IEICE TRANS. INF. & SYST., VOL.E94–D, NO.5 MAY 2011

PAPER Special Section on Formal Approach

An SMT-Based Approach to Bounded Model Checking of Designs in State Transition Matrix*

Weiqiang KONG[†a)], **Tomohiro SHIRAISHI**[†], **Noriyuki KATAHIRA**[†], **Masahiko WATANABE**[††], ***Nonmembers***, **Tetsuro KATAYAMA**[†††], ***and*** **Akira FUKUDA**[††††], ***Members***

SUMMARY State Transition Matrix (STM) is a table-based modeling language that has been frequently used in industry for specifying behaviors of systems. Functional correctness of a STM design (i.e., a design developed with STM) could often be expressed as invariant properties. In this paper, we first present a formalization of the static and dynamic aspects of STM designs. Consequentially, based on this formalization, we investigate a symbolic encoding approach, through which a STM design could be bounded model checked w.r.t. invariant properties by using Satisfiability Modulo Theories (SMT) solving technique. We have built a prototype implementation of the proposed encoding and the state-of-the-art SMT solver – Yices, is used in our experiments to evaluate the effectiveness of our approach. Two attempts for accelerating SMT solving are also reported.

key words: *state transition matrix, bounded model checking, invariant properties, satisfiability modulo theories*

1. Introduction

State Transition Matrix (STM) [2] is a table-based modeling language for specifying behaviors of systems. In a STM table, the horizontal axis declares possible status that a system under consideration can be in, the vertical axis declares possible events that may occur to the system, and a row-column intersection cell declares behaviors of the system when the designated event is dispatched (occurs) in the designated status. A whole system can be described by defining each of its sub-systems as a STM, and the sub-system STMs communicate via shared variables or message passing.

STM has been frequently used in industry for developing software designs, and been adopted as the modeling language of some commercial model-based CASE tools such as ZIPC [3]. Functional correctness of a STM design could often be expressed as invariant properties. However, assuring correctness, i.e., having a STM design satisfy certain desired invariant properties, is a non-trivial task. On the one hand, designers may introduce subtle logical errors incautiously into a STM design, especially when the design involves multiple communicating STMs, which makes it difficult for designers to maintain an overall image of the whole design. On the other hand, there is a lack of mechanized formal verification support for STM designs, e.g., ZIPC provides facilities for syntactic check only.

The subject of this paper is to provide mechanical formal verification support to STM designs. To this end, we first formalize the structure of STMs and the dynamic behaviors of a STM design. Consequentially, based on the formalization, we investigate a symbolic encoding approach for STM designs, through which a design could be bounded model checked (BMC) w.r.t. invariant properties by using Satisfiability Modulo Theories (SMT) solving technique [4]. More specifically, in this approach, all execution sequences within a given bound of a STM design and the negation of a predicate to be proved invariant of the design, are encoded into a quantifier-free formula whose satisfiability is to be checked w.r.t. some decidable background theories such as the theory of integers and the theories of various data structures such as arrays. Satisfiability of the resulted formula can be determined by state-of-the-art SMT solvers. If satisfied, a model (interpretation to all the (state) variables involved) of the formula is a witness of some bad behaviors of the design.

We have built a prototype implementation of the proposed encoding. In the implementation, SMT-LIB (Ver. 1.2, URL: goedel.cs.uiowa.edu/smtlib) [5] is chosen as our target language for encoding, since formulas to be checked for satisfiability written in this language can be solved by most state-of-the-art SMT solvers such as Yices [6]. We have conducted experiments with Yices to evaluate effectiveness of our approach. In the experiments, although general invariant properties could be expressed and checked with our encoding approach, we focused on three specific types that are often desired by industrial practitioners for STM designs [7]: (1) Unreachability of Invalid Cells, (2) Static Constraints, and (3) Dynamic Constraints. Additionally in the experiments, two simple ideas for accelerating SMT solving speed have been examined, and the results show that they are effective for most of the cases.

In this paper, we focus on STM designs that use shared variables as the means of communication. A simplified *Money-Changer* (MC) system modeled with STMs is used as our demonstration example (Fig. 1). The system consists of two components modeled as two STMs. (1) A de-

Manuscript received July 15, 2010.
Manuscript revised November 15, 2010.

[†]The authors are with Fukuoka Industry, Science and Technology Foundation (Fukuoka IST), Fukuoka-shi, 814–0001 Japan.
[††]The author is with CATS Co. Ltd., Yokohama-shi, 222–0033 Japan.
[†††]The author is with the Faculty of Engineering, University of Miyazaki, Miyazaki-shi, 889–2192 Japan.
[††††]The author is with the Graduate School of ISEE, Kyushu University, Fukuoka-shi, 819–0395 Japan.

*The paper is a revised and extended version of the paper [1] presented at ICCSA 2010.

a) E-mail: kong@lab-ist.jp

DOI: 10.1587/transinf.E94.D.946

vice called CHANGER, which supplies smaller denominations when equivalent amount of money in a larger denomination is inserted. The small denominations are delivered to another device called RETURNER. (2) Device RETURNER receives small denominations from CHANGER and waits until they are taken. The MC system is modeled in a greatly simplified means by abstracting away details irrelevant to the demonstration purpose. For example, `x10KYenRequest` is used to denote a request to change a banknote of 10K denomination (e.g., Japanese currency), but whether the (10K) banknote is inserted or not, and where it goes if inserted, are not modeled (i.e., supply just as requested). In addition, the numbers 20000 and 10000 inside the table denote equivalent amount of money but of small denominations (e.g., 1K denominations). Last, we intentionally introduced misoperations (design errors) to the system for demonstration purpose, e.g., the system's behaviors are *unreasonably* defined when there are not enough small denominations, which is discussed in detail in Sect. 4.

Organization. Section 2 presents a formalization of the static and dynamic aspects of a STM design. Section 3 describes the proposed encoding of a STM design into a quantifier free formula. Section 4 introduces the prototype implementation and experiments. Section 5 describes two attempts for accelerating SMT solving. Section 6 mentions related work, and Sect. 7 concludes the paper and proposes future work.

2. State Transition Matrix (STM)

STM is a table-based modeling language. Programming language and table notations can be mixed for specifying a system design. In this paper, a subset of STM notations is considered with the motivation of giving a precise description that can be formally verified. Adopting a similar approach used in [8], [9] for formalizing UML state machines, we formalize the structure of STMs and dynamic behaviors of a STM design.

