IPSJ Transactions on System LSI Design Methodology Vol. 3 268–282 (Aug. 2010)

Regular Paper

Approximate Model Checking Using a Subset of First-order Logic

Kiyoharu Hamaguchi,[†1] Kazuya Masuda[†1] and Toshinobu Kashiwabara[†1]

In order to reduce the computational complexity of model checking, we can use a subset of first-order logic, called EUF, but the model checking problem using EUF is generally undecidable. In our previous work, we proposed a technique for checking invariant property for an over-approximate set of states including all the reachable states. In this paper, we extend this technique for handling not only invariants but also temporal properties written in computational tree logic with EUF extension. We show that model checking becomes possible for designs which are hard to handle without the proposed technique.

1. Introduction

The difficulty of model checking technique [1] lies in state explosion of hardware or software designs to be verified. Although there are many advantages of the technique such as exhaustiveness of error search or test-pattern-free verification, and we have now practical model checking tools such as SMV [12], SPIN [13], CBMC [14], and UCLID [15], state explosion still hinders application of the technique to large and complicated designs.

One of the approaches for tackling the state explosion is abstraction, where original designs are simplified so that they come to have a smaller number of states, but at the same time, the resulting abstracted designs reflect or contain all the behaviors of the original designs.

In our previous work [2], we proposed an algorithm using Quantifier-free First-order Logic with Equality and Uninterpreted Functions (EUF) for this purpose. In this framework, arithmetic operations are abstracted away by function or predicate symbols of the EUF logic. By handling them as symbols without considering specific meanings, we can avoid state explosion caused by the complexity of sequences of arithmetic operations.

For example, this EUF-based approach can be used for verification of digital signal processing units. Suppose that such a unit receives a command, and performs a specified arithmetic operation depending on the command. We can give a property such that, for a specific command and input data, output data is always correctly computed after some cycles. Even if the computation includes complicated arithmetic operations which are intractable at Boolean level in terms of model checking, verification can succeed in our framework, because of abstraction.

This approach also enables design or property descriptions which include some abstract function call. If we try to have the function call at Boolean level, we need to have the concrete description of the function call, which would be very costly in general. In the EUF-based approach, we do not have to have such detailed descriptions. Instead, we use a function symbol abstractly. For example, we can write down a property such that the result of a function call is "true" inside some loop, then the execution eventually exits the loop. If the design has a corresponding conditional branch, and if the design and property use the same predicate and function symbols to represent branch conditions, verification can be successful without specifying the details of the function call.

When we consider state machines defined by the EUF logic, the "unbounded" model checking problem has been known to be undecidable [4]. In our approach [2], we perform over-approximate state traversal instead of precise state traversal, while guaranteeing decidability. We have shown that this approach is effective to some designs with fairly complicated arithmetic operations, such as an zero-point calculation by Bisection method, or an ADPCM encoding algorithm. These examples cannot be handled by any EUF-based or Boolean-based model checking techniques.

The above work, however, can handle only invariant properties, that is, properties to be valid at any state reachable from initial states. We cannot specify temporal properties such as "if some condition holds at a state, then some event happens eventually".

The contribution of this paper is an extension of our approach [2] for handling

†1 Osaka University

more general temporal properties. We define EUF-based computational tree logic (CTL), called EUF-CTL, and show a model checking algorithm for the logic. More precisely, this EUF-CTL is based on so called ACTL [1], which does not allow existential path quantifier **E** but only universal path quantifier **A** for temporal expressions. This restriction is necessary to guarantee the conservativeness of the verification result, when we handle over-approximation of reachable state sets [1].

The algorithm is based on the classical CTL model checking algorithm, but its straightforward application to our approach does not work well. Each state in a state transition graph generated by our algorithm [2], represent, in actuality, a set of multiple states. And thus, validity of atomic formulas at each state cannot be determined generally. Our algorithm in this paper resolves this problem and enables model checking for temporal properties. We implemented the algorithm and applied it to some designs.

The remainder of this paper is organized as follows. Firstly, we discuss related works on this paper, and we give the definition of EUF, its state machine. Secondly, we briefly explain the procedure of generating state transition graphs we proposed before [2]. Then, we define the syntax and the semantics of EUF-based CTL (EUF-CTL). We show how to modify the previously proposed algorithm for enabling model checking. Finally, we show some experimental results.

2. Related Works and Comparison

There have been proposed several approaches using the EUF logic for verifying state transition systems with complex datapaths.

Burch and Dill [3] applied the EUF logic for proving the equivalence of pipelined and non-pipelined microprocessors. Their approach reduces the problem to validity checking of the EUF logic in which equivalence within a given finite number of cycles is focused. The recent tools such as Ref. 15) or SAL [6] can handle EUF-based descriptions, but they are based on bounded model checking, that is, they check the designs up to a given finite number of cycles.

As for unbounded model checking with more powerful logic, Cyrluk and Narendran [7] proposed ground temporal logic which is an extension of linear-temporal logic (LTL) for verification. They showed, generally, the model checking problem using this logic is undecidable. They also showed a decidable fragment of the logic, but it is a very small class. For example, it does not allow "until" operator, which can be handled in the approach of this paper.

Bohn, et al. [8] proposed first-order CTL and its unbounded model checking procedure. Xu, et al. [10] also showed first-order LTL, which is weaker than ground temporal logic in terms of expressiveness, and its unbounded model checking procedure using multiway decision diagrams [9]. Both of these approaches cannot guarantee termination of the procedures, unlike our approach.

The EUF-CTL we handle in this paper is weaker than those in the above. For example, the past or future value of a data variable represented as an EUF variable, cannot be referred to from a present state, that is, it is not allowed to use an expression such as $v = \mathbf{X}u$, meaning that the value of v is equivalent to that of u at the next cycle. It remains an open question whether we can enhance the expressiveness of temporal logic, without losing decidability based on over-approximation.

3. EUF and State Machine

This section defines the syntax and the semantics of the EUF logic and state machines using that logic. Datapaths in designs are abstracted by this logic.

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*. The syntax is shown in the below. The number of arguments is called *arity*, which is finite for any function or predicate.

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

The term-height of term t, denoted by *term-height*(t), is defined as follows:

term-height(t) =

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

where MAX is the function which returns the maximum from its arguments and f is a function symbol. This notion is used for approximating state transition graphs. For example, for variables `c1` and `c2` and function symbols `f` and `g`, 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 $t = \text{ITE}(\alpha, t_1, t_2)$ 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 $\mathcal{D}^k \to \mathcal{D}$ and $\mathcal{D}^k \to \{true, false\}$, 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 either an equation, a Boolean variable or negated forms of them.

```
x := 1;                          x := c1;
if(y > 1)                        if(p(y,c1))
      y := (x+y) * 2;      ⟹          y := g(f(x,y),c2);
else                             else
      y := (x+y) / 2;                  y := h(f(x,y),c2);
```

Fig. 1 Example: a high-level description.

```
b1' := ITE(b1=false, true, b1)
t1' := ITE(b1=false, c1, t1)
t2' := ITE(b1=true,
       ITE(p(t2,c1)=true, g(f(t1,t2),c2), h(f(t1,t2),c2)), t2)
```

Fig. 2 Example: transition functions.

