is inconclusive.

Here, we explain our approach. As was stated, the original invariant checking problem is undecidable. The intention of the following algorithm is to generate a finitely represented over-approximation set of states. As we described in Section 4.1, a "state" in our state traversal procedure can be regarded a set of states under many interpretations satisfying its reachability condition set. The effect of the term-height reduction is to extend the set of represented states. This is because term-height reduction removes some of constraints among the terms stored in state vectors. In other words, it causes over-approximation at each state. As we see in the following subsections, term-height reduction together with state merging generates a finite state transition graph for a given *maxh*. The resulting state transition graph can be regarded as an over-approximation of the original non-approximated and possibly infinite state transition graph. Then, we can solve the invariant checking problem, by checking the given property at each state of the over-approximated state transition graph.

5.1 Overall Algorithm

Figure 3 shows the overall procedure. The details of term-height reduction and state inclusion check are given in the following subsections.

The procedure is given a property P, the initial state s_{init}, the transition relation $Trans$ of the EUF state machine and $maxh$ as the maximum height for term-height reduction procedure. RS stores states which are reachable from the initial state. CS stores states which are candidates for reachable states.

The candidate state set CS is set and updated in line 2 and 15. In the while loop in lines 3-17, the procedure picks a candidate state s from CS (line 4), and lines 6-8 check whether s should be added to RS. Firstly, it checks whether the reachable condition C of $s = (\overrightarrow{v}, C)$ is satisfiable in line 6, which means s is reachable from the initial state. This can be done by satisfiability checking of the conjunction of the formulas in C. Line 7 checks whether s is contained in RS, in terms of "state inclusion relation" discussed in Section 5.3. Line 8 checks whether P holds in s, based on formula (2).

```
     AppInvChk($P$,$s_{init}$,$Trans$,$maxh$)
     //$P$: property, $s_{init}$: initial state, $Trans$: transition relation
     //$maxh$: maximum term-height
1:   $RS = \{\}$; // Reachable state set
2:   $CS = \{s_{init}\}$; // Candidate state set for $RS$
3:   while ($CS \neq \{\}$) {
4:       choose $s \in CS$;
5:       $CS = CS - \{s\}$;
6:       if ($\neg$SatisfiableReachableCondition($s$)) continue;
7:       if (StateInclusionCheck($s, RS$)) continue;
8:       if ($\neg$PropertyCheck($s, P$)) {return("Inconclusive");}
9:       $RS = RS \cup \{s\}$;
10:      $NS$ = ComputeNextState($s, Trans$);
11:      while ($NS \neq \{\}$) {
12:          choose $ns \in NS$;
13:          $NS = NS - \{ns\}$
14:          $ns'$ = TermHeightReduction($ns, maxh$);
15:          $CS = CS \cup \{ns'\}$
16:      }
17:  }
18:  return ("Property holds");
```

Fig. 3 Approximate Invariant Checking Procedure.

For the reachable state s, line 10 computes the set of next states NS, using the procedure shown in Section 4.1. The procedure picks states from NS one by one, and perform “term height reduction” to them with the given parameter $maxh$ (line 14). The resulting state ns' is added to CS. Term-height reduction is discussed in Section 5.2.

If the number of reachable states is finite, then eventually this procedure terminates, when all the candidate states turn out to be included in RS by checking in line 7, and CS becomes empty.

5.2 Term-height Reduction

Term-height reduction is the key idea for our algorithm. This operation keeps the term-heights of the terms stored at state vectors less than or equal to a given limit $maxh$. By doing this, the number of terms stored at state vectors becomes finite, if we ignore the variables newly introduced in the state traversal procedure for input Boolean variables and input term variables. How to handle these variables is described in Section 5.3.

When the terms whose heights are larger than $maxh$ occur in a state vector $\overrightarrow{v}$ or a condition set C, we reduce their term-heights by replacing the subterms with new variables, so that the term-heights are all restricted to less than or equal to $maxh$.

Definition 1 For a term t whose height is larger than 0, the *reduced subterms of t*, denoted RT_t, are defined as those satisfying all the following conditions:

(1) For any $t_r \in RT_t$, $\textit{term-height}(t_r) = 1$.