2.1 Structure of STMs

We first describe an action language $\mathcal{L}$ that is needed for defining the structure of STMs. $\mathcal{L}$ is chosen to be a simple subset of C language and thus the syntax and semantics follow the conventions of C. Type system of $\mathcal{L}$ consists of Boolean, integers, and reals. Supported *expressions* of $\mathcal{L}$ are (1) Boolean literals `true` and `false`, integer literals and real literals, (2) Variable identifiers, and (3) infix expressions *leftexpr op rightexpr*, where *op* can be one of +, -, *, &&, ||, >, <, >=, <=, ==, or !=, with the semantics of C. Supported *statements* of $\mathcal{L}$ are (1) assignments of the form *lhs* = *rhs*, and (2) if statements of the form `if` *condition* {*statement1*} `else` {*statement2*}. We use $\mathcal{L}_B \subset \mathcal{L}$ to denote the set of Boolean expressions of $\mathcal{L}$, and $\mathcal{L}_{st} \subset \mathcal{L}$ to denote the set of statements of $\mathcal{L}$. Note that although the operator * is allowed in infix expressions of $\mathcal{L}$, we restrict it to the multiplication of a constant with a variable, e.g., `3*x`. This is because that the SMT solver Yices (Ver. 1.0.19) that we used in our experiments only supports *linear arithmetic*, which is also generally the case for most state-of-the-art SMT solvers, e.g., Z3 [10] and CVC3 [11]. See Yices home page for details.

Assuming this action language $\mathcal{L}$, the *structure* of a STM H is a tuple $\langle S, E, C\rangle$, where

- S is a finite set of status. Each status $s \in S$ is associated with a (unique) index number denoted by $index(s) \in Nat$. During execution of H, only one status denoted by $active(H)$ is *active*. Initially, the active status is s where $index(s) = 0$.
- E is a finite set of events consisting of external events E_{ext} and internal events E_{int}, where $E_{ext} \cap E_{int} = \emptyset$. An event $e_E \in E_{ext}$ is represented by a Boolean variable of $\mathcal{L}_B$ (whose name is prefixed with a lower-case "`x`" by convention), and an event $e_I \in E_{int}$ is represented by a Boolean expression (possibly a Boolean variable) of $\mathcal{L}_B$. Each event is associated with a (unique) index number denoted by $index(e) \in Nat$.
- C is a finite set of cells consisting of normal cells C_{nor}, ignore cells C_{ign}, and invalid cells C_{inv}. Each normal cell $c_N \in C_{nor}$ is a tuple $\langle s, e, u, a, s'\rangle \in S \times E \times (\mathcal{L}_B \cup \{\texttt{null}\}) \times (\mathcal{L}_{st} \cup \{/\}) \times S$. We define $source(c_N) = s$, $event(c_N) = e$, $guards(c_N) = u$, $actions(c_N) = a$, and $target(c_N) = s'$. Each ignore cell is a tuple $\langle s, e, /\rangle$, and each invalid cell is a tuple $\langle s, e, \times\rangle$. Functions *source* and *event* are also defined for $c_I \in C_{ign} \cup C_{inv}$ as for c_N, but *guards*, *actions* and *target* are not.

Events of a STM and guards of a normal cell are expressed as $\mathcal{L}_B$ expressions, and actions of a normal cell are a list of $\mathcal{L}_{st}$ statements (or just a symbol "/"). A cell c of a STM H, which is pinpointed by its index numbers ($index(source(c))$, $index(event(c))$) together with its guards if $guards(c) \neq \texttt{null}$ [†], specifies the behavior of H (under condition of $guards(c)$ if available) when event $event(c)$ is dispatched while H is in status $source(c)$ (i.e., $active(H) = source(c)$). If $c \in C_{nor}$, $actions(c)$ is executed [††] atomically and after that, H moves to status $target(c)$. If $c \in C_{ign}$, denoted in a STM table by the symbol "/", nothing changes. If $c \in C_{inv}$, denoted in a STM table by the symbol "×", an error occurs. Informally, an ignore cell means that the dispatch of an event in a status is ignored, and an invalid cell means that the dispatch of an event in a status is never possible (which however should be verified, as shown in Sect. 4).

Taking STM RETURNER in Fig. 1 as an example, we explain the above notations. Cell (1, 1) with guard `payMoney != 0` (for simplicity, c is used to denote this cell) is a normal cell, where $source(c) =$ `RETURN`, $event(c) =$ `xReceive`, $target(c) =$ `WAIT`, and $actions(c)$ consisting of

[†] Literal `null` is used only in the formalization (not in real STM tables) to denote that no guard is defined for a normal cell. For example, cell (0, 0) of STM RETURNER, denoted by c, has no guard, i.e., $guards(c) =$ `null`.

[††] Symbol "/" is used to denote *doing nothing*, and in the case $actions(c) = /$, execution of "/" changes nothing.

□ CHANGER		STOP	WAIT_REQUEST		WAIT_MONEY_TAKEN
		0	1		2

xChangePrepare	0	WAIT_REQUEST
		changeMoney=changeMoney+20000;
		xChangePrepare=false;
			/		/

x10KYenRequest	1	/	changeMoney >= 10000
			WAIT_MONEY_TAKEN
			payMoney=10000; payment=true;
			changeMoney=changeMoney-payMoney;
			x10KYenRequest=false;
				changeMoney < 10000
				STOP
				payMoney=0;
				payment=true;
				x10KYenRequest=false;
					/

getMoney	2	/	×		WAIT_REQUEST getMoney=false;

□ RETURNER		WAIT	RETURN	
		0	1	
payment	0	RETURN payment=false;	×	

xReceive	1	/	payMoney == 0
			WAIT
			getMoney=true;
			xReceive=false;
				payMoney != 0
				WAIT
				payMoney=0;
				getMoney=true;
				xReceive=false;

Fig. 1 A Money-Changer (MC) system consisting of CHANGER and RETURNER.

three $\mathcal{L}$ statements in form of assignment. Intuitive meaning of c is that if an external event for taking small denominations (denoted by event `xReceive`) occurs when device RETURNER is returning (non-zero) denominations (denoted by status `RETURN`), the device sets the variable `payMoney` as 0 to express that no small denomination is currently provided, set the variable `getMoney` as `true` to express that the small denominations have been taken, and set the variable `xReceive` as `false` to express that the event has been consumed. After that, device RETURNER changes to status `WAIT` to wait for small denominations from CHANGER for another exchange. Cell (0, 1) – $\langle$`WAIT`, `xReceive`, /$\rangle$ – is an ignore cell, meaning intuitively that receiving small denominations when the device is waiting for denominations from CHANGER changes nothing and is just ignored. Cell (1, 0) – $\langle$`RETURN`, `payment`, $\times\rangle$ – is an invalid cell, meaning intuitively that the event small denominations are being offered or ready (denoted by event `payment`) should not be dispatched when RETURNER is returning small denominations (not be taken yet).

