ACTL Local Model Update with Constraints

Michael Kelly[1], Fei Pu[2], Yan Zhang[1], and Yi Zhou[1]

[1] Intelligent Systems Laboratory
School of Computing & Mathematics
University of Western Sydney, Australia
{mkelly,yan,yzhou}@scm.uws.edu.au
[2] School of Computer and Information Engineering
Zhejiang Gongshang University, China
pufei@zjgsu.edu.cn

Abstract. The recent development of model update aims to enhance model checking functions and provides computer aided modifications in system development. On the other hand, constraints have been playing an essential role in describing rational system behaviours. In previous model update approaches, constraints are usually not considered in the update process. In this paper, we present an ACTL - a widely used fragment of Computation Tree Logic (CTL), local model update approach where constraints have been explicitly taken into account. This approach handles constraints effectively by integrating constraint automata into the underlying model update. We demonstrate the effectiveness of our approach through the case study of the correction of the well known mutual exclusion program.

1 Introduction

In current approaches of system verification and modifications, an important idea is to integrate AI techniques, such as belief revision and model update, into model checking to develop effective system repairing tools, e.g. [3,7,11]. However, a major obstacle restricting this idea to apply to large scale problem domains, is that the update prototype has to take the entire system model (e.g. a complete Kripke model) as the input. This, obviously, is not generally feasible for large scale domains.

On the other hand, it is well understood that counterexamples generated from a model checking procedure plays an essential role in system repairing, because a counterexample usually localizes certain information that reveals how the system fails the specification property, e.g. [6]. Therefore, a natural way to overcome the difficulty of model update approach mentioned above is that we should develop a local model update that only applies to counterexamples and effectively generates candidate modifications for the original system. Furthermore, since constraints have been used in specifying system's rational behaviours, the underlying update approach should also carefully address this issue during an update process.

In this paper, we present an ACTL local model update approach where constraints have been explicitly taken into account. This approach handles constraints effectively by integrating constraint automata into the underlying model update. We demonstrate the effectiveness of our approach through the case study of the correction of the well known mutual exclusion program.

2 Preliminaries

ACTL is a special fragment of Computation Tree Logic (CTL) and has attracted considerable studies from researchers, e.g. [2,4]. Besides Boolean connectives, ACTL provides both linear time operators X, F, G and U and the branching time operator A. The linear time operators allow one to express properties of a particular evaluation of the systems given by a series of events in time, and the branching time operator takes into account the multiple possible future scenarios starting from a given state at certain time.

ACTL has the following syntax given in Backus Naur form:

$\phi ::= \top \mid \bot \mid p \mid \neg p \mid \phi \wedge \psi \mid \phi \vee \psi \mid \mathrm{AX}\phi \mid \mathrm{AG}\phi \mid \mathrm{AF}\phi \mid \mathrm{A}[\phi \mathrm{U} \psi]$, where p is any propositional atom (variable). Let AP be a set of propositional variables. A *Kripke structure* M over AP is a triple $M = (S, R, L)$, where S is a finite set of states, $R \subseteq S \times S$ is a binary relation representing state transitions, and $L : S \to 2^{AP}$ is a labeling function that assigns each state with a set of propositional variables.

An ACTL formula is evaluated over a Kripke structure. A *path* in a Kripke structure from a state is a(n) (infinite) sequence of states. Note that for a given path, the same state may occur an infinite number of times in the path (i.e., the path contains a loop). To simplify our following discussions, we may identify states in a path with different position subscripts, although states occurring in different positions in the path may be the same. In this way, we can say that one state precedes another in a path without much confusion. Now we can present useful notions in a formal way.

Let $M = (S, R, L)$ be a Kripke structure and $s_0 \in S$. A *path* in M starting from s_0 is denoted as $\pi = [s_0, \cdots, s_i, s_{i+1}, \cdots]$, where $(s_i, s_{i+1}) \in R$ holds for all $i \geq 0$. If $\pi = [s_0, s_1, \cdots, s_i, \cdots, s_j, \cdots]$ and $i < j$, we denote $s_i < s_j$. For any $s \in S$, the satisfaction relation between (M, s) and an ACTL formula ϕ, denoted by $(M, s) \models \phi$, is defined in a standard way as described in [8].