3.3 EUF State Machine

Example 1 We give an example first. **Figure 1** shows a C-like high-level description of a design. The left C-like code can be translated to the right description in the figure. In this example, the inequality is represented by predicate p, and addition, multiplication and division are represented by function symbols f, g and h respectively. The constants 1 and 2 are represented by variables $c1$ and $c2$ respectively.

Figure 2 shows its transition functions. Term state variables $t1$ and $t2$ are introduced to represent x and y, and $b1$ is used as a program counter, whose initial value is false. □

In order to obtain EUF state machines, in this paper, we do not assume any sophisticated procedure for predicate abstraction. We assume that we are given high-level descriptions as in Fig. 1, and that we can transform them into EUF-based transition functions. Basically, each arithmetic operators are replaced with function symbols, and equality or inequality are replaced with predicates.

Precisely, an EUF state machine is defined by a set of transition functions. 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 $b'_i := F_i$ $(1 \leq i \leq m)$ and $t'_j := T_j$ $(1 \leq j \leq n)$, 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 $(i \neq j)$.

We call the following formula *transition relation*:

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

In our previous work [2], the behavior of an EUF state is determined by a sequence of interpretations, each of which gives an interpretation for variables, functions and predicates at each step. In this paper, however, we use an *interpretation tree*, in order to have correspondence to the CTL semantics with EUF extension. An interpretation tree can be regarded as a collective representation of interpretation sequences.

In the following, we formulate an interpretation tree $\tilde{\sigma}_M = (V, E)$ for an EUF state machine M. V is a set of interpretations of the EUF logic, each of which specifies the values for state variables or input variables at each step along the behavior of M. The interpretations for function or predicate symbols are fixed throughout all the nodes for each interpretation tree. E is the set of transitions between such interpretations, which correspond to those between states in M. Details are shown in the below.

Note that an interpretation tree generally contains an infinite number of nodes, and that each node can have an infinite number of children.

(M-1) The interpretations of term variables, Boolean variables, function symbols and predicate symbols are the same at every node, except for input variables, variables assigned to state variables at initial states and next state variables. In other words, for each non-input variable c_j, each non-input Boolean variable a_j, each function symbol f_j and each predicate symbol p_j, $\sigma(c_j) = \sigma^0(c_j)$, $\sigma(a_j) = \sigma^0(a_j)$, $\sigma(f_j) = \sigma^0(f_j)$ and $\sigma(p_j) = \sigma^0(p_j)$ for each node $\sigma \in V$, where σ^0 is an interpretation for the root node of the interpretation tree.

(M-2) As for each input term variable c_j and for each input Boolean variable a_j at node σ, we introduce new variable names c_j^σ and a_j^σ for distinction. $\sigma(c_j^\sigma)$ and $\sigma(a_j^\sigma)$ are given arbitrary at node σ. For descendant node σ_c of σ, $\sigma_c(c'_j) = \sigma(c'_j)$ $(k = i, i+1, \ldots)$, and $\sigma_c(a'_j) = \sigma(a'_j)$ $(k = i, i+1, \ldots)$. The interpretations of these variables for ancestors of σ are not defined.

(M-3) For an initial state in which each term state variable t_j and a Boolean state variable b_j are mapped to variable c_j and a_j, $\sigma^0(t_j) = \sigma^0(c_j)$ and $\sigma^0(b_j) = \sigma^0(a_j)$.

(M-4) $(\sigma, \sigma') \in E$ if and only if, for transition functions $b'_j := F_j$ and $t'_j := T_j$, $\sigma'(b_j) = \sigma(F_j)$ and $\sigma'(t_j) = \sigma(T_j)$.

We call an interpretation tree which satisfies the above conditions a *normal interpretation tree*. In this paper, we assume that interpretation trees are all normal.

The model checking problem for an EUF-based state machine M and an EUF-CTL formula f is defined over interpretation trees of M. Basically, it checks whether, at the root of all possible interpretation trees for M, f holds or not. The precise definition will be given later in Section 5.1.

4. Generating State Transition Graphs

In this section, we briefly explain the algorithm for generating state transition graphs based on our previous work [2].

We show, firstly, a simple state traversal procedure. This procedure does not terminate in general, because generated state transition graphs can have an infinite number of states. Later, we show techniques called term-height reduction and state merging, which generates an over-approximate state transition graphs with a finite number of states.

The infinite state set may seem to be strange, because the original description assumes Boolean interpretations and, as a result, the state space is originally finite. On the other hand, the EUF-based (non-approximate) state traversal can produce infinite state space. The reason of this is that equivalent concrete states under Boolean interpretations can be represented by different forms of

term vectors.

4.1 A Simple State Traversal Procedure

Each state s is composed of the following 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.

We can consider more than one interpretations which satisfy C. In other words, a "state" in the graph can be regarded as a representation of *a set of states* instead of a single state. However, in the rest of this paper, we call the set of states simply as a state.

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

Example 2 **Figure 3** shows a part of state traversal for the example shown in Fig. 2. Each box corresponds to a state. The binary values for $b1$ in the state vectors are omitted in the figure for brevity. The initial values for state term variables $t1$ and $t2$ are $d0$ and $c0$ respectively, where $d0$ and $c0$ are EUF variables. □

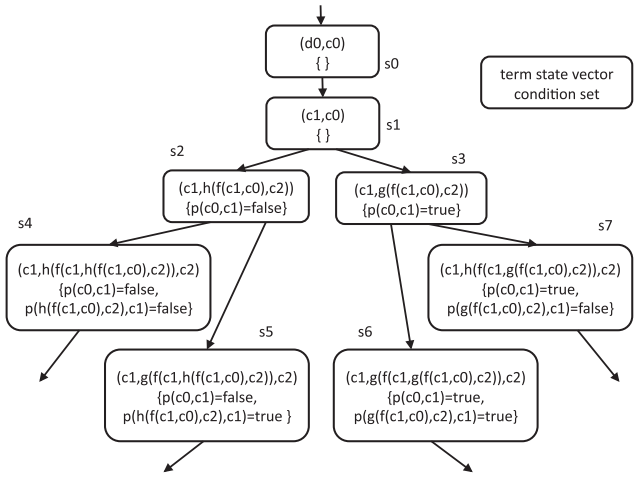

Fig. 3 Simple state traversal.

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, t is assigned to the term state variable t_l at the next state.

(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. State s is not generated, if its condition set C' is unsatisfiable.