(2) Let t' be the term obtained from t by replacing all the subterms of t in RT_t with new variables. Then, $\textit{term-height}(t) = \textit{term-height}(t') + 1$.

(3) If some term is deleted from RT_t, condition (2) does not hold.

□

Term-height reduction for a term t is a manipulation replacing each subterm of t in RT_t with a new variable. For example, the reduced subterm of term `g(g(c1,f(c2)), f(c1))` is `f(c2)`. By applying term-height reduction for this term, we obtain term `g(g(c1,c3),f(c1))`, where `c3` is a new variable. Also, the reduced subterms of `g(g(c1,c3),f(c1))` are { `g(c1,c3)`, `f(c1)` }. We apply term-height reduction to this term and obtain the term `g(c4,c5)`, where `c4` and `c5` are new variables. By applying term-height reduction for a term, we can decrease the term-height just by one. Note that for any term t whose height is larger than 0, the reduced subterms of t exist uniquely. Also, our algorithm replaces the same subterm with the same variable.

We keep the record of the mapping from a subterm to a new variable until the algorithm terminates. When term state variables are updated by new terms in state traversal, this mapping is applied to all the subterms included. Note that the record of the mapping can get larger in the progress of state traversal, but this does not affect termination of the procedure, because the termination condition does not depend on this record of mapping.

Since term-height reduction removes constrains among terms stored in state vectors, it causes over-approximation at each state. This leads to the following theorems.

Theorem 1 For any EUF formula F, F is valid if the formula F' obtained by applying term-height reduction for a term t in F is valid.

(*Proof*) Suppose that F' is valid. Let us evaluate F with an arbitrary interpretation σ. Then, t has some valuation $\sigma(t)$. Suppose that t in F is replaced to term variable v in F' by term-height reduction. We can construct an interpretation σ' for F' such that $\sigma'(v) = \sigma(t)$, and the other valuations do not change. Since F' is valid, $\sigma'(F')$ is true. Then, $\sigma(F)$ is also true, because the corresponding t and v have the same valuation, and the valuations for the other variables, functions or predicates do not change. For any interpretation, the above argument holds. Thus, if F' is valid, so is F. □

When term-height reduction is repeated for more than one term in F, the similar theorem holds for the resulting formula F'.

The term-height reduction procedure, TermHeightReduction$(s, maxh)$, which generates state $s' = (\overrightarrow{v}', C')$ from $s = (\overrightarrow{v}, C)$ with parameter $maxh$, is as follows. Here, $\overrightarrow{v}' = (\overrightarrow{b}', \overrightarrow{t}')$.

(1) s' is initialized with s.

(2) If $\overrightarrow{v}'$ or C' contains terms whose heights are larger than $maxh$, their heights are reduced repeatedly until becoming less than or equal to $maxh$.

(3) The conditions in C' containing any variable which does not occur in $\overrightarrow{t}'$ are all deleted.

The above step (3) can remove some constraints from C'. As for formula (2), we can obtain the following theorem. Here, similarly to formula (2), we define P^s and $P^{s'}$ for s and s' respectively.

$$P^s \triangleq (\bigwedge_{(t_1,t_2,R_e)\in C} t_1 R_e t_2) \rightarrow P_v, \quad P^{s'} \triangleq (\bigwedge_{(t_1,t_2,R_e)\in C'} t_1 R_e t_2) \rightarrow P_{v'}$$

Theorem 2 If $P^{s'}$ is valid, then P^s is valid.
(*Proof*) Note that P^s can be expressed as:

$$P^s = \bigvee_{(t_1,t_2,R_e)\in C} \neg(t_1 R_e t_2) \vee P_v$$

Let us s'' be the state before applying step (3) in the above term height reduction procedure. We can construct $P^{s''}$ similarly. From Theorem 1, if $P^{s''}$ is valid, then P^s is valid. The conditions of form $\neg(t_1 R_e t_2)$ in $P^{s'}$ is fewer than those in $P^{s''}$ because of step (3). Thus, if $P^{s'}$ is valid, then $P^{s''}$ is valid. This concludes the theorem. □

5.3 State Merging Based on Inclusion Relation

State merging we explain in this section is basically the same as that introduced by Isels, et al. [5]. We formulate it in our terminology. We also show a theorem on relationship between state merging and validity checking (Theorem 3), which was not explicitly given before [5].