Now we introduce the concept of tree-like Kripke structures [4]. Let G be a directed graph. A *strongly connected component* (SCC) C in G is a maximal subgraph of G such that every node in C is reachable from every other node in C. C is *nontrivial* iff either it has more than one node or it contains one node with a self-loop. The *component graph* $c(G)$ of G is the graph where the vertices are given by the SCCs of G, and where two vertices of $c(G)$ are connected by an edge if there exists an edge between vertices in the corresponding SCCs. Then we say a graph G is *tree-like* if (1) all its SCCs are cycles; and (2) $c(G)$ is a directed tree. We should note that condition (1) is necessary because some SCCs may not be cycles. For instance, in a graph $G = (V, E)$, where $V = \{s_1, s_2, s_3\}$ and $E = \{(s_1, s_2), (s_2, s_3), (s_3, s_3), (s_3, s_2)\}$, the subgraph $G' = (\{s_2, s_3\}, \{(s_2, s_3), (s_3, s_3), (s_3, s_2)\})$ is a SCC, but it is not a cycle because edge (s_3, s_3) also forms a self-loop.

Consider a Kripke model (M, s_0), where $M = (S, R, L)$ and $s_0 \in S$. We say that (M, s_0) is a *tree-like Kripke model* if its corresponding graph $G(M) = (S, R)$ is tree-like. In this case, we call the initial state s_0 the *root* of this tree-like model. Since a tree-like Kripke model may not be a strict tree (e.g. it may contain some cycles along a branch), we cannot follow the traditional notions of *child and parent* in a tree-like model. Instead, we define the following new concepts. We say state s is an *ancestor* of state s', if there is a path $\pi = [s_0, \cdots, s, \cdots, s', \cdots]$ such that $s < s'$ in π and for all $s^* \in \pi$ where $s^* < s$, $s^* \neq s'$. s is a *parent* of s' if s is an ancestor of s'

and $(s, s') \in R$. In this case, we also call s' is a *successor* of s. A state s is called *leaf* if it is not an ancestor of any other states. A tree-like model (M', s') is called a *submodel* of (M, s), if $M' = (S', R', L')$, $s' \in S'$, $S' \subseteq S$, $R' \subseteq R$, for all $s^* \in S'$, $L'(s^*) = L(s^*)$, and in M, s' is an ancestor of all other states in M'.

Fig. 1 from [4] shows an example of a tree-like counterexample for a specific ACTL formula.

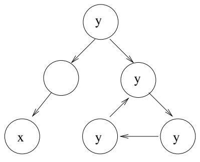

Fig. 1. A counterexample for AG$\neg x \vee$AF$\neg y$

It has been proved by Clarke et al that if an ACTL formula is not satisfied in a Kripke structure, then this Kripke structure must contain a tree-like counterexample with respect to this formula [4].

Theorem 1. *[4] ACTL has tree-like counterexamples.*

3 ACTL Local Model Update

Now we start to formalize the update on tree-like Kripke structures. For convenience, in the rest of the paper, we will call (M, s) a *tree-like Kripke model* without explicitly mentioning the corresponding tree-like Kripke structure $M = (S, R, L)$ where $s \in S$. We also define $\mathsf{Diff}(X, Y) = (X \setminus Y) \cup (Y \setminus X)$ where X, Y are two sets.

Definition 1. *Let (M, s) and (M_1, s_1) be two tree-like Kripke models. We say that a binary relation $H \subseteq S \times S_1$ is a* weak bisimulation *between (M, s) and (M_1, s_1) if:*

1. *$H(s, s_1)$;*
2. *given $v, v' \in S$ such that v is a parent of v', for all $v_1 \in S_1$ such that $H(v, v_1)$, the condition holds: (a) if v_1 is not a leaf, then there exists successor v_1' of v_1 such that $H(v', v_1')$, or (b) if v_1 is a leaf, then $H(v', v_1)$ (forth condition);*
3. *given $v_1, v_1' \in S_1$ such that v_1 is a parent of v_1', for all $v \in S$ such that $H(v, v_1)$, the condition holds: (a) if v is not a leaf, then there exists a successor v' of v such that $H(v', v_1')$, or (b) if v is a leaf, then $H(v, v_1')$ (back condition).*