4.2 Approximate State Traversal

An approximate state graph is generated using the following techniques, that is, term-height reduction and state merging. The details have been given in the other literature [2].

4.2.1 Term-height Reduction

In order to control term-height among terms in the state traversal procedure, we give parameter $maxh$ as an input for the algorithm.

In the procedure, state vector $\overrightarrow{v}$ or condition set C can contain some term whose height is larger than a given integer $maxh$ ($maxh \geq 0$). Then, we replace the subterm of the term by a new variable so that the term-height is smaller than or equal to $maxh$.

We call this operation term-height reduction. For example, suppose that $maxh = 1$ and we have term $g(g(c1, f(c2)), f(c1))$. Firstly, the term-height is reduced by one by replacing $f(c2)$ by new variable $c3$. Then, we have $g(g(c1, c3), f(c1))$. Furthermore, we replace $g(c1, c3)$ and $f(c1)$ by $c4$ and $c5$ to have $g(c4, c5)$ with term-height 1. We record all these reductions such as $f(c2) \rightarrow c3$, $g(c1, c3) \rightarrow c4$ and $f(c1) \rightarrow c5$ as conversion rules, and apply them to the terms which may appear later in the procedure.

This operation reduces constraints or relations among terms in a state. This implies we can have more of possible interpretations for each state. In other words, term-height reduction causes over-approximation to the state transition graph.

4.2.2 State Merging

As we discussed earlier, a state in a state transition graph represents a set of states under more than one interpretations. State merging of a state by other state, in actuality, is based on inclusion relation between the two sets of states. If a newly generated state in the state traversal can be merged into some other state among the already generated ones, then the new state is merged to the old state. The following procedure is basically the same as that by Isles, et al. [5].

Inclusion relation for two states is basically determined by checking the match of each element in their state vectors, and by checking logical implication among their reachablity condition sets. However, literal match of the state vectors does not work well, because new variables are introduced in the state traversal procedure. Thus, state vector match is checked under some renaming of variables.

For example, suppose that we have state vectors $t = (f(c3, c2), c2)$ and $t' = (f(c1, c2), c2)$. By considering renaming $\{c3 \rightarrow d1, c2 \rightarrow d2\}$ for t and $\{c1 \rightarrow d1, c2 \rightarrow d2\}$ for t', we can match the two state vectors. State inclusion is defined precisely as follows. If state s' can be merged to state s, then all transition edges to s' are redirected to s and s' is deleted.

More precisely, state inclusion relation is defined 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 1 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

Example 3 **Figure 4** shows term-height reduction and state merging for the state transition graph in Fig. 3. We assume $maxh = 2$. The result of approximation is shown in **Fig. 5**. □

4.2.3 Overall Algorithm

Suppose that we have a set of candidate reachable states CS which contains

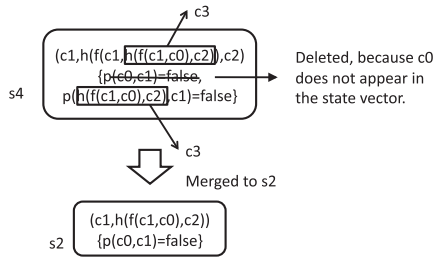

Fig. 4 State merging.

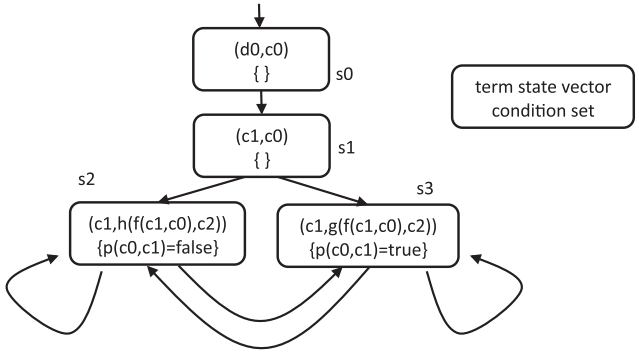

Fig. 5 Approximate state graph.

only the initial state at first. We also have a set of reachable states RS, which is initially empty. We pick up a state cs from CS, and check its reachability by satisfiability checking of cs's reachable condition set, and check if cs can be merged into some other state in the reachable set. If it is reachable and cannot be merged, then cs is inserted to RS. By using the transition relation, the set of next candidate states from cs, that is, NS is computed. After applying term-height reduction to each state in NS, the next candidate states are inserted into CS. The above process is repeated while CS is not empty. When CS becomes empty, we can obtain all reachable states in RS.

4.3 State Transition Graph and Interpretation Tree

We define an interpretation tree for a non-approximate or approximate state transition graph. Since non-approximate state transition graphs are generated from the transition relation of an EUF state machine, interpretation trees for the EUF state machine, as defined in Section 3.3 and those for its state transition graph should be the same. Approximate state transition graphs, however, do not have direct correspondence to the EUF state machine. Thus, we give a slightly different definition as interpretation tree for state transition graphs, which is common for non-approximate or approximate state transition graphs.

Suppose that we have a state transition graph G for an EUF machine M. Each node σ of an interpretation tree $\tilde{\sigma}_G$ corresponds to a state in G. We define a mapping function $\eta_S : \Sigma_G \to S$, where Σ_G is a set of interpretations and S is a set of states of G. Note that more than one nodes in $\tilde{\sigma}_G$ can be mapped to a state in G.

Interpretation tree $\tilde{\sigma}_G$ for G must satisfy (M-1)–(M-3) for interpretation tree for M, in addition to the following (G-4'), (G-5) and (G-6). Condition (G-6) is applied only to approximate state transition graphs. In order to distinguish properties for state transition graph G, we use (G-1)–(G-3) instead of (M-1)–(M-3), respectively.

(G-4') (σ, σ') is an edge in $\tilde{\sigma}_G$ if and only if $(\eta_S(\sigma), \eta_S(\sigma'))$ is an edge in G.

(G-5) For state $s = ((\overrightarrow{t}, \overrightarrow{b}), C) \in S$, $\sigma(t_j) = \sigma(\overrightarrow{t}[j])$, $\sigma(b_j) = \sigma(\overrightarrow{b}[j])$ and $\sigma(\bigwedge_{c \in C} c) = true$, where t_j is the j-th term state variable and $\overrightarrow{t}[j]$ is the j-th element of vector $\overrightarrow{t}$.

(G-6) Suppose that s is merged to s', and that variable v in s corresponds to v' in s' through the mapping function. Then, $\sigma_{s'}(v') = \sigma_s(v)$.

5. EUF-CTL and Model Checking

5.1 EUF-CTL

The logic we use is based on ACTL. ACTL does not have existential path operator **E**. In order to avoid expressing **E** by complementation, the negation operator is used immediately before atomic formulas only. This means that, in the logic defined in the below, the negation operator can be used immediately before (non-temporal) EUF formulas only.