Since we introduce new variables in the state traversal and term-height reduction, the number of terms which appear in state vectors can get larger arbitrarily. State merging avoids this. Suppose that we have state s with term state vector $(f(c1, c2), c2)$ and condition set $(c1 \neq c2)$, and a state s' with $(f(c3, c2), c2)$ and $(c3 \neq c2)$, respectively, where both of the Boolean state variables have the same values. Since we consider assignment of arbitrary values to $c1$ and $c3$ under constraints by the condition sets, we can safely view the two states as the same state.

State merging based on inclusion relation is an extension of this observation. Suppose that we have two states s and s' with condition sets C and C', respectively. If their state vectors match syntactically with each other *under appropriate renaming of variables*, and condition C' *implies* condition C, then we can safely merge s' to s. That is, even if delete s' and redirect transition edges to s' to s, we can have a proper state transition graph. Renaming is done by finding a mapping from original variable names to a set of new variable names. Its details are as follows.

For a state $s = (\overrightarrow{v}, C)$, where $\overrightarrow{v} = (\overrightarrow{b}, \overrightarrow{t})$ and $\overrightarrow{t} = (t_1, \ldots, t_n)$, let V_t be the set of variables occurring in $\overrightarrow{t}$, D be a set of variables $\{d_1, d_2, \ldots, d_{|V_t|}\}$, where $V_t \cap D = \phi$, and map_t^D be a bijective function from V_t to D. We denote by $map_t^D[t_i]$ the term obtained from a term t_i in which each variable $c \in V_t$ is replaced with $map_t^D(c)$. Furthermore, we denote by $map_t^D[\overrightarrow{t}]$ the vector of terms obtained from $\overrightarrow{t}$ in which each term t_i is replaced with $map_t^D[t_i]$. Also, we denote by $map_t^D[C]$ the condition set obtained from C in which each condition (t_1, t_2, R_e) is replaced with $(map_t^D[t_1], map_t^D[t_2], R_e)$.

Definition 2 For two states $s = (\overrightarrow{v}, C)$ and $s' = (\overrightarrow{v}', C')$, where $\overrightarrow{v} = (\overrightarrow{b}, \overrightarrow{t})$, $\overrightarrow{v}' = (\overrightarrow{b}', \overrightarrow{t}')$, we say "*s includes s'*", denoted by $s \geq s'$, if the following conditions are all satisfied.

(1) $\overrightarrow{b} = \overrightarrow{b}'$
(2) $|V_t| = |V_{t'}|$
(3) For a set of variables D, there exist two functions map_t^D and $map_{t'}^D$ such that:
- $map_t^D[\overrightarrow{t}] = map_{t'}^D[\overrightarrow{t}']$ and
- $\bigwedge_{(t'_1,t'_2,R'_e)\in map_{t'}^D[C']} (t'_1 R'_e t'_2) \longrightarrow \bigwedge_{(t_1,t_2,R_e)\in map_t^D[C]} (t_1 R_e t_2)$ is valid

The third condition in Definition 2 means that when we compare $\overrightarrow{t}$ with $\overrightarrow{t}'$, or C with C', we focus on the forms of terms and disregard the names of variables, to check the term equivalence between $\overrightarrow{t}$ and $\overrightarrow{t'}$ and to check the inclusion relation between conditions C and C'.

In our implementation, the terms in $\overrightarrow{t}$ are compared with those in $\overrightarrow{t}'$ from the top of the lists, one by one. If renaming of the variables can make the two terms syntactically equivalent, then the two mappings map_t^D and $map_{t'}^D$ are updated. Otherwise, checking the conditions of Definition 2 is aborted. If construction of map_t^D and $map_{t'}^D$ is successful, the implication of C by C' is checked under the mappings, which can be done by using some SMT solver.

In a state traversal, for a state r and its next state s', if $s \geq s'$ for some state s we have already visited, then we move from r to s instead of s', and do not visit s' and its successors. We say that "*s' is merged into s*." s is said to be a merging state of s'.