2.2 Dynamic Behaviors of a STM Design

A system design D specified in STM generally consists of multiple STMs $H_1, \ldots, H_n$ that execute in an interleaving manner. *Dynamic behaviors* of D are defined based on the notion of *global states*, G.

We fix some notations to be used next. $Var(D)$ is used to denote the set of all $\mathcal{L}$ variables involved in D for expressing its events, guards, and actions. Given a set of variables X, $value(X)$ maps each $x \in X$ to its current value in x's domain. A global state $g \in G$ is a tuple $\langle active(H_i), value(Var(D))\rangle$, $1 \le i \le n$, where $active(H_i)$ is the currently active status of H_i.

Dynamic behaviors of D, denoted by $M(D)$, is captured by the tuple $\langle G, g_{init}, \Delta\rangle$, where G is the set of global states, $g_{init} \in G$ is an initial global state, and $\Delta \subseteq G \times G$ is a transition relation characterizing how D may evolve from one global state to another.

- In g_{init}, active status of each STM H_i is $s \in H_i.S$ [†], where $index(s) = 0$, and each variable $x \in Var(D)$ has its initial default value.
- For a normal cell $c \in H_i.C_{nor}$, c is said to be *enabled* in a global state g if $enable(c, g) = ((active(H_i) = source(c)) \wedge eval(event(c), g) \wedge eval(guards(c), g))$ evaluates to `true`, where $eval(expr, g)$ evaluates the value of an $\mathcal{L}$ expression $expr$ in the context of g. Specifically, we define $eval(\texttt{null}, g) = \texttt{true}$ when $guards(c) = \texttt{null}$. An enabled normal cell c could be executed. Executing c in g moves the design D to another global state g', and is denoted by $\langle g, g'\rangle \in \Delta$. The statements of $actions(c)$ are executed atomically as the conventions of C programming language. At each execution step of D, more than one cell might be enabled. In this case, one of them is selected and executed nondeterministically.

An external event is dispatched by the environment where system design D resides in, and an internal one is dispatched by the execution of D. An (external or internal) event $event(c)$ is not *implicitly consumed* after normal cell c is executed in a global state g, i.e., the Boolean value of the $\mathcal{L}_B$ expression denoting this event is not changed to `false` automatically in g'. Its truth value is just re-evaluated in g' based on the current values of its involved variables.

Given a system design D whose dynamic behavior is captured by $M(D) = \langle G, g_{init}, \Delta\rangle$, an *execution sequence* of $M(D)$ is defined as: $g_0, g_1, \ldots, g_n, \ldots$, where (1) $g_0 = g_{init}$, and (2) $\forall i \in Nat.\ \langle g_i, g_{i+1}\rangle \in \Delta$. Given a finite execution sequence sq, $len(sq)$ is used to denote the length of sq, e.g., $len(g_0, \ldots, g_n) = n+1$. *Reachable global states* w.r.t. $M(D)$ are those global states that are contained in any execution sequence of $M(D)$. The set of all reachable global states w.r.t. $M(D)$ is denoted by $\mathcal{R}_{M(D)}$. An *invariant prop-*

[†] $H_i.S$ denotes status set S of H_i. This way of reference applies to events set E and cells set C of H_i as well.

erty w.r.t. $M(D)$ is a state predicate $\rho : G \rightarrow$ Boolean such that ρ holds in all reachable global states of $M(D)$, i.e., $\forall g : \mathcal{R}_{M(D)}.\ \rho(g)$. An invariant property $\rho(g)$ could be parameterized with variables of data types of $\mathcal{L}$. Any execution sequence to $g \in \mathcal{R}_{M(D)}$ such that $\neg\rho(g)$ is called a counterexample for an invariant $\forall g : \mathcal{R}_{M(D)}.\rho(g)$.

3. A Symbolic Encoding Approach

The initial idea for converting a transition system into a quantifier-free formula and using SAT solving [4] to boundedly check the formula has been proposed in [12]. A more recent work [13] uses a similar approach to analyze model programs (representing transition systems) but based on SMT solving. Our encoding essentially follows the same techniques used in these two work (especially the second one) but is tuned to STM designs.

We demonstrate our encoding by considering a STM design D consisting of n STMs $H_1, \ldots, H_n$ and the given bound is `bd`. For each step k, $0 \leq k \leq$ `bd`, we use $x[k]$, $expr[k]$, and $stmt[k]$ to denote, respectively, a new variable, a new expression, and a new statement (all of language $\mathcal{L}$) used in step k. The ways of generating $expr[k]$ and $stmt[k]$ from $expr$ and $stmt$, respectively, are introduced later.

Formula for initial step 0, denoted by $step[0]$, representing the initial global state, is written as:

$$step[0] = \left(\bigwedge_{x \in Var(D)} x[0] = value(x)\right) \wedge \left(\bigwedge_{i=1}^{n} statusH_i[0] = 0\right)$$

The formula simply expresses that all variables involved in D, given the step number 0 (i.e., substitute each variable $x \in Var(D)$ with a new variable $x[0]$), have their initial default values. In addition, the active status of STM H_i, which is denoted by an additional variable $statusH_i$ ($statusH_i \notin Var(D)$ and its domain is $\{index(s) \mid s \in H_i.S\}$), is also given step number 0 and has the value 0.