We assume that variables are those used in the descriptions for an EUF state machine. An EUF-CTL formula is defined recursively as follows: 1) if p is a (non-temporal) EUF formula, p is an EUF-CTL formula, 2) if f and g are EUF-CTL

formulas, $f \vee g$ and $f \wedge g$ are also EUF-CTL formulas. 3) if f is a (non-temporal) EUF formula and g is an EUF-CTL formula, $f \rightarrow g$ is also an EUF-CTL formula, 4) if f and g are EUF-CTL formulas, then $\mathbf{AX}f$, $\mathbf{AF}f$, $\mathbf{AG}f$, $\mathbf{A}(f\mathbf{U}g)$ are also EUF-CTL formulas.

The semantics is defined for an interpretation tree $\tilde{\sigma}$. Let s be a node in $\tilde{\sigma}$. $\pi = s_0, s_1, s_2 \ldots$ represents an infinite path in the tree. π^i is the suffix of π starting from s_i. $\tilde{\sigma}, s \models f$ means f is true at node s in $\tilde{\sigma}$, and $\tilde{\sigma}, \pi \models f$ means f is true along path π in $\tilde{\sigma}$.

Then, the truth value of a given EUF-CTL formula is defined as follows:

(1) $\tilde{\sigma}_M, s \models p \Leftrightarrow s(p) = true$
(2) $\tilde{\sigma}_M, s \models f_1 \vee f_2 \Leftrightarrow \tilde{\sigma}_M, s \models f_1$ or $\tilde{\sigma}_M, s \models f_2$
(3) $\tilde{\sigma}_M, s \models f_1 \wedge f_2 \Leftrightarrow \tilde{\sigma}_M, s \models f_1$ and $\tilde{\sigma}_M, s \models f_2$
(4) $\tilde{\sigma}_M, s \models f_1 \rightarrow f_2 \Leftrightarrow \tilde{\sigma}_M, s \models \neg f_1$ or $\tilde{\sigma}_M, s \models f_2$
(5) $\tilde{\sigma}_M, s \models \mathbf{AX}f_1 \Leftrightarrow$ for all the paths $\pi = s_0, s_1, \ldots$ from s, $\tilde{\sigma}_M, s_1 \models f_1$.
(6) $\tilde{\sigma}_M, s \models \mathbf{AF}f_1 \Leftrightarrow$ for all the paths $\pi = s_0, s_1, \ldots$ from s, there exists $k \geq 0$ such that $\tilde{\sigma}_M, s_k \models f_1$.
(7) $\tilde{\sigma}_M, s \models \mathbf{AG}f_1 \Leftrightarrow$ for all the paths $\pi = s_0, s_1, \ldots$ from s, and for all $k \geq 0$, $\tilde{\sigma}_M, s_k \models f_1$.
(8) $\tilde{\sigma}_M, s \models \mathbf{A}(f_1\mathbf{U}f_2)$ for all the paths $\pi = s_0, s_1, \ldots$ from s, there exists $k \geq 0$ such that $\tilde{\sigma}_M, s_k \models f_2$, and for all j such that $0 \leq j \leq k$, $\tilde{\sigma}_M, s_j \models f_1$.

The model checking problem for an EUF-CTL formula f and an EUF state machine M, is to check, for an arbitrary interpretation tree $\tilde{\sigma}_M$ for M with root node r, whether $\tilde{\sigma}_M, r \models f$ holds or not.

5.2 Extended Approximate State Transition Graph

Each state in a state transition graph represents a set of states under more than one interpretations. As a result, the truth value of an EUF-formula at a state in the state transition graph cannot be determined in general, because it depends on an interpretation chosen from all possible interpretations. This causes a difficulty when we try to perform a standard model checking algorithm for CTL on an approximate state transition graph, because the algorithm assumes that the truth values of each atomic formula appeared in a given EUF-CTL formula are specifically true or false at each state. In order to handle this problem, we split each state to multiple states so that the truth value of each atomic formula is determined at each state.

For a given EUF-CTL formula f, we define set Z, each of whose element is an equation appeared in f. If f contains r equations, then the size of Z is r. For $z_i \in Z$ $(i = 1, 2, \ldots, r)$, we define $z_i^{p_i}$ $(p_i \in \{0, 1\})$, where $z_i^1 = z_i$ and $z_i^0 = \neg z_i$. For a binary vector $\overrightarrow{p} = (p_1, p_2, \ldots, p_r)$ of length r, we define as follows:

$$z^{\overrightarrow{p}} = \bigwedge_{i=1,2,\ldots,r} z^{p_i}$$

Extension is done by splitting a newly generated state s in the graph generation procedure, to 2^r states by augmenting the original state with $replace(z^{\overrightarrow{p}})$ for $\overrightarrow{p} \in \{0, 1\}^r$, where $replace(z^{\overrightarrow{p}})$ means a formula obtained by replacing each of the state variables in z by its corresponding element in the state vector of s. The condition $replace(z^{\overrightarrow{p}})$ is added to reachable condition set C of the state. For example, if $Z = \{f(x) = y, g(x, y) = z\}$ and term state vector $(x, y, z) = (f(c0), c1, c2)$, then we have four states having, in their condition sets, $(f(f(c0)) = c1) \wedge (g(f(c0), c1) = c2)$, $(f(f(c0)) = c1) \wedge \neg(g(f(c0), c1) = c2)$, $\neg(f(f(c0)) = c1) \wedge (g(f(c0), c1) = c2)$ and $\neg(f(f(c0)) = c1) \wedge \neg(g((c0), c1) = c2)$, respectively.

The initial state must be split similarly before starting the graph generation procedure.

We perform this extension *after* term-height reduction and state merging are done. In other words, the newly introduced constraints in the condition sets are not affected by term-height reduction or state merging. This does not change the set of possible interpretation trees for resulting approximate state transition graphs, because this procedure simply split each state. The correctness is shown through theorems later in this section.

We call the resulting graph an extended approximate state transition graph. Interpretation trees for an extended approximate state transition graph can also be defined as in Section 4.3.