Definition 2. *Let (M, s), (M_1, s_1) and (M_2, s_2) be three tree-like models, H_1 and H_2 be two weak bisimulations between (M, s) and (M_1, s_1) and between (M, s) and (M_2, s_2) respectively. We say that H_1 is* as bi-similar as *H_2, denoted by $H_1 \leq H_2$, if for all nodes $v \in S$, the following condition holds:*

1. *there exists an ancestor v' of v such that for all $v_1 \in S_1$ and $v_2 \in S_2$ satisfying $H_1(v', v_1)$ and $H_2(v', v_2)$, $\mathsf{Diff}(L(v'), L_1(v_1)) \subset \mathsf{Diff}(L(v'), L_2(v_2))$; or*
2. *for all $v_1 \in S_1$ and $v_2 \in S_2$ satisfying $H_1(v, v_1)$ and $H_2(v, v_2)$, $\mathsf{Diff}(L(v), L_1(v_1)) \subseteq \mathsf{Diff}(L(v), L_2(v_2))$.*

We write $H_1 < H_2$ iff $H_1 \leq H_2$ but $H_2 \not\leq H_1$.

Definition 2 specifies how we compare two weak bisimulations among three tree-like models. Intuitively, if H_1 and H_2 are two weak bisimulations between (M, s) and (M_1, s_1), and between (M, s) and (M_2, s_2) respectively, then $H_1 \leq H_2$ means that M_1 represents at least the same amount of *similar information* of M as M_2 does under H_1 and H_2 respectively.

Definition 3 (Tree-like model update). *Let ϕ be an ACTL formula and (M, s) a tree-like model such that $M \not\models \phi$. A tree-like model (M_1, s_1) is called a result of updating (M, s) with ϕ, if and only if*

1. $(M_1, s_1) \models \phi$;
2. *there is a weak bisimulation H_1 between (M, s) and (M_1, s_1) such that there does not exist another tree-like model (M_2, s_2) satisfying $(M_2, s_2) \models \phi$ and a weak bisimulation H_2 between (M, s) and (M_2, s_2) such that $H_2 < H_1$. In this case we say that (M_1, s_1) is an update result* under H_1.

Definition 3 is a declarative representation for the result from a tree-like model update. Condition 1 simply states that after the update, the resulting tree-like model should satisfy the updating formula. Condition 2 ensures that the resulting tree-like model is minimal from the original model under the weak bisimulation.

Example 1. Consider a tree-like model M as described in Fig. 2, which is a counterexample of AG$\neg x \vee$ AF$\neg y$. Then according to Definition 3, it is not hard to verify that (M_1, s_1) is a result of the update of (M, s) with AG$\neg x \vee$ AF$\neg y$, where (M_2, s_2) is not although it also satisfies AG$\neg x \vee$ AF$\neg y$.

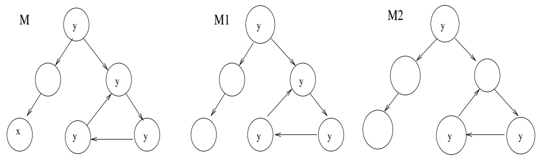

Fig. 2. Updating (M, s) with AG$\neg x \vee$ AF$\neg y$

4 Constraints and Constraint Automata

The minimal change principle for tree-like model update is purely defined based on Kripke structures, while no system constraints and other domain dependent information are considered in generating an update result. However, when we perform a model update, we may require this update not violate other specified system functions (e.g. breaking a deadlock should *not* violate a liveness in a concurrent program). Further, even if an updated model satisfies our minimal change criterion (i.e. Definition 3), it may not represent a valid result under the specific domain context. For instance, as showed in Example 1, (M_1, s_1) is a minimal updated model to satisfy formula AG$\neg x \vee$ AF$\neg y$. In practice, however, M_1 might not be acceptable if changing the variable x's value is not allowed in all states in model (M, s).

This motivates us to take relevant system constraints into account when we perform a model update. Besides logic based *system domain constraints*, which can usually be specified using ACTL (or CTL) formulas, there are some more complex constraints that are usually not expressible or difficult to be represented in the form of ACTL (or CTL) formulas. For instance, constraints related to system actions cannot be directly represented by ACTL (CTL) formulas. In the following, we study two such typical constraints: *variable constraints* and *action constraints* related to the underlying system behaviours.