In Fig. 3, procedure StateInclusionCheck(s', RS) checks, based on the above inclusion relation in Definition 2, whether there exists some state $s \in RS$ which includes s'.

As for state inclusion, we can obtain the following theorem. We define P^s and $P^{s'}$ as in Section 5.2.

Theorem 3 Let $s \geq s'$. If P^s is valid, then $P^{s'}$ is valid.
(*Proof*) Suppose that P^s is valid. Consider an interpretation σ' for $P^{s'}$. We construct σ for P^s so that each pair of variables v in s and v' in s' such as $map_t^D(v) = map_{t'}^D(v)$ has the same interpretation, that is, $\sigma(v) = \sigma'(v')$. If $\bigwedge_{(t'_1,t'_2,R_e)\in C'} t'_1 R_e t'_2$ in $P^{s'}$ is false under σ', then $P^{s'}$ is true. Otherwise, $\bigwedge_{(t_1,t_2,R_e)\in C} t_1 R_e t_2$ is true under σ, because $\bigwedge_{(t'_1,t'_2,R'_e)\in map_{t'}^D[C']} (t'_1 R'_e t'_2) \rightarrow \bigwedge_{(t_1,t_2,R_e)\in map_t^D[C]} (t_1 R_e t_2)$ is valid (Condition (3)). Then, P_v is also true under σ, because P^s is valid.

Thus, $P^{s'}$ is also true under σ', because the corresponding variables have the same valuation both under σ and σ', and the valuations for the other variables, functions or predicates are also the same. For any interpretation σ', the above argument holds. Therefore, if P^s is valid, then $P^{s'}$ is valid. □

Example 3 **Figure 4** shows state traversal for the transition functions in Example 1 and 2 under *maxh* = 1. When the transition from s to s' occurs, the height of `t1` gets larger than *maxh* and term-height reduction is performed. Then, $s \geq s'$ holds under the domain $D = \{\texttt{d1}, \texttt{d2}\}$ and functions $map_t^D = \{\texttt{c1} \rightarrow \texttt{d1}, \texttt{c2} \rightarrow \texttt{d2}\}$, $map_{t'}^D = \{\texttt{c3} \rightarrow \texttt{d1}, \texttt{c2} \rightarrow \texttt{d2}\}$. Therefore, s' is merged into s, and s has a self-loop. □

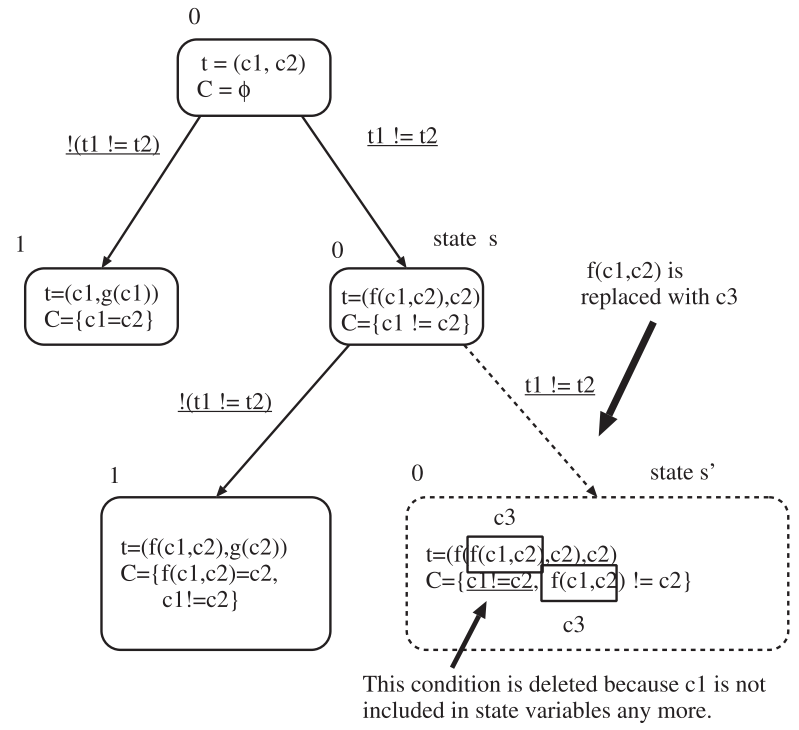

Fig. 4 State Traversal Example (*maxh* = 1).