Example 4 We assume property $\mathbf{AF}(t2 = g(f(t1, c0), c2))$ for model checking and $maxh = 2$. **Figure 6** shows the extended graph for Fig. 3. For example, $c0 = g(f(c1, c0), c2)$ at state $s11$ comes from $t2 = g(f(t1, c0), c2)$ replaced by term state vector $(t1, t2) = (c1, c0)$ at $s11$. The boxes surrounded by dashed lines $s310$ and $s300$ are not generated in actuality, because the condition sets are

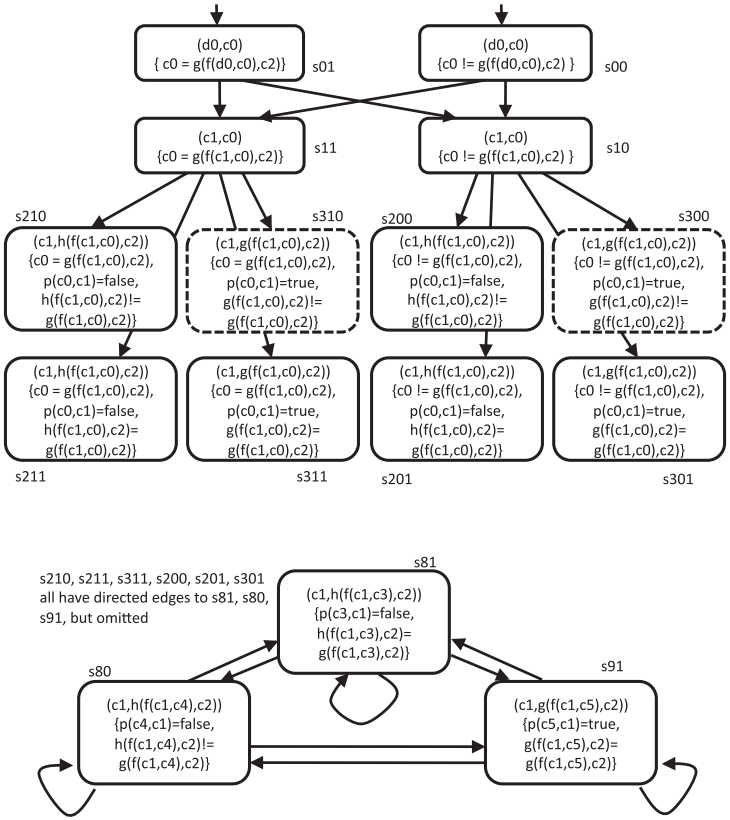

Fig. 6 An extended approximate state transition graph.

unsatisfiable. □

5.3 EUF-CTL Model Checking

The model checking algorithm is basically the same as the standard CTL model checking algorithm. For a given EUF-CTL formula f, whether each subformula in f holds or not at each state is checked in a bottom-up manner. More precisely, this algorithm labels subformula g_f of f to state s if and only if g_f is true at root node σ_r of any interpretation tree such that $\eta_S(\sigma_r) = s$.

As for an atomic formula z, labeling z is performed by checking whether $replace(z)$ is true at C, that is, by checking whether the formula each of which state variable in z is replaced by its corresponding element in the state vector of state s, is included in C or not. This can be done easily because of the way of augmenting extended approximate state transition graphs.

As for a non-atomic formula f, if we assume that subformulas in f have been processed, that is, whether they are valid or not has been labeled at each state, then we can determine whether f is valid or not at each state using those labels. We omit the details because the rest of the algorithm is the same as the standard CTL model checking algorithm [1].

The correctness of the algorithm is guaranteed throughout the following lemmas and theorem. The proofs are given in Appendix.

Lemma 1 The set of interpretation trees for an EUF state machine M is equivalent to the set of interpretation trees for the non-approximate state transition graph G generated for M. □

Lemma 2 The set of interpretation trees for the non-approximate state transition graph G is equivalent to the set of interpretation trees for the non-approximate extended state transition graph G^{ex}. □

Lemma 3 Any interpretation tree for a non-approximate extended or non-extended state transition graph is a sub-tree of an interpretation tree for an approximate extended or non-extended state transition graph. □

Theorem 1 Suppose that we have generated an extended approximate state transition graph G_a^{ex} for an EUF state machine M and that we perform the model checking algorithm. If an EUF-CTL formula f is labeled at state s in G_a^{ex}, then f is true at root node σ_r of any interpretation tree for M an EUF machine M such that $\eta_S(\sigma_r) = s$. □

5.4 Counter-Examples

By using some popular counter-example generation algorithm for CTL [1], we can obtain a state sequence which violate the property in the extended approximate state transition graph. Because of approximation in the state traversal and also of abstraction by function symbols of EUF, this state sequence can be spurious.

We can check the feasibility of this state sequence as follows. Firstly, we generate a non-approximate state sequence corresponding to the counter-example

state sequence, by tracing the same transition conditions from the initial state, without approximation, that is, without term-height reduction. Note that the transition conditions are all inserted to reachability condition sets in states. Secondly, we check the satisfiability of the reachability condition set at each state, using some Boolean SAT solver under Boolean interpretation for EUF functions or predicates. If it is satisfiable, the state sequence is a feasible, non-spurious counter-example. The computational cost could be large, though, because of Boolean interpretation.

6. Experimental Results

We implemented our algorithm in the C++ language and performed some experiments with Intel Core2 Duo T8300 2.40 GHz of 3 GB Memory under Windows XP. We used Yices [16] as an EUF SAT solver.

Only a small number of common benchmarks in form of EUF state machines are available [6),15)], as far as we know. Those benchmarks have features our method cannot handle presently, such as unbounded memories or list structures, because they are intended for bounded model checking. We used, in this paper, designs of a FIR filter circuit and a simple C program for Bisection Method, and applied our algorithm. The counter-example generation procedure has not been implemented yet.

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 FIR Filter Circuit

We show the example of model checking to FIR filter (Finite Impulse Response Filter) design description. N-taps FIR filter is a digital filter that computes the output using signal values of the last N cycles. It does not depend on the past output signals.

We consider 3-taps FIR filter. 3-taps FIR filter circuit calculates the following output Y at every cycle, where X_3 is a present input, X_{3-i} $(1 \leq i \leq 2)$ are inputs of i cycles before, and H_i $(0 \leq i \leq 2)$ is a constant.

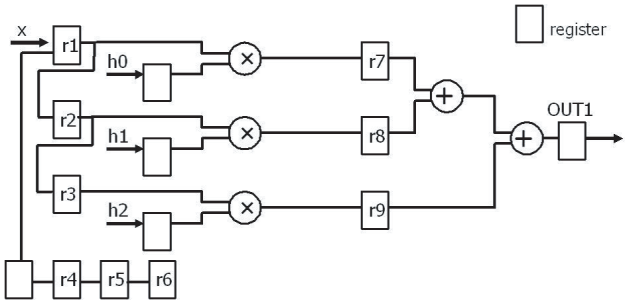

Fig. 7 Block chart of 3-taps FIR filter.