Given a set of propositional variables V and a set of system actions A, we define a *Variable Constraint Automaton* constructed from V and A to be a finite deterministic automaton $\mathcal{VC}(V, A) = (S, \Sigma, \delta, q_0, F, v)$, where $S \subseteq 2^V \cup \{v\}$ is the set of states, $\Sigma = A$ is the input action symbols, $\delta : S \times \Sigma \to S$ is the total state transition function, $q_0 \in S$ is the *initial state*, $F \subseteq S$ is the set of final states, and $v \in S$ is the unique violation state.

A variable constraint automaton represents certain relations bound between a set of variables and a set of system actions. Consider two states $s_i, s_j \in S$, where s_i and s_j are not the violation state v, then a transition from s_i to s_j via action a: $\delta(s_i, a) = s_j$, indicates that by executing action a, variables' values represented by state s_i have to be changed to the corresponding variables' values represented by state s_j. Consider the variable constraint automaton depicted in Figure 3, action $i := t + 1$ ties two variables i and t, so that i's value must depend on t's value when this action is executed. On the other hand, action $i > 0?$ will not affect i and t's values, but execution of action $i < 0?$ will lead to violation state v ($*$ stands for any action symbols from Σ).

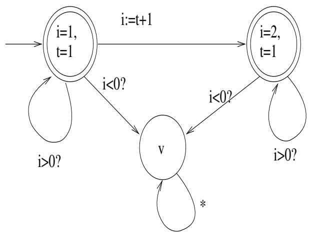

Fig. 3. A variable constraint automaton

Given a set of system actions A, we define an *Action Constraint Automaton* constructed from A to be a finite deterministic automaton $\mathcal{AC}(A) = (S, \Sigma, \delta, q_0, F, v)$, where $S \subseteq 2^A \cup \{v\}$ is the set of states, $\Sigma = \{\mathsf{preceded}, \mathsf{next}, \mathsf{exclusive}\}$ is the set of input action constraint symbols, $\delta : S \times \Sigma \to S$ is the total state transition function, q_0 is the initial state, F is the set of final states, and v is the unique violation state.

In an action constraint automaton, each state except the violation state is identified by a set of system actions. Then a transition between two states represents certain execution constraint between two specific sets of actions. For instance, if a_i and a_j are two system actions and states s_i and s_j are identified by actions $\{a_i\}$ and $\{a_j\}$ respectively, then $\delta(s_i, \mathsf{preceded}) = s_j$ means that action a_i should be executed *earlier* than action a_j, $\delta(s_i, \mathsf{next}) = s_j$ indicates that an execution of action a_i must *enforce* an immediate execution of action a_j, and $\delta(s_i, \mathsf{exclusive}) = s_j$ states that an execution of action a_i must exclude a following execution of action a_j.

As an example, Figure 4 shows that the philosopher has to pick up left fork first before he picks up right fork (similarly, here $*$ stands for any action constraint symbol from Σ). Although here we only consider three typical action constraints (i.e. preceded, next and exclusive), it is easy to extend this action constraint automaton to capture other action constraints.

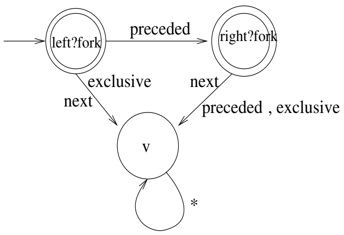

Fig. 4. An action constraint automaton

Generally from a given system, we may generate more than one variable and action constraint automata. Now we take the constraint automata into account when we perform model update. As discussed in the beginning of this section, in order to produce a more meaningful update result, our minimal change principle should be enhanced by integrating domain constraints, and system behaviour related variable and action constraints. Towards this aim, we first associate a set of system actions to a given Kripke structure. Recall that in the extent of model checking and model update, a Kripke structure actually represents the underlying system's behaviours where each state transition in the Kripke structure is caused by an execution of some system action. Therefore, for a given system, we can associate a set of system action A to the corresponding Kripke structure $M = (S, R, L)$ such that each state transition $(s, s') \in R$ is labeled by some action $a \in A$.