To define formulas for steps other than 0, we first describe the encoding rules for a normal cell $c \in H_i.C_{nor}$ since the execution of such a cell corresponds to an execution step of D. Enable condition for c in step k, $1 \leq k \leq$ `bd` is defined as:

$$\begin{aligned} enable(c)[k] = guards(c)[k-1] \ \wedge\ & event(c)[k-1] \\ \wedge\ & statusH_i[k-1] = source(c) \end{aligned}$$

The expressions $guards(c)[k-1]$ and $event(c)[k-1]$ are generated by simply giving step number $k-1$ to all the variables involved in them. The last equation, which checks active status, is also generated by giving variable $statusH_i$ the step number $k-1$. Effects of execution of c in step k is defined as:

$$effects(c)[k] = actions(c)[k] \wedge statusH_i[k] = target(c)$$

If $actions(c) = /$, $actions(c)[k]$ is simply replaced by `true` or just omitted. Otherwise, assume that there are m $\mathcal{L}$ statements $st_1, \ldots, st_m$ in $actions(c)$, we generate a step k formula for each st_l, $1 \leq l \leq m$, by induction on the structure of $\mathcal{L}$ statements as follows.

If st_l is an assignment $lhs_l = rhs_l$ where lhs_l is a variable, the step k formula $st_l[k]$ is defined as $lhs_l[k] = rhs_l[k-1/k]$. Meaning of $rhs_l[k\text{–}1/k]$ is that (1) if variables $lhs_1, \ldots, lhs_{l-1}$ occurs in rhs_l, then the occurrence of these variables in rhs_l is given the step number k, and (2) for all other variables, step number $k-1$ is given. Taking actions of cell (1, 1) with guards `changeMoney >= 10000` of STM CHANGER as an example. Variable *changeMoney* in left-hand-side (lhs) in the third assignment is given the step number k, the same variable occurs in right-hand-side (rhs) is given $k-1$, and the variable *payMoney* in the rhs is given k, i.e., $changeMoney[k] = changeMoney[k-1] - payMoney[k]$, where *payMoney* in the first assignment is given k.

If st_l is a conditional statement `if` *condition* {*statement1*} `else` {*statement2*}, the step k formula $st_l[k]$ is defined as:

$$\begin{aligned} &\left(condition[k-1/k] \wedge statement1[k] \wedge \bigwedge_{x \in X_1} x[k] = x[k-1]\right) \vee \\ &\left(\neg\, condition[k-1/k] \wedge statement2[k] \wedge \bigwedge_{x \in X_2} x[k] = x[k-1]\right) \end{aligned} \tag{1}$$

where $statement1[k]$ and $statement2[k]$ are defined in the same way as for st_l. X_1 (respectively, X_2) is a set of variables that are assigned in *statement*2 (*statement*1) but not in *statement*1 (*statement*2). This elegant way of defining X_1 and X_2 is borrowed from [13]. For $condition[k\text{–}1/k]$, if variables $lhs_1, \ldots, lhs_{l-1}$ occurs in *condition* of st_l, then these variables are given step number k, and all the other variables are given $k-1$. Note that this encoding approach does not support multiple assignments to one variable in $actions(c)$, since in that case neither $k-1$ nor k may be appropriate to use. We extend this encoding by considering such situation in Sect. 4.3.

Based on the above definitions of $st_l[k]$, $actions(c)[k]$ is defined as follows:

$$actions(c)[k] = \left(\bigwedge_{l=1}^{m} st_l[k]\right) \wedge \left(\bigwedge_{x \in X} x[k] = x[k-1]\right)$$

where X contains all variables that are in $Var(D)$ but are not assigned in st_l, $1 \leq l \leq m$. Note that variables $x \in (X_1 \cup X_2)$ have become assigned variables in st_l (See Formula (1)) and are not contained in X.

For simplicity of demonstration, we use $AllVar(D)$ to denote the set of variables $Var(D) \cup \bigcup_{i=1}^{n}\{statusH_i\}$. The encoding rule for a normal cell c is defined as:

$$\begin{aligned} c[k] = (enable(c)[k] \wedge\ & effects(c)[k]) \\ \vee & \left(\neg\, enable(c)[k] \wedge \bigwedge_{x \in AllVar(D)} x[k] = x[k-1]\right) \end{aligned}$$

Intuitively, the above formula simply says that if normal cell

c is enabled, then effects will take place; otherwise, all variables remain unchanged.

Recall that an external event is dispatched by the environment where system D resides in. Therefore, to simulate the occurrence of external events, it is also needed to define an encoding rule for each $e \in \bigcup_{i=1}^{n} H_i.E_{ext}$. In this case, e should be just an $\mathcal{L}$ variable of type Boolean (denoting the external event. See the structure of STMs in Sect. 2.1). The rule, denoted by $ext(e)[k]$, is defined as follows:

$$ext(e)[k] = \Bigg(\neg e[k-1] \ \wedge \ e[k] = \texttt{true} \ \wedge \bigwedge_{x \in (AllVar(D) \setminus \{e\})} x[k] = x[k-1] \Bigg) \vee \Bigg(e[k-1] \ \wedge \bigwedge_{x \in AllVar(D)} x[k] = x[k-1] \Bigg)$$

The above rule simply changes the Boolean value of e to `true` when e is `false`, and all the other variables remain unchanged. Nothing changes if e is already `true`.

We are ready to define the complete formula for step number k, $1 \leq k \leq \texttt{bd}$. Since D is composed of STMs $H_1, \ldots, H_n$ that run in an interleaving manner, in each execution step of D, only one normal cell or a simulation of an external event is executed even if multiples are enabled. Assume $|\bigcup_{i=1}^{n} H_i.C_{nor}| + |\bigcup_{i=1}^{n} H_i.E_{ext}| = w$, we introduce, for each step k, a set of fresh Boolean flag variables $\{fl_1[k], \ldots, fl_w[k]\}$. Each variable $fl_j[k]$ ($1 \leq j \leq w$) uniquely corresponds to the execution of either a $c \in H_i.C_{nor}$ or an $e \in H_i.E_{ext}$ in step k. We use the symbol $Flag(c,k)$ (and respectively $Flag(e,k)$) to denote the variable in $\{fl_1[k], \ldots, fl_w[k]\}$ that corresponds to c (and respectively e), in step k. The formula $step[k]$ is defined as follows:

$$step[k] = \bigvee_{i=1}^{n} \Bigg(\Bigg(\bigvee_{c \in H_i.C_{nor}} (c[k] \ \wedge \ Flag(c,k)) \Bigg) \vee \Bigg(\bigvee_{e \in H_i.E_{ext}} (ext(e)[k] \ \wedge \ Flag(e,k)) \Bigg) \Bigg) \quad (2)$$

Furthermore, we define a global constraint, named as $AsynConstr$, on the above-introduced Boolean variables as follows:

$$AsynConstr = \bigwedge_{k=1}^{\texttt{bd}} \Bigg(\bigwedge_{j=1}^{w} \Bigg(fl_j[k] \Rightarrow \neg \Bigg(\bigvee_{m=1, m \neq j}^{w} fl_m[k] \Bigg) \Bigg) \Bigg)$$

The constraint simply says that in each execution step k, if a flag variable $fl_j[k]$ is true then all the other flag variables in the same step should be false. This constraint works (together with Formula (2)) for restricting *only one normal cell or a simulation of an external event is executed in each step*, since each $fl_j[k]$ is combined in conjunction with either a $c[k]$ or an $ext(e)[k]$ (See Formula (2)), and therefore $fl_j[k]$ and $c[k]$ (or ext(e)[k]) should both be true if their combination is to be satisfiable.