5.4 Termination and Correctness

The state transition graph obtained by the procedure in Fig. 3 using term-height reduction and state merging is called *approximate state transition graph.* The infinite state graph obtained by the procedure in Section 4.1 without term-height reduction and state merging is called *non-approximate state transition graph.* The following is the key property for termination of our algorithm.

Theorem 4 The size of any approximate state transition graph is finite.
(*Proof*) Suppose that term-height is restricted to a finite value. Then, the number of term forms occurring in a state vector is finite because a function and a predicate have the finite number of arguments. Also, the number of elements

in a condition set is finite because the conditions containing any variable which does not occur in the state variables are all deleted. Thus, disregarding the name of variables, there exists only the finite number of components for a state. Since state merging resolves the difference among variable names, the size of the abstract state transition graph is finite. □

Theorem 5 For any non-approximate state transition graph G and its approximate state transition graph G_a, a property P holds at any state in G if P holds at any state in G_a.
(*Proof*) Firstly we show that, for any possible normal interpretation sequence $\tilde{\sigma} = (\sigma^0, \sigma^1, \ldots,)$ in G, we can have $\tilde{\sigma_a} = (\sigma_a^0, \sigma_a^1, \ldots,)$ in G_a such that the valuations of term state variables and Boolean state variables at each i-th step under σ^i equal to those under σ_a^i.

Suppose that we choose an arbitrary state sequence $\tilde{s} = (s^0, s^1, \ldots)$ in G, and a normal interpretation sequence $\tilde{\sigma} = (\sigma^0, \sigma^1, \ldots)$ along $\tilde{s}$. We can find a state transition sequence $\tilde{s_a} = (s_a^0, s_a^1, \ldots)$ in G_a which corresponds to $\tilde{s}$ as follows. In the state traversal procedure (Section 4.1), we choose α_j as well as values or terms to be inserted to input variables, in order to generate s^{i+1} from s^i in G. Since we use the same procedure to compute next states in G and G_a, we can choose $\tilde{s_a} = (s_a^0, s_a^1, \ldots)$ so that generation of s_a^{i+1} from s_a^i in G_a is based on the same α_j as that of s^{i+1} from s^i in G.

Suppose that we have constructed a partial interpretation sequence up to i-th step $(\sigma_a^0, \sigma_a^1, \ldots, \sigma_a^i)$ which is equal to $(\sigma^0, \sigma^1, \ldots, \sigma^i)$ in terms of valuations of term state variables and Boolean state variables at each step.

The state traversal procedure 1) generates, from s_a^i, state $s_a^{i+1\prime}$ as a next state candidate; 2) applies term-height reduction to $s_a^{i+1\prime}$ to obtain $s_a^{i+1\prime\prime}$ when necessary; 3) perform state merging for $s_a^{i+1\prime\prime}$, when possible, to obtain s_a^{i+1}.

We give the same valuations to the corresponding sub-terms or variables in s^{i+1} and $s_a^{i+1\prime}$, that is, we can construct an interpretation $\sigma_a^{i+1\prime}$ for $s_a^{i+1\prime}$ which gives the same valuations of term variables and Boolean variables as σ^{i+1}. Furthermore, as in the proofs in Theorem 1 and 3, we can also construct interpretations $\sigma_a^{i+1\prime\prime}$ for $s_a^{i+1\prime\prime}$, and then σ_a^{i+1} for s_a^{i+1}, which give the same valuations of term variables and Boolean variables as σ^{i+1}.

The above argument assures that any possible normal interpretation sequence in G is also a normal interpretation sequence in G_a.

Next, we discuss the the validity of the formula (2) at each state. Theorem 3 and 2 assure that proving the validity of P at state s_a^{i+1} in G_a implies the validity at state $s_a^{i+1\prime\prime}$, and then that at $s_a^{i+1\prime}$ in G_a.