Definition 4. *Let $M = (S, R, L)$ be a Kripke structure, A a set of system actions associated to M, V a set of propositional variables, $\mathcal{VC}(V, A) = (S^{\mathcal{VC}}, \Sigma^{\mathcal{VC}}, \delta^{\mathcal{VC}}, q_0^{\mathcal{VC}}, F^{\mathcal{VC}}, v^{\mathcal{VC}})$ a variable constraint automaton, and $\mathcal{AC}(A) = (S^{\mathcal{AC}}, \Sigma^{\mathcal{AC}}, \delta^{\mathcal{AC}}, q_0^{\mathcal{AC}}, F^{\mathcal{AC}}, v^{\mathcal{AC}})$ an action constraint automaton. We say that M* complies to *$\mathcal{VC}$ and $\mathcal{AC}$, if the following conditions hold:*

1. *For each state transition $\delta^{\mathcal{VC}}(s_1, a) = s_2$ in $\mathcal{VC}(V, A)$ where s_1, s_2 are not the violation state, if there is a state $s \in S$ such that $s_1 \subseteq L(s)$ and $(s, s') \in R$ is labeled with a, then $s_2 \subseteq L(s')$ (i.e variable bindings through action a);*
2. *For each $\delta^{\mathcal{AC}}(s_1, \mathsf{preceded}) = s_2$ in $\mathcal{AC}(A)$, where s_1, s_2 are not the violation state, for each $a \in s_1$ and $a' \in s_2$, there exists a path $\pi = [s_0, \cdots, s_i, s_{i+1}, \cdots, s_j, s_{j+1}, \cdots]$ in M such that (s_i, s_{i+1}) is labeled with a and (s_j, s_{j+1}) is labeled with a' (i.e. a occurs earlier than a');*
3. *For each $\delta^{\mathcal{AC}}(s_1, \mathsf{next}) = s_2$ in $\mathcal{AC}(A)$, for each $a \in s_1$ and $a' \in s_2$, there exists a path $\pi = [s_0, \cdots, s_i, s_{i+1}, s_{i+2}, \cdots]$ in M such that (s_i, s_{i+1}) is labeled with a and (s_{i+1}, s_{2+2}) is labeled with a' (i.e. a' occurs next to a);*

4. *For each $\delta^{\mathcal{AC}}(s_1, \mathsf{exclusive}) = s_2$ in $\mathcal{AC}(A)$, for each $a \in s_1$ and $a' \in s_2$, there does not exist a path $\pi = [s_0, \cdots, s_i, s_{i+1}, \cdots, s_j, s_{j+1}, \cdots]$ in M such that (s_i, s_{i+1}) is labeled with a and (s_j, s_{j+1}) is labeled with a' (i.e. a's execution excludes a''s execution).*

Given a set of domain constraints $\mathcal{C}$ (ACTL formulas) and a class of constraint automata $\Im$, we say that a tree-like model (M, s) *complies to* $\mathcal{C}$ and $\Im$ if $(M, s) \models \mathcal{C}$ and (M, s) complies to each constraint automaton in $\Im$. Now we can extend our previous tree-like model update with complying to domain constraints and constraint automata.

Definition 5 (Update complying to constraints). *Let (M, s) be a tree-like model, $\mathcal{C}$ a set of ACTL formulas specifying the domain constraints, $\Im$ a class of system constraint automata, and ϕ a satisfiable ACTL formula such that $(M, s) \not\models \phi$. A tree-like model (M_1, s_1) is called a* result of updating (M, s) with ϕ complying to $\{\mathcal{C}, \Im\}$, *iff*

1. *$(M_1, s_1) \models \phi$ and complies to $\mathcal{C}$ and $\Im$;*
2. *there is a weak bisimulation H_1 between (M, s) and (M_1, s_1) such that there does not exist another tree-like model (M_2, s_2) satisfying that $(M_2, s_2) \models \phi$, (M_2, s_2) complies to $\mathcal{C}$ and $\Im$, and a weak bisimulation H_2 between (M, s) and (M_2, s_2) such that $H_2 < H_1$.*

5 A Case Study: The Mutual Exclusion Program

Based on the approach described earlier, we have implemented a system prototype to perform the tree-like local model update. A detailed report about the algorithms and system prototype implementation is referred to our other two papers [10,12]. To test our approach, we have undertaken four major case studies on this prototype. In this section, we provide some information one particular case study - the well known mutual exclusion program, and show how our approach can help to find a proper system modification.