Table 1 Experimental results for FIR filter.

maxh	maximum states	graph generation total (s)	model checking total (s)	total times (s)
3	35	3.374	0.109	3.483

$$Y = \sum_{i=0}^{2} X_{3-i} H_i$$

Our FIR filter design is as shown in **Fig. 7**. We compared the output $OUT1$ with that calculated by the equation $Y = \sum_{i=0}^{2} X_{3-i} H_i$, using $r4$, $r5$ and $r6$ in the pipeline registers, and checked those outputs were equivalent. We store the input sequence in the pipeline. We gave the property $(\mathbf{AG}(\mathbf{AX}(\mathbf{AX}(\mathbf{AX}(\mathbf{AX}(\mathbf{AX}(\mathbf{AX}(OUT1 = f0(f0(f1(r4, h0), (f1(r5, h1))), (f1(r6, h2)))))))))))$, which checks the equivalence 5 cycles after the circuit starts working. $r4$, $r5$, and $r6$ are the term variables, and they correspond to the register numbers in Fig. 7. We prepared an extra register to match the timing of output $OUT1$ and the evaluation of $f0(f0(f1(r4, h0), (f1(r5, h1))), (f1(r6, h2)))$.

We show the experimental results in **Table 1**. We changed the parameter value $maxh$ increasingly from 0 one by one, until the property held. The first column shows the value of $maxh$, when the property held. The second column shows that the maximum number of states in the generated state transition graphs until the property held. The third column shows the total times for generating state transition graphs, and the fourth column shows the total times for model

checking until the property held. The fifth column shows the total time of the third and fourth column.

6.2 Bisection Method

Bisection method is an algorithm for solving an equation numerically. The number of execution of the while body depends on input value, and thus is indeterminate.

We gave some properties as follows as examples. We show the corresponding pseudo code of the bisect algorithm in **Fig. 8**. Labels (a)–(d) in Fig. 8 correspond to the conditions of the properties in the below. The part enclosed with the dotted line is a loop part. $n0$ is introduced as a program location counter, which points the location of current execution. Here, $n0 = 39$ means that the execution proceeded to the beginning of the loop, and $n0 = 52$ means that the execution exits this procedure. The meaning of the property of (a) is that, if the three conditions in property (a) holds, the program cannot reach the beginning of the loop. The meaning of the property of (b)–(d) is that, if either of the conditions in (b)–(d) holds at the beginning of the loop, the program exits.

(a) ((((f0(t0) = c0) ∨ (f0(t1) = c0)) ∨ (p1(f0(t0), f0(t1)) = true)) ⇒ (**AG**(n0 ≠ 39))))

(b) (**AG**(((n0 = 39) ∧ (p0(f2(f3(f1(t0,t1),c2),t0), t3) = true)) ⇒ (**AF**((n0 = 52) ∧ (b0 = true)))))

(c) (**AG**(((n0 = 39) ∧ (p0(f2(t1,f3(f1(t0,t1),c2)), t3) = true)) ⇒ (**AF**((n0 = 52) ∧ (b0 = true)))))

(d) (**AG**(((n0 = 39) ∧ (f0(f3(f1(t0,t1),c2)) = c0)) ⇒ (**AF**((n0 = 52) ∧ (b0 = true)))))

We show the experimental results in **Table 2**. The first column shows the checked properties. The other columns are the same as in Table 1. This experiments required 3 Mbyte for running.

6.3 Consideration

In this section, we consider what kind of designs can be handled with this method. **Table 3** shows the detailed results on the runtimes for Bisection method algorithm. "sat total" means total runtimes by the SAT solver, and "sat inclusion check" means runtimes by the SAT solver for state inclusion check explained in Section 4.2.2.

```
double bisect(double a, double b, double tolerance,
                    double (*f)(double), int error, &result) {
  double c, fa, fb, fc;

  error = 0;
  if (tolerance < 0)
    tolerance = 0;
  fa = f(a);
  if (fa == 0){                                   (a)
    return a;
  fb = f(b);
  if (fb == 0)
    return b;
  if (samesign (fa, fb)){
    error = 1;
    exit;
  }
  while (true) {
    c = (a + b) / 2;
    if (c – a < tolerance) break;                 (b)
    if (c – a = tolerance) break;
    if (b – c < tolerance) break;                 (c)
    if (b – c = tolerance) break;
    fc = f(c);
    if (fc == 0)                                  (d)
      return c;
    if (samesign (fc, fa)) {
      a = c;
      fa = fc;
    } else {
      b = c;
      fb = fc;
    }
  }
  return c;
}
```

Fig. 8 Pseudo code of bisection algorithm.

Table 2 Experimental results for bisection method algorithm.

property	maxh	maximum states	graph generation total (s)	model checking total (s)	total times (s)
(a)	1	277	338.947	1.173	340.120
(b)	3	92	192.273	1.739	194.012
(c)	3	92	196.639	1.530	198.169
(d)	3	96	203.800	1.598	205.398

While the required memory was up to 3 Mbyte, approximately 70%–80% in the runtime was used for satisfiability checking in state graph generation. Thus, the runtimes are bottleneck for our algorithm. State inclusion check in state

Table 3 Detailed runtime results for biesection method algorithm.

property	graph generation total (s)	sat total (s)	sat inclusion check (s)
(a)	338.947	273.340	6.304
(b)	192.273	158.719	8.492
(c)	196.639	161.667	8.714
(d)	203.800	167.384	8.997

merging checks all state pairs, and thus, the number of comparisons is quadratic to the number of states in the worst case. The runtimes by the SAT solver for checking state inclusion are, however, less than 5%, because state merging calls the SAT solver only when term state vectors of two states can match syntactically. These imply that the dominant factor in the runtimes is the rest of "sat total", that is, reachability checking at each state, which is performed once at each state generation. Based on this, supposing that runtime increases linearly to the number of states, we could estimate that we can handle roughly up to 10,000 states in practical time.

If we think arithmetic core modules such as digital cosine transform in digital signal processing units, we can expect that the number of control states is relatively small, e.g., up to 100 states, and the number of different term state vectors for each control state is limited, e.g., up to 100 distinct vectors, because of term-height reduction. Then, such modules would fall on the range the proposed algorithm can cover.

Next, we consider maxh. Unfortunately, it looks difficult to determine an appropriate maxh beforehand. A term assigned to a term state variable can be regarded as a certain kind of history data storing the most recently performed operations to the term state variable. If the stored data in terms are large enough, we can check the property under the EUF-based abstraction for function or predicate symbols.

In reality, there are some factors which make the behavior of the algorithm more complicated. The term-height reduction procedure stores every replacement of each term by a new variable, and uses the variable so that the same terms are replaced with the same variables. This mechanism tends to shorten the necessary term-height. Furthermore, the necessary term-height depends on the formulas in the EUF-CTL temporal property.

In typical digital signal processing units, input data go through multiple operations, and go out as output data. In this case, the number of operations to be performed can be the necessary term-height, that is, maxh, which could be estimated from the analysis of design. Because of the above factors, however, maxh could be smaller than this rough estimation. Because of lack of a good strategy, here we increase the maxh one by one from 0.

6.4 Comparison with Other Works

Previous works in the literature such as Ref. 4), 5), 9) handled the same problem, but they do not guarantee the termination of the procedure. Since the tools such as UCLID [15] or EUREKA [11] are based on bounded model checking, they cannot obtain the complete result that the proposed method can. To our best knowledge, only our proposed algorithm can handle unbounded model checking for the EUF state machines we used in the experiments.