Finally, given a STM design D, an execution bound `bd`, and a state predicate ρ to be proved invariant w.r.t. D, the formula for checking ρ in all execution sequences of D within bound `bd` is as follows:

$$BMC(D, \rho, \texttt{bd}) = \Bigg(\bigwedge_{k=0}^{\texttt{bd}} step[k] \Bigg) \wedge AsynConstr \wedge \Bigg(\bigvee_{k=0}^{\texttt{bd}} (\neg \rho[k]) \Bigg) \quad (3)$$

where $\rho[k]$ is generated by simply given all variables involved in it with step number k. The satisfiability of such a formula can be solved by SMT solvers. If satisfied, a variable-value pair for all the variables $AllVar(D)[0], \ldots, AllVar(D)[k]$ comprises a witness of D's execution that violates predicate ρ.

4. Implementation and Experiments

We have built a prototype implementation of the above encoding approach. In this section, we demonstrate our implementation with some experiments on the MC system with Yices [6].

4.1 Implementation

Input of the prototype is a STM design (typically consisting of multiple STMs) developed using the ZIPC tool [3] whose structure and semantics are compliant with the STM's definitions in Sect. 2, and output of the prototype is a quantifier-free formula in SMT-LIB language (Version 1.2) [5] that represents $BMC(D, \rho, \texttt{bd})$. SMT-LIB language is chosen as the target language for implementing the encoding since formulas to be checked for satisfiability written in this language can be solved by several state-of-the-art SMT solvers [†], such as Yices [6].

We show part of the output formulas for steps 0 and 1 of the MC system in Figs. 2 and 3, respectively. Some SMT-LIB notations are introduced for understanding our implementation. A variable x (used in step k) of type Boolean is declared (with SMT-LIB keyword in typewriter font) as "`:extrapreds (x_k)`", and a variable y (in step k) of types integers and reals are declared as "`:extrafuns (y_k Int)`" and "`:extrafuns (y_k Real)`", respectively. A (sub)formula denoting $step[k]$ is declared as "`:assumption (`$step[k]$`)`", and the formula denoting negation of the predicate is declared as "`:formula (or (not(`$\rho[0]$`)) ... (not(`$\rho[\texttt{bd}]$`)))`". We use $rule_j$, $1 \leq j \leq w$, to denote the encoding rule for either a normal cell or a simulation of an external event, and we give a name (identifier), $\$ruleT_j$, to each $rule_j$. The step formula $step[k]$ is represented as:

[†]Some other state-of-the-art SMT solvers that support the SMT-LIB language include Z3 [10] and CVC3 [11] etc. More can be found in the webpage for SMT-LIB.

```
:extrapreds ((xChangePrepare_0) (x10KYenRequest_0) (xReceive_0) (payment_0) (getMoney_0))
:extrafuns  ((payMoney_0 Int) (changeMoney_0 Int) (statusChr_0 Int) (statusRer_0 Int))
:assumption ((and (= xChangePrepare_0 false) (= x10KYenRequest_0 false) (= xReceive_0 false) (= payment_0 false)
                  (= getMoney_0 false) (= payMoney_0 0) (= changeMoney_0 0) (= statusChr_0 0) (= statusRer_0 0)))
```

Fig. 2 Formula for Step 0 in SMT-LIB language.

```
:extrapreds ((xChangePrepare_1) (x10KYenRequest_1) (xReceive_1) (payment_1) (getMoney_1)
             (flag1_1) (flag2_1) (flag3_1) (flag4_1) (flag5_1) (flag6_1) (flag7_1) (flag8_1) (flag9_1) (flag10_1))
:extrafuns  ((payMoney_1 Int) (changeMoney_1 Int) (statusChr_1 Int) (statusRer_1 Int))
:assumption (
(flet ($r1_1 (if_then_else (and (= xChangePrepare_0 true) (= statusChr_0 0))
             (and (= xChangePrepare_1 false) (= x10KYenRequest_1 x10KYenRequest_0) (= xReceive_1 xReceive_0)
                  (= payment_1 payment_0) (= getMoney_1 getMoney_0) (= payMoney_1 payMoney_0)
                  (= changeMoney_1 (+ changeMoney_0 20000)) (= statusChr_1 1) (= statusRer_1 statusRer_0))
             (and (= xChangePrepare_1 xChangePrepare_0) (= x10KYenRequest_1 x10KYenRequest_0) (= xReceive_1
                  xReceive_0) (= payment_1 payment_0) (= getMoney_1 getMoney_0) (= payMoney_1 payMoney_0)
                  (= changeMoney_1 changeMoney_0) (= statusChr_1 statusChr_0) (= statusRer_1 statusRer_0))))
(flet ($r2_1 (...)) ..... (flet ($r10_1 (...))
(or (and $r1_1 flag1_1) (and $r2_1 flag2_1) ... (and $r10_1 flag10_1)))))))))))))
```

Fig. 3 Formula for Step 1 in SMT-LIB language.

$$
\begin{aligned}
step[k] = {} & \texttt{(flet ((\$}ruleT_1\ rule_1\texttt{)} \\
& \ldots \\
& \texttt{(flet ((\$}ruleT_w\ rule_w\texttt{)} \\
& \texttt{(or (and \$}ruleT_1\ fl_{1_k}\texttt{)} \\
& \qquad \ldots \\
& \qquad \texttt{(and \$}ruleT_w\ fl_{w_k}\texttt{))}_{\ldots w-1 \ldots}\texttt{)}
\end{aligned}
$$

where subscript $w-1$ denotes omitting brackets ")". `flet` is a SMT-LIB keyword to bind a formula with an identifier. The formula $step[k]$ intuitively means that $\$ruleT_j$, denoting $rule_j$, is used in the last `or` combined formula. Formula for step 0 in Fig. 2 corresponds to the initial global state. Formulas for steps k, $1 \leq k \leq$ `bd`, are written in a similar way as for step 1 in Fig. 3. Rules denoted by `$r1_1`,...,`$r7_1` correspond to the seven normal cells (of CHANGER and RETURNER). Rules denoted by `$r8_1`,...,`$r10_1` correspond to the simulation of the external events `xChangePrepare`, `x10KYenRequest`, and `xReceive`, respectively. In Fig. 3, we detailed the contents of `$r1_1`, while omitting the others. Readers are referred to [1] for more details of the implementation.