Consider the concurrent program encoded in SPIN in above table. The program consists of processes PA() and PB(), which share two common boolean variables x and y. To ensure mutual exclusion of the assignments to x and y, some control variables $flag$ and $turn$, are introduced. Then there are two critical sections in each process, one for the assignments to x (statement 13 and statements 45-54 in PB()), and another one for the assignments to y in PA() (statements 23 in PA() and 52 in PB(), respectively. Notice that in PB(), the critical section for y is nested into critical section for x. Each variable flagiV (i=1,2 and V=A,B) indicates the request of process PV() to enter critical section i, and turniB tells whether such a request by process PB() in case of simultaneous requests should be granted.

The specification is formalized in an ACTL formula: φ =AG($\neg(ta \wedge (tb \vee tc))$), which describes that the program satisfies mutual exclusion for assignments to x and y, respectively. For example, PA() must not execute statement 13, if PB() executes statement 45 or 54 at the same time.

As mentioned in [3], this program contains about 10^5 states. We apply SPIN model checker to check whether this program satisfies property φ. With SPIN optimization,

Table 1. A mutual exclusion program - SPIN source code

1: *bool* flag1A, flaf2A;	38: proctype PB() {
2: *bool* turn1B,turn2B;	39: do
3: *bool* flag1B,flag2B;	40: :: flag1B=true;
4: *bool* x,y;	41: turn1B=false;
5: *bool* ta,tb,tc,td;	42: do
6: proctype PA() {	43: if
7: do	44: :: !flag1A|| turn1B $\rightarrow$
8: :: flag1A=true;	45: atom{x=x && y; tc=true;}
9: turn1B=false;	46: tb=false;
10: do	47: flag2B=true;
11: :: if	48: turn2B=false;
12: :: !flag1B||!turn1B $\rightarrow$	49: do
13: atom{x=x && y; ta=true;}	50: :: if
14: ta=false;	51: :: !flag2A||turn2B $\rightarrow$
15: flag1A=false;	52: y=!y;
16: if	53: td=false;
17: :: turn1B $\rightarrow$	54: atom{x=x||y; tb=true};
18: flag2A=true;	55: tb=false;
19: turn2B=true;	56: flag2B=false;
20: do	57: flag1B=false;
21: ::if	58: break;
22: ::!flag2B||!turn2B $\rightarrow$	59: :: else;
23: y=false;	60: fi;
24: tc=false;	61: od;
25: flag2A=false;	62: break;
26: break;	63: :: else;
27: ::else;	64: fi:
28: fi;	65: od;
29: od;	66: od;
30: ::else;	67: }
31: fi;	
32: break;	68: int{
33: ::else;	69: run PA();
34: fi;	70: run PB();
35: od;	71: }
36: od;	
37: }	

the program still contains 1800 states during the checking process. After SPIN model checking, it reports that property φ does not hold for this program and returns a tree-like counterexample (in fact it is linear), which only contains 22 states, as shown in Fig. 5.

Now we consider to update this counterexample by using our approach. First, we construct relevant constraint automata for this program, as discussed in section 4. In this case study, the particular variable constraint automaton shown in Fig. 6 will be directly embedded into the update process. In Fig. 6, "$*1$" indicates any statement in process PA() except statement $A : x = x\&\&y$, and "$*2$" indicates any statement in