7. Conclusion

In this paper, we define a temporal logic based on the logic EUF, that is, EUF-CTL and a model checking problem using the logic for an EUF state machine. Since the original problem is undecidable, our algorithm guarantees its termination by over-approximating the state space to be explored. We implemented the algorithm and showed that our approach can handle some examples for which the other model checking based techniques do not work well, because of the computational complexity caused by complex arithmetic operations in the designs.

The limitation of the EUF-based approach is not to consider the semantics of the function or predicate symbols. We cannot refer to the semantics of the function symbols. If "4-bit right-shift" operation in the property is implemented as two consecutive "2-bit right-shift" operations in the design, then these are not identified as equivalent.

Furthermore, EUF cannot handle the commutative law or the associative law over arithmetic operations, that is, $f(x, y)$ is always regarded as different from $f(y, x)$, even if f obeys the commutative law. We are considering combined use of multiple logics for handling such cases, in which we interpret only a small

number of function symbols with less abstract logic such as Boolean logic, to obtain less conservative results.

An extension for handling different logics such as linear-time temporal logic remains as a future topic. It is an open question if a standard tableau construction technique [1)] works properly or not under the approximation scheme in this paper. As another extension, we can consider a temporal expression related not to formulas but to variables, such as $t1 = (\mathbf{XX}t2)$, which means that term $t1$ is equivalent to term $t2$ of 2 cycles later.

Acknowledgments 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) Shimizu, H., Hamaguchi, K. and Kashiwabara, T.: Approximate Invariant Property Checking Using Term-Height Reduction for a Subset of First-Order Logic, *6th International Conference on Automated Technology for Verification and Analysis*, *LNCS 5311*, pp.318–331 (2008).
3) Burch, J.R. and Dill, D.L.: Automated verification of pipelined microprocessor control, *Computer-Aided Verification*, *LNCS 818*, pp.68–80 (1994).
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) Cyrluk, D. and Narendran, P.: Ground Temporal Logic: A Logic for Hardware Verification, *Computer-Aided Verification*, *LNCS 818*, pp.247–259 (1994).
8) Bohn, J., Damm, W., Grumberg, O., Hungar, H. and Laster, K.: First-Order-CTL Model Checking, *Foundations of Software Technology and Theoretical Computer Science*, *LNCS 1530*, pp.283–294 (1998).
9) 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, No.1, pp.7–46 (1997).
10) Xu, Y., Song, X., Cerny, E. and Mohamed, O.A.: Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs, *The Computer Journal*, Vol.47, No.1, pp.71–84 (2004).
11) Armando, A., Benerecetti, M., Carotenuto, D., Mantovani, J. and Spica, P.: The Eureka Tool for Software Model Checking, *22nd IEEE/ACM ASE Conference* (2007).
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) Yices: An SMT Solver, http://yices.csl.sri.com/

Appendix

A.1 Proof of Lemma 1

Suppose that we have an interpretation tree $\tilde{\sigma}_M$ for an EUF state machine M. We consider an interpretation tree $\tilde{\sigma}_G$ for a state transition graph G constructed from M. The interpretations for both of the roots are the same, because of (M-3) and (G-3) are the same.

Suppose that all of the ancestors of a node σ_G in $\tilde{\sigma}_G$ match those of a node σ_M in $\tilde{\sigma}_M$, and that $\sigma_G = \sigma_M$. Also, we assume $\eta_S(\sigma_G) = s$, where s is a state in G.

We consider a child σ'_M of σ_M. From (M-4), $\sigma'_M(b_j) = \sigma_M(F_j)$ and $\sigma'_M(t_j) = \sigma_M(T_j)$. Let us define σ^*_M as a combined interpretation of σ'_M for the next state variables and σ_M for the other variables. Then, formula (1) is true under σ^*_M.

We see if we can generate, as a child of s, some state s_i. In the construction of G, formula (1) is transformed to $\alpha \triangleq \alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_p$, and thus some α_i must be true for σ^*_M. We check if state s_i can be generated for this α_i as a child of s. The formulas in reachable condition set C_s are all true under σ_G because of (G-5). Recall that α_i is decomposed as $\beta_1 \wedge \beta_2 \wedge \cdots \wedge \beta_q$, and β_j's without next state variables are added to the condition set C_{s_i} of s_i. These β_j's are true for σ^*_M and thus σ_M. Therefore, C_{s_i} is satisfiable, and s_i is generated as a state in G. We next see if it is possible to have σ'_G for s_i which is equivalent to σ'_M.

Since we can generate s_i, σ'_G for s_i must satisfy (G-1)–(G-6). From (G-1),

any interpretation given at a state is the same as that given at its next state except for next state variables. And, since $\sigma_G = \sigma_M$, we can say $\sigma'_G(\vee_{c\in C_{s_i}} c) = \sigma_G(\vee_{c\in C_{s_i}} c) = \sigma_M(\vee_{c\in C_{s_i}} c) = true$. From (G-5), $\sigma'_G(t_j) = \sigma'_G(\overrightarrow{t}[j])$. Again from (G-1), $\sigma'_G(\overrightarrow{t}[j]) = \sigma_G(\overrightarrow{t}[j])$, and this is equivalent to $\sigma_M(\overrightarrow{t}[j])$, and also to $\sigma'_M(\overrightarrow{t}[j])$, because of (M-1). This is also the case for b_j and $\overrightarrow{b}[j]$. By giving the same interpretation to input variables for σ'_M and σ'_G, we can have $\sigma'_M = \sigma'_G$.

On the other hand, suppose that we have an interpretation tree $\tilde{\sigma}_G$ for state transition graph G, and an interpretation tree $\tilde{\sigma}_M$ for an EUF machine M.

From (G-3) and (M-3), we can see $\sigma^0_G = \sigma^0_M$, where σ^0_G and σ^0_M are both the initial node of the two interpretation tree.

Suppose that all of the ancestors of a node σ_M in $\tilde{\sigma}_M$ match those of a node σ_G in $\tilde{\sigma}_G$, $\sigma_M = \sigma_G$. We also assume that σ_G has a child σ'_G and that (s, s') is an edge of G, and $\eta_S(\sigma_G) = s$, $\eta_S(\sigma'_G) = s'$. We see if we can have node σ'_M in $\tilde{\sigma}_M$, which is equivalent to σ'_G.