4.2 Experiments

Although general invariant properties could be expressed and checked by using our encoding approach, we focus on three specific ones in our experiments.

Unreachability of Invalid Cells (UIC). An invalid cell indicates that a specific event should never be dispatched in a specific status. When considered by the designer as invalid, the cell is marked as $\times$ in a STM table. However, it should be verified whether the cell is really unreachable or not. We extend $active(H)$ as $active(H, g)$ to denote the active status of STM H in a reachable global state g. A UIC property corresponding to STM H_i is expressed as $\forall c \in H_i.C_{inv}.\ \forall g \in \mathcal{R}_{M(D)}.\ \neg((active(H_i, g) = source(c)) \wedge eval(event(c), g))$.

For the MC system, cell (1, 2) of CHANGER and cell (1, 0) of RETURNER are invalid cells. The first invalid cell means intuitively that event *small denominations have been received* (denoted by event `getMoney`) should not be dispatched when *CHANGER is waiting for a request* (denoted by `WAIT_REQUEST`). The second has been explained in Sect. 2.1. We express the cells as two properties (step numbers are to be given later):

```
UIC1 = not((statusChr = 1) and (getMoney = true))
UIC2 = not((statusRer = 1) and (payment = true))
```

Static Constraints (STC). Static constraints demand certain correlation between status of different STMs. Such a correlation can be in the form of $\forall g \in \mathcal{R}_{M(D)}.\ active(H_A, g) = sa \Rightarrow active(H_B, g) = sb$. Two sample static constraint properties of the MC system are declared as follows:

```
STC1 = ((statusRer = 1) implies (statusChr = 2))
STC2 = ((statusChr = 1) implies (statusRer = 0))
```

Dynamic Constraints (DYN). Dynamic constraints demand certain correlation between status-change in one STM with a specific status of another STM. Such a correlation can be in the form of, e.g., when the status of STM A has just changed from sa to sa' ($sa' \neq sa$), the status of STM B should be sb', i.e., $\forall g \in \mathcal{R}_{M(D)}.\ active(H_A, g) = sa \wedge active(H_A, g') = sa' \Rightarrow active(H_B, g') = sb'$. A sample dynamic constraint property is as follows:

```
DYN = (statusChr = 1 and statusChr' = 2)
      implies (statusRer' = 0)
```

where a primed (with "'") variable denotes a successor state variable. Method for removing it is explained later when giving step numbers to the property.

For demonstration purpose, we define two more (irrational) static constraint properties as follows:

```
FSTC1 = (statusChr = 2) implies (statusRer = 0)
FSTC2 = (statusChr = 2) implies (statusRer = 1)
```

Assume a bound value, say 3, for bd, we show the SMT-LIB formulas representing STC1 and DYN with step numbers. Other properties mentioned above could be represented in a similar way.

```
:formula
(or
(not (implies (= statusRer_0 1) (statusChr_0 2)))
(not (implies (= statusRer_1 1) (statusChr_1 2)))
(not (implies (= statusRer_2 1) (statusChr_2 2)))
(not (implies (= statusRer_3 1) (statusChr_3 2))))

:formula
(or
(not (implies
     (and (= statusChr_0 1) (= statusChr_1 2))
     (= statusRer_1 0)))
(not (implies
     (and (= statusChr_1 1) (= statusChr_2 2))
     (= statusRer_2 0)))
(not (implies
     (and (= statusChr_2 1) (= statusChr_3 2))
     (= statusRer_3 0))))
```

Note that the two formulas should be written in different SMT-LIB files and checked separately. Operator not is added since we expect to falsify (i.e., find counterexamples of) the predicates.

Yices 1.0.19 is used (on Windows XP, 2.66 GHz, 3.25 GB RAM) in our experiments to check these properties. Results are shown in the left-hand side (lhs) of Fig. 4. Time in this figure (and in all the other figures hereafter) is an average of 5 times executions of Yices. Counterexamples are found for all the properties when certain bounds are set (we did not check FSTC1 and FSTC2 for the original MC system). A counterexample is indicated by the output sat of Yices, and the details of the counterexample – a set of variable-value pairs for all variables, which shows how the input formula is satisfied – can be output if appropriate parameters are set to Yices. Note that understanding the counterexamples by observing value-changes of all the variables is extremely cumbersome. However, the structure (execution sequence) of a counterexample could be easily understood by observing the introduced Boolean flag variables [†], since in each step k only one of the flag variables has the value true, which indicates the normal cell or simulation of external event executed in k.

Prop.	bd	Verdict	Time
UIC1	20	unsat	0.51
UIC1	21	sat	0.48
UIC2	20	unsat	0.45
UIC2	21	sat	0.56
STC1	16	unsat	0.46
STC1	17	sat	0.43
STC2	18	unsat	0.57
STC2	19	sat	0.45
DYN	19	unsat	0.53
DYN	20	sat	0.55

Prop.	bd	Verdict	Time
UIC1	150	unsat	53.77
UIC2	150	unsat	42.78
STC1	150	unsat	45.89
STC2	150	unsat	55.40
FSTC1	4	unsat	0.14
FSTC1	5	sat	0.15
FSTC2	3	unsat	0.15
FSTC2	4	sat	0.14
DYN	150	unsat	54.17

Fig. 4 Results of checking *original* (lhs) and *revised* (rhs) MC system using Yices. Time is counted in *Seconds*.