The remaining task is to show that, if formula (2) at state $s_a^{i+1\prime}$ is valid, then that at state s^{i+1} is also valid. Suppose that formula (2) is valid at $s_a^{i+1\prime}$. Choose an arbitrary partial normal interpretation sequence $(\sigma^0, \sigma^1, \ldots, \sigma^i, \sigma^{i+1})$ for G. As in the above, we can construct $(\sigma_a^0, \sigma_a^1, \ldots, \sigma_a^i, \sigma_a^{i+1\prime})$ for G_a.

Furthermore, from Property 1, we can say the left-hand sides of the implication in formula (2) (that is, $\bigwedge_{(t_1,t_2,R_e)\in C} t_1 R_e t_2$) at state s^i, s^{i+1} and s_a^i are true under σ^i, σ^{i+1} and σ_a^i respectively. Then, that at $s_a^{i+1\prime}$ is true under $\sigma_a^{i+1\prime}$, because the same conditions are added to C along the transition from s^i and s^{i+1}, and that from s_a^i and $s_a^{i+1\prime}$, and both of σ^{i+1} and $\sigma_a^{i+1\prime}$ assign the same valuations to the same conditions.

The right-hand sides of the implication in formula (2) at s^{i+1} and $s_a^{i+1\prime}$ have the same valuations. Since we have assumed the validity of formula (2) at state $s_a^{i+1\prime}$, the right-hand sides are both true. Thus, formula (2) at s^{i+1} is true. This argument holds for any normal interpretation sequence $(\sigma^0, \sigma^1, \ldots, \sigma^i, \sigma^{i+1})$. Therefore, we can conclude that, if formula (2) at state $s_a^{i+1\prime}$ is valid, then that at state s^{i+1} is also valid. □

6. Experimental Results

We implemented our algorithm in the C++ language and performed some experiments with Intel Core 2 duo 2.4 GHz of 4 GB Memory under Mac OS X 10.5.6. We used CVC3 16) as an EUF SAT solver. The SAT solver was used to check, the satisfiability of reachable conditions, formula (2) and the third condition of Definition 2.

To the best of our knowledge, only a small number of benchmarks written in form of EUF state machines, are available 6),15). Such benchmarks, intended for bounded model checking, contain features our method cannot handle presently, such as unbounded memories or list structures. In this paper, we prepared designs

```
float bisect(float left, float right, // left < right
        float diff, float (*f)(float)) // diff >= 0 is an allowed error
{
    float mid, fleft, fright, fmid;

    fleft = f(left);
    fright = f(right);
    if (fleft == 0) return left;
    if (fright == 0) return right;
    if (samesign(fleft, fright)) {
        exit;
    }
    while (true) {
        mid = (left + right) / 2;
        if (mid - left <= diff) break;
        if (right - mid <= diff) break;
        fmid = f(mid);
        if (fmid == 0) return mid;
        if (samesign(fmid, fleft)) {
            left = mid;
            fleft = fmid;
        } else {
            right = mid;
            fright = fmid;
        }
    }
    return mid;
}
```

Fig. 5 Pseudo Code for Bisection Method.

of a simple C program for Bisection Method and an ADPCM encoder and applied our algorithm.

The run-times we show in this section do not include construction of a DNF of the transition relation from transition functions. In the following examples, we gave the transition relations in DNF as inputs.

6.1 Bisection Method

Bisection method is an algorithm for solving an equation. We show its pseudo code in **Fig. 5**. The number of executions of the while body depends on input value `diff`, and is indeterminate.

We verify equivalence of returned values between BISECT1 which executes the code sequentially, and BISECT2 which is obtained by modifying BISECT1 to execute the loop body in one step. Both of the obtained EUF state machines were executed concurrently.

Table 1 Experimental Result: Bisection Method.

maxh	*# of new vars.*	*# of states*	*time(sec)*	*result*
0	7	18	5.3	inconclusive
1	6	113	46.8	successful
2	5	113	46.7	successful
3	6	117	49.3	successful
4	7	127	54.0	successful
5	8	205	94.2	successful
6	13	283	125.1	successful
7	14	448	220.1	successful
8	27	595	301.4	successful
9	28	964	560.1	successful

The experimental result is shown in **Table 1**, where *# of new vars.* means the total number of new variables introduced by term-height reduction and *# of states* means the number of states in the obtained abstract state transition graph.