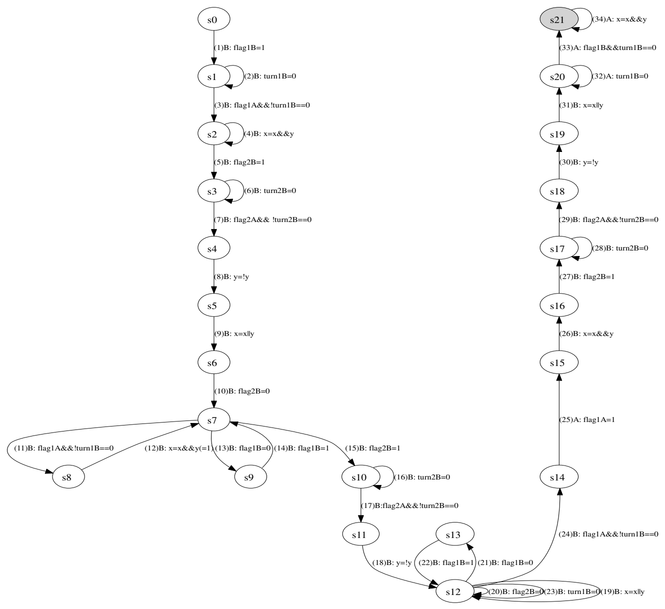

Fig. 5. A counterexample for AG$(\neg(ta \wedge (tb \vee tc))$

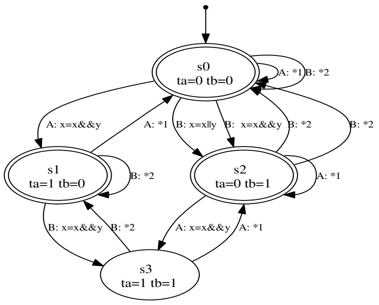

Fig. 6. The variable constraint automaton for ta and tb

process PB() except statements $B : x = x\&\&y$ and $B : x = x||y$. Intuitively, this variable constraint automaton represents the constraints between variables ta and tb with respect to various actions (statements) in the program.

Considering the counterexample shown in Fig. 5, we observe that state $s21$ violates property φ where $L(s21) = \{ta = 1, tb = 1, tc = 0, flag1A = 1, ...\}$. Note that the transition from $s20$ to $s21$ in Fig 5 corresponds to the automaton states $s2$ and $s3$ in Fig. 6 respectively. By applying our tree-like local model update with the associated constraint automaton (Definition 6), the counterexample will be minimally updated to satisfy φ: state $s21$ will be updated to either (1) $s21^{'}$: $L(s21^{'}) = \{ta = 0, tb = 1, tc = 0, flag1A = 1, ...\}$, or (2) $s21^{''}$: $L(s21^{''}) = \{ta = 1, tb = 0, tc = 0, flag1A = 1, ...\}$, while all other states in the local model will remain unchanged.

This update suggests that one possible modification for the original program (see Table 1) is to change statement 9 in PA() from “turn1B=false;” to “turn1B=true;”. A further SPIN model checking for the revised mutual exclusion program will confirm that this result from such local model update is a final correction to the original program.

6 Conclusion

In this paper we have developed an approach for ACTL tree-like local model update. In order to effectively generate the update result, we have proposed a minimal change principle based on weak bisimulation on tree-like structures, defined domain dependent constraint automata, and integrated them under a unified update formulation.

References

1. Berard, B., et al.: System and Software Verification: Model-Checking techniques and tools. Springer, Heidelberg (2001)
2. Buccafurri, F., Eiter, T., Gottlob, G., Leone, N.: On ACTL formulas having linear counterexamples. Journal of Computer and System Sciences 62, 463–515 (2001)
3. Buccafurri, F., Eiter, T., Gottlob, G., Leone, N.: Enhancing model checking in verification by AI techniques. Artificial Intelligence 112, 57–104 (1999)
4. Clarke Jr., E., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002), pp. 19–29 (2002)
5. Clarke Jr., E., Grumberg, O., Jha, S., Liu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of ACM 50, 752–794 (2003)
6. Fraser, G., Wotawa, F.: Nondeterministic testing with linear model-checking counterexamples. In: Proceedings of the 7th Intel. Conf on Quality Software, QSIC 2007 (2007)
7. Harris, H., Ryan, M.: Theoretical foundations of updating systems. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, pp. 291–298 (2003)
8. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems, 2nd edn. Cambridge University Press, Cambridge (2004)
9. Jobstmann, B., Staber, S., Griesmayer, A., Bloem, R.: Finding and fixing faults. Journal of Computer and System Sciences (2010) (to appear)
10. Kelly, M., Zhang, Y., Zhou, Y.: Local model update with an application to sliding window protocol (2010) (under review)
11. Zhang, Y., Ding, Y.: CTL model update for system modifications. Journal of Artificial Intelligence Research 31, 113–155 (2008)
12. Zhang, Y., Kelly, M., Zhou, Y.: Foundations of tree-like local model updates (2010) (under review)