From the construction of s' from s, some α_i is used. This α_i is true under the combined interpretation σ^*_G. Note that C does not contain any present or next state variables. From (G-5) and (G-1), we can see $\sigma'_G(\vee_{c\in C_{s'}} c) = \sigma_G(\vee_{c\in C_{s'}} c) = \sigma_M(\vee_{c\in C_{s'}} c)$. From (M-1), $\sigma_M(\vee_{c\in C_{s_i}} c) = \sigma'_M(\vee_{c\in C_{s_i}} c)$. Also, we can see $\sigma'_G(t_j) = \sigma'_G(\overrightarrow{t}[j]) = \sigma_G(\overrightarrow{t}[j]) = \sigma_M(\overrightarrow{t}[j])$. From (M-4), $\sigma'_M(t_j) = \sigma_M(\overrightarrow{t}[j])$. Thus, we can have $\sigma'_G(t_j) = \sigma'_M(t_j)$. Similarly, we can see $\sigma'_G(b_j) = \sigma'_M(b_j)$.

Again, by giving the same interpretation to input variables for σ'_G and σ'_M, we can have $\sigma'_G = \sigma'_M$.

A.2 Proof of Lemma 2

Suppose that we have an interpretation tree $\tilde{\sigma}_G$ for state transition graph G, and an interpretation tree $\tilde{\sigma}_{G^{ex}}$ for its extended state transition graph G^{ex}.

For a root node σ^0_G in $\tilde{\sigma}_G$, suppose that $\sigma^0_G(z^{\overrightarrow{p}}) = true$ for some $z^{\overrightarrow{p}}$. From the way of constructing G^{ex} and (G-5), we can find an interpretation tree with root node $\sigma^0_{G^{ex}}$ such that $\sigma^0_{G^{ex}}(z^{\overrightarrow{p}}) = true$.

Suppose that all of the ancestors of a node σ_G in $\tilde{\sigma}_G$ match those of $\sigma_{G^{ex}}$ in $\tilde{\sigma}_{G^{ex}}$, and $\sigma_G = \sigma_{G^{ex}}$. We assume that $\eta_S(\sigma_G) = s$ and $\eta_{S^{ex}}(\sigma_{G^{ex}}) = s^{ex}$. If σ_G has a child σ'_G at state s', then, for some vector $\overrightarrow{p}$, $\sigma'_G(z^{\overrightarrow{p}})$ must be true. From the way of construction of G^{ex}, there exists a state $s^{ex\prime}$ in G^{ex} which includes $z^{\overrightarrow{p}}$ in its condition set. For this $s^{ex\prime}$, $\sigma'_{G^{ex}}$ can be introduced so that $\sigma'_{G^{ex}}(z^{\overrightarrow{p}})$ is true, because of (G-5). Again, by giving the same interpretation to input variables for σ'_G and $\sigma'_{G^{ex}}$, we can have $\sigma'_G = \sigma'_{G^{ex}}$.

On the other hand, If $\sigma_{G^{ex}}$ has a child $\sigma'_{G^{ex}}$ at state $s^{ex\prime}$, then, from the way of construction, for some vector $\overrightarrow{p}$, $\sigma'_{G^{ex}}(z^{\overrightarrow{p}})$ is true. We can always find σ'_G as a child of σ_G such that $\sigma'_G(z^{\overrightarrow{p}}) = true$, because $z^{\overrightarrow{p}}$ is satisfiable. Otherwise, $s^{ex\prime}$ cannot be generated as a state. Again, by giving the same interpretation to input variables we can conclude $\sigma'_G = \sigma'_M$.

A.3 Proof of Lemma 3

Suppose that we have a non-approximate graph G and its approximate graph G_a. We show an arbitrary interpretation tree $\tilde{\sigma}_G$ for G is a subtree of some interpretation tree $\tilde{\sigma}_{G_a}$ for G_a.

From the way of construction interpretation trees, the root node of $\tilde{\sigma}_G$ can be the same as that of $\tilde{\sigma}_{G_a}$.

Suppose that we have node σ_G in $\tilde{\sigma}_G$ such that $\eta_S(\sigma_G) = s$, and its ancestors match a subtree of some interpretation tree $\tilde{\sigma}_{G_a}$, and also $\eta_S(\sigma_{G_a}) = s_a$, and $\sigma_G = \sigma_{G_a}$.

We assume that s has an ancestor s' with σ'_G, that is, $\eta_S(\sigma'_G) = s'$. Let us have state s'_a as a child of s_a such that s'_a is generated by using the same α_i as s'. If no state merging nor term-height reduction has been done in generation of s'_a, then, by giving σ'_{G_a} the same interpretations to input variables as those by σ'_G, we can find out σ'_{G_a} such that $\sigma'_{G_a} = \sigma'_G$.

Suppose that state merging has been done in generation of s'_a. Firstly, we give, to σ'_{G_a}, the same interpretations to the renamed variables as those which were supposed to be given at the node before state merging. Since condition set C''_{s_a} before state merging of s'_a is true and (G-5), the condition set C'_{s_a} for s'_a is also true under σ'_{G_a}. Thus, we can obtain $\sigma'_{G_a} = \sigma'_G$.

Suppose that s'_a is given as a resulting state of term-height reduction, and that term t has been replaced by variable c. Then, by giving σ'_{G_a} the same interpretations to c as that which was supposed to be given for t at a state before term-height reduction, we can obtain $\sigma'_{G_a} = \sigma'_G$.

A.4 Proof of Theorem 1

The proof is given by induction on structure of a given EUF-CTL formula.

Suppose that some equation p (EUF equation) is labeled at state s in an ap-

proximate (extended or non-extended) state transition graph G_a. Then, for all the interpretation trees, p is true at their root nodes, because p is included in the condition set of s, and, from (G-5), p is true at the root node of any interpretation trees starting from s. Likewise, for negations of equations or (non-temporal) EUF formulas, if they are labeled at s, then the formulas are true at the root node of any interpretation trees starting from s.

Suppose that $\mathbf{AF}f$ is labeled at state s. Then, for any path from s in G_a, there exists at least a state where f is labeled. This implies that, in any interpretation tree starting from s, for any path in the tree, there is a node which makes f is true. Thus, $\mathbf{AF}f$ is also true at a root node of any interpretation tree starting from s. Likewise, we can easily show that, for other EUF-CTL formulas including temporal operators labeled at state s, they also true at a root node of any interpretation tree starting from s.

Then, Lemma 1, 2, and 3 guarantees that any interpretation tree for M is a subtree of an interpretation tree for G_a^{ex}. If some formula f is true for some interpretation tree, then it is also true for its subtree, because f describes a property for all the paths in the tree.

(Received December 1, 2009)
(Revised February 26, 2010)
(Accepted April 14, 2010)
(Released August 16, 2010)

(Recommended by Associate Editor: *Shinji Kimura*)

Kiyoharu Hamaguchi received his 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 an Associate Professor. His current interests include formal verification and computer aided design.

Kazuya Masuda received his bachelor degree at School of Engineering Science, Osaka University, Japan in 2007. He is currently a student in the master course of Graduate School of Information Science and Technology.

Toshinobu Kashiwabara received his 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.