In this experiment, the result for $maxh = 0$ is "Inconclusive", that is, the algorithm cannot judge whether the invariant property holds or not. The reason of the very short run-time for this case is that the algorithm stops state generation, as soon as it detects that the invariant property does not hold at a newly generated state (line 8 in Fig. 3).

The rest of the results for various *maxh*'s are all successful, that is, the invariant which claims the equivalence of computed values holds. In this sense, the result for the first and second lines ($maxh = 0$ and 1) are enough to show the correctness. The other lines are given to show the change of performance to various *maxh*'s.

6.2 ADPCM Encoder

An ADPCM (Adaptive Differential Pulse Code Modulation) encoder transforms sound data to digital data. The high-level description of an ADPCM encoder [17] is written in the C language and has about 70 lines. This description also contains multiplication and division. Furthermore, since it has a loop structure whose number of iteration depends on an input parameter, state enumeration for it does not terminate if our technique is not used. The loop structure contains 9 if-branches.

We verify equivalence of output values between ADPCM1 which executes the

Table 2 Experimental Result: ADPCM Encoders.
All of the results are all successful in this experiment.

maxh	*# of new vars.*	*# of states*	*time (sec)*	*result*
0	47	266	161.2	successful
1	30	1016	667.0	successful
2	21	1974	1273.7	successful
3	16	2956	1952.0	successful

high-level description sequentially, and ADPCM2 which is obtained by modifying ADPCM1 to execute the loop body in less steps. **Table 2** shows the experimental result.

In this experiment, the shown results for various *maxh*'s are all successful, that is, the invariant holds.

6.3 Consideration

Without term-height reduction, in both of the above examples, the term-heights occurred at state variables continue to get larger infinitely, as state traversal proceeds. For larger *maxh*, state merging is delayed at later steps, and as a result, both of the number of states and the run-time increase. This suggests that we should increase *maxh* from a smaller value.

On the other hand, verification for the design which does not contain feedback structures, such as filter designs, is expected to terminate without term-height reduction, because the term-height of any term state variable does not exceed some constant. In this case, term-height reduction would not be necessary, and can even degrade the performance.

6.4 Comparison with Other Works

The proposed algorithm handles unbounded state traversal and invariant checking for EUF state machines. Other works [4),5),8)] which handle the same problem do not guarantee the termination of the procedure. In actuality, the algorithms proposed by the above previous works do not terminate for the two examples used in this section. Since the tools such as UCLID [15)] or EUREKA [7)] are based on bounded model checking, they cannot obtain the complete result that the proposed method can.

7. Conclusion

In this paper, we proposed a term-height reduction technique to compute an over-approximate state set for an EUF state machine. Together with the state merging technique, our algorithm can show equivalences of modified C programs as well as two DSP designs, which the existent state-traversal based verification methods cannot verify.

If the invariant depends on the property of operators, however, checking the invariant can fail. To compensate for this, we will study some approach using the other logic together with EUF, or some other method [10)] to consider the property of functions when we check satisfiability for EUF formulas.

As other future works, it would be necessary to extend the algorithm to deal with arrays or memories of arbitrary length, or to check more general temporal properties which are not invariants. Furthermore, we would like to handle the cases in which the designated property does not hold. For this purpose, we would need to combine the other logic such as Boolean logic or linear arithmetic, together with EUF.

In our approach, a limit on term-height influences the degree of over-approximation. Since the term-height reduction implies discarding the least recent operations done to state variables, larger height means less of approximation. Finding an appropriate limit depends on the properties to be checked and also affects the performance. In this paper, this is not considered, because small heights are sufficient for the examples we handled. In order to determine the limit in general, we could increase the limit from some small value, while checking generated counter-examples are surely spurious by, for example, Boolean SAT. This also remains as a future work.

Acknowledgments We would like to thank anonymous reviewers for their helpful comments. This study was supported in part by funds from the Grant-in-Aid for Scientific Research (C) under Grant No.19500043, from Japan Society for the Promotion of Science (JSPS).