Taking UIC1 as an example, we explain reasons for the occurrence of the counterexamples. When a money-change request comes (i.e., event x10KYenRequest is dispatched), if CHANGER does not have enough money (this is the case after two money-change requests have been correctly processed), CHANGER executes its cell (1, 1) with guard changeMoney < 10000 by: dispatches event payment, and switches to status STOP, etc. From now on, CHANGER may be initialized again when external event xChangePrepare is dispatched and then CHANGER can switch to status WAIT_REQUEST. On the other hand, RETURNER will be able to execute its cell (0, 0) since event payment has been dispatched, and consequentially execute (1, 1) with guard payMoney = 0, and finally getMoney will become true. The system falls into a situation that violates UIC1.

Based on the above analysis, we remove the first two assignments in cell (1, 1) with guard changeMoney < 10000 of CHANGER, and check these properties again. The results are shown in the right-hand side (rhs) of Fig. 4. No counterexample is found this time for all the five properties when bound is set to 150. Two counterexamples for (irrational) properties FSTC1 and FSTC2 are found when bound is set to 5 and 4, respectively, whose details are omitted here.

It should be pointed out that the encoded SMT-LIB formula representing $BMC(D, \rho, \mathtt{bd})$ problem of the MC system could be solved by Yices using different background theories, such as linear integer arithmetic, linear real arithmetic, or possibly bit-vector arithmetic. Yices analyzes the input formula and decides the theory (or theories) to use by itself if no theory (to be used) are specified explicitly in the formula. However, we have found that specifying a theory explicitly could greatly enhance solving efficiency. For example, if no theory is specified, solving STC1 and STC2 cost 144.46s and 267.63s, respectively, when bd was set to 50. In our experiments on the MC system, we specified and let Yices use (quantifier free) linear real arithmetic for our problem, which could be declared in SMT-LIB format as :logic QF_LRA. Note that specifying this also permits the mixed usage of both integers and reals by Yices.

4.3 Encoding Multiple Assignments to Variables

As mentioned, our approach for encoding $actions(c)[k]$ of a normal cell does not support multiple assignments to the same variable in $st_1, \ldots, st_m$, as, e.g., shown in Fig. 5, since neither $k-1$ nor k might be an appropriate step number to give to the variable.

Inspired by the work in [14] for SMT-based encoding

[†]Note that $\$ruleT_1, \ldots, \$ruleT_w$ are local variables used in the binding structure, and thus are not output by Yices.

```
changeMoney = changeMoney + 20000;
xChangePrepare = false;
changeMoney = changeMoney - 10000;
```

Fig. 5 Hypothetical multiple assignments to changeMoney.

of sequential C source codes, we extend our previous approach by maintaining a number $\alpha(x,k)$ for each variable $x \in AllVar(D)$, which indicates the number of indexes (i.e., step number, as we called in previous sections) given to x prior to step k. $rule_i(x)$ is used to denote the number of assignments to x existing in $rule_i$, $1 \le i \le w$, in $step[k]$, and the biggest number among $rule_i(x)$ is denoted by $max(x)$. Then, numbers *from* $\alpha(x,k)$ *to* $max(x)$ will be used as indexes for encoding $action(c)[k]$, rather than only $k-1$ and k. In addition, $\alpha(x,k+1) = \alpha(x,k) + max(x)$.

Assume that the hypothetical situation for variable changeMoney in Fig. 5 is the actions of cell (0, 0) of STM CHANGER of the MC system shown in Fig. 1, and also assume that $\alpha(\texttt{changeMoney},1) = 0$ and $max(\texttt{changeMoney}) = 4$, the rule $r1_1 shown in Fig. 3 could be changed into:

```
$r1_1 (if_then_else
(and (= xChangePrepare_0 true) (= statusChr_0 0))
(and (= changeMoney_1 (+ changeMoney_0 20000))
     (= xChangePrepare_1 false)
     (= changeMoney_2 (- changeMoney_1 10000))
     (= changeMoney_3 changeMoney_2)
     (= changeMoney_4 changeMoney_3)
     (= statusChr_1 1)...)
(and (= changeMoney_1 changeMoney_0)
     (= changeMoney_2 changeMoney_1)
     (= changeMoney_3 changeMoney_2)
     (= changeMoney_4 changeMoney_3)
     (= xChangePrepare_1 xChangePrepare_0)...))
```

Note that, by assuming $max(\texttt{changeMoney}) = 4$, we assume that changeMoney is assigned four times in a some other normal cell of the MC system. And then numbers starting from $\alpha(\texttt{changeMoney},2) = \alpha(\texttt{changeMoney},1) + max(\texttt{changeMoney}) = 4$ are to be used as indexes for changeMoney in step 2. Therefore, although changeMoney is only assigned twice in cell (0, 0), it is necessary to define changeMoney_3 and changeMoney_4 as well in $r1_1 to fill up the gap.

5. Attempts for Accelerating SMT Solving

It could be observed from Fig. 4 that SMT solving speed slows down greatly as with the increase of step numbers (i.e., as with the increase of the size of the formula under concern). We describe two attempts made in our experiments for accelerating SMT solving.

SMT solving is typically extension of SAT solving [4] with specialized solvers for first-order theories, e.g., Yices integrates a SAT solver that relies on the techniques of the well-known SAT solver – Chaff [15]. Most often solving algorithms implemented in SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [16], in which Boolean Constraints Propagation (BCP) (i.e., identify consequential variable assignments that are required by the current variable state to satisfy the target formula) is known to cost 80-90 percent of the total solving time [15]. Introducing additional (but possibly redundant) knowledge, which is usually employed in the (*conflict-analysis*) *learning process* [4] of SAT solving algorithms, is an effective method for pruning search space and hence reducing solving cost. Our attempts for accelerating SMT solving are inspired by this idea of making use of additional knowledge. However, rather than being inside the solving algorithms, our attempts work outside the algorithms by introducing additional knowledge into the formula that encodes the bounded model checking problem for a STM design. This makes our attempts applicable for end-users of SMT solvers as well.

5.1 Structure Knowledge of a STM Design

Our first attempt is to introduce structure knowledge of a STM design. The knowledge simply characterizes the fact that any normal cell c of a STM H_i, $1 \le i \le n$, where $source(c) \neq active(H_i, g)$, could not be executed in the global state g. The kind of knowledge might possibly be helpful for restricting the choices of the next transitions to be executed.