References

1) Clarke, E.M., Grumberg, O. and Peled, D.A.: *Model Checking*, The MIT Press (1999).
2) Burch, J.R. and Dill, D.L.: Automated verification of pipelined microprocessor control, *Computer-Aided Verification, LNCS 818*, pp.68–80 (1994).
3) Hamaguchi, K., Urushihara, H. and Kashiwabara, T.: Verifying Signal-Transition Consistency of High-Level Designs Based on Symbolic Simulation, *Formal Methods in Computer Aided Design, LNCS 1954*, pp.455–469 (2000).
4) Hojati, R., Isles, A., Kirkpatrick, D. and Brayton, R.K.: Verification using uninterpreted functions and finite instantiations, *Formal Methods in Computer-Aided Design, LNCS 1166*, pp.218–232 (1996).
5) Isles, A.J., Hojati, R. and Brayton, R.K.: Computing Reachable Control States of Systems Modeled with Uninterpreted Functions and Infinite Memory, *10th International Conference on Computer Aided Verification*, pp.256–267 (1998).
6) SAL: http://sal.csl.sri.com/.
7) Armando, A., Benerecetti, M., Carotenuto, D., Mantovani, J. and Spica, P.: The Eureka Tool for Software Model Checking, *22nd IEEE/ACM ASE Conference* (2007).
8) Corella, F., Zhou, Z., Song, X., Langevin, M. and Cerny, E.: Multiway Decision Graphs for Automated Hardware Verification, *Formal Methods in System Design*, Vol.10, Issue 1, pp.7–46 (1997).
9) Bryant, R.E., Lahiri, S.K. and Seshia, S.A.: Convergence Testing in Term-Level Bounded Model Checking, *Correct Hardware Design and Verification Methods, LNCS 2860*, pp.348–362 (2003).
10) Kozawa, H., Hamaguchi, K. and Kashiwabara, T.: Satisfiability Checking for Logic with Equality and Uninterpreted Functions under Equivalence Constraints, *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences*, pp.2778–2789 (2007).
11) Biere, A., Cimatti, A., Clarke, E.M. and Zhu, Y.: Symbolic Model Checking without BDDs, *Proc. Tools and Algorithms for the Analysis and Construction of Systems, LNCS 1579*, pp.193–207 (1999).
12) McMillan, K.L.: *Symbolic Model Checking*, Kluwer Academic Publishers (1993).
13) Holzmann, G.J.: The model checker SPIN, *IEEE Trans. Softw. Eng.*, Vol.23, No.5, pp.279–295 (1997).
14) Clarke, E.M., Kroening, D. and Lerda, F.: A Tool for Checking ANSI-C Programs, *Proc. Tools and Algorithms for the Analysis and Construction of Systems*, pp.168–176 (2004).
15) Bryant, R.E., German, S. and Velev, M.N.: Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions, *Computer-Aided Verification, LNCS 2404*, pp.78–92 (2002).
16) CVC3 page: http://www.cs.nyu.edu/acsys/cvc3/.
17) OPENCORES.ORG: http://www.opencores.org/.

(Received May 25, 2009)
(Revised September 3, 2009)
(Accepted October 31, 2009)
(Released February 15, 2010)

(Recommended by Associate Editor: *Qiang Zhu*)

Hiroaki Shimizu received a degree of master in information science and technology, Osaka University, Japan in 2008. He is currently with IIM Corp.

Kiyoharu Hamaguchi received the B.E., M.E. and Ph.D. degrees in information science from Kyoto University, Japan, in 1987, 1989 and 1993 respectively. In 1994, he joined the Department of Information Science, Kyoto University. He is currently with Graduate School of Information Science and Technology, Osaka University as Associate Professor. His current interests include formal verification and computer aided design.

Toshinobu Kashiwabara received the B.E., M.E. and Dr.Eng. degrees from Osaka University, Japan, in 1969, 1971 and 1974 respectively. He joined the faculty of Osaka University in 1974, and is currently a Professor at Graduate School of Information Science and Technology. His research interests include circuit layout and design of combinatorial algorithms. He is a member of IEEE.