By using the Boolean flag variables that we introduced, the above mentioned knowledge could easily be expressed. Taking STM RETURNER as an example. Assume step number k, and assume that the flag variables associated with normal cells (0, 0) and (1, 1) with two different guards are $flagT1$, $flagT2$, and $flagT3$, respectively. The knowledge for STM RETURNER in step k is as follows: $(statusRer_{k-1} = 0 \Rightarrow \neg(flagT2_k \vee flagT3_k)) \wedge (statusRer_{k-1} = 1 \Rightarrow \neg flagT1_k)$. We use $know_i[k]$, $1 \le i \le n$, to denote the structure knowledge for STM H_i in step k. Formula (3) could now be changed into:

$$BMC(D,\rho,\mathtt{bd}) = \left(\bigwedge_{k=0}^{\mathtt{bd}}\left(step[k] \wedge \bigwedge_{i=1}^{n} know_i[k]\right)\right) \wedge AsynConstr \wedge \left(\bigvee_{k=0}^{\mathtt{bd}}(\neg\rho[k])\right)$$

The method of giving step number k to $know_i$ (i.e., $know_i[k]$) is same as the one for predicate $\rho[k]$ introduced in Sect. 3. Specifically, in the case that $k=0$, we define $know_0$ as true, which could simply be omitted. We add the knowledge for STMs CHANGER and RETURNER, and check the *original* and *revised* MC system using Yices again. From the results shown in Fig. 6, we observed that (1) the usage of structure knowledge does not change the satisfiability of the original properties (column Verdict); and (2) this usage speeds up the solving time for most of the cases.

5.2 Boundedly Proved Invariants

Our second attempt is to make use of those predicates that

Prop.	bd	Verdict	Time	
			Without Know	*With Know*
UIC1	20	unsat	0.51	0.62
UIC1	21	sat	0.48	0.64
UIC2	20	unsat	0.45	0.53
UIC2	21	sat	0.56	0.48
STC1	16	unsat	0.46	0.55
STC1	17	sat	0.43	0.39
STC2	18	unsat	0.57	0.59
STC2	19	sat	0.45	0.50
DYN	19	unsat	0.53	0.68
DYN	20	sat	0.55	0.57

Prop.	bd	Verdict	Time	
			Without Know	*With Know*
UIC1	150	unsat	53.77	33.11
UIC2	150	unsat	42.78	41.60
STC1	150	unsat	45.89	35.75
STC2	150	unsat	55.40	38.81
FSTC1	4	unsat	0.14	0.11
FSTC1	5	sat	0.15	0.13
FSTC2	3	unsat	0.15	0.15
FSTC2	4	sat	0.14	0.17
DYN	150	unsat	54.17	41.07

Fig. 6 Results of checking *original* (upper) and *revised* (lower) MC system without and with *Know* (denoting *Knowledge*).

have already been proved to be invariants (within a certain bound) of a STM design. We name such invariants as *Bounded Invariants* or *B-Invariants* for short. A B-Invariant essentially characterizes certain constraints (knowledge) on the state variables of the STM design within a certain execution bound. Therefore, these constraints might be helpful for pruning search space and thus reducing the solving costs.

Assume that there are h B-Invariants $inv_1 \ldots inv_h$, i.e., we have already checked them by using the SMT-based BMC method introduced before, and no counterexamples are reported within a given bound. For demonstration simplicity, we assume the bound for all these B-invariants to be bd. Then, to check the predicate ρ, Formula (3) could now be changed into:

$$BMC(D,\rho,\mathsf{bd}) = \left(\bigwedge_{k=0}^{\mathsf{bd}}\left(step[k] \wedge \bigwedge_{t=1}^{h} inv_t[k]\right)\right) \wedge AsynConstr \wedge \left(\bigvee_{k=0}^{\mathsf{bd}}(\neg\rho[k])\right)$$

As shown in Sect. 4.2, UIC1, UIC2, STC1, STC2, and DYN are B-Invariants w.r.t. bound 150 for the revised MC system. For demonstration simplicity, we name them as ①–⑤, respectively. To examine the effectiveness, we conducted a series of experiments and only part of the results are listed in Fig. 7 due to space limitation. In the experiments, each of the B-Invariants is checked by assuming the existence of the remaining ones (or their combinations). For the irrational predicates FSTC1 and FSTC2, the satisfiability is not changed when check them by assuming and using the B-Invariants STC1, STC2, and their combination. For UIC1, we checked it by assuming and using all the remaining B-Invariants and some of their combination. In addition to only using B-Invariants, we have also experimented using the combination of B-Invariants with structure knowledge (*know* in Fig. 7) mentioned in Sect. 5.2. The satisfiability remains unchanged and the solving time is greatly decreased for most of the cases.

No.	Prop.	B-Inv	bd	Verdict	Time
①	UIC1	**Without B-Inv**	**150**	**unsat**	**53.77**
		②	150	unsat	43.96
		③	150	unsat	17.78
		④	150	unsat	22.25
		⑤	150	unsat	50.22
		②+③	150	unsat	19.62
		②+④	150	unsat	33.37
		②+③+④	150	unsat	23.48
		②+③+④+⑤	150	unsat	12.55
		②+③+④+⑤+*Know*	150	unsat	19.08
②	UIC2	**Without B-Inv**	**150**	**unsat**	**42.78**
		③	150	unsat	24.61
		④	150	unsat	32.72
		③+④	150	unsat	15.30
		①+③+④+⑤+*Know*	150	unsat	20.28
③	STC1	**Without B-Inv**	**150**	**unsat**	**45.89**
		④	150	unsat	37.88
		①+②+④+⑤+*Know*	150	unsat	35.61
④	STC2	**Without B-Inv**	**150**	**unsat**	**55.40**
		③	150	unsat	24.25
		①+②+③+⑤+*Know*	150	unsat	24.13
⑤	DYN	**Without B-Inv**	**150**	**unsat**	**54.17**
		③	150	unsat	21.27
		④	150	unsat	17.29
		③+④	150	unsat	17.28
		①+②+③+④+*Know*	150	unsat	16.14
FSTC1		**Without B-Inv**	**5**	**sat**	**0.15**
		③	5	sat	0.15
		④	5	sat	0.13
		③+④	5	sat	0.14
FSTC2		**Without B-Inv**	**4**	**sat**	**0.14**
		③	4	sat	0.14
		④	4	sat	0.09
		③+④	4	sat	0.14

Fig. 7 Results of checking *revised* MC system by using B-Invariants (and *Know*). *B-Inv* is short for *B-Invariants*.

A trivial and direct way of making use of B-Invariants is to add all checked B-Invariants into the formula to be checked next. However, it is worth pointing out that it is not always better to use more B-Invariants w.r.t. reducing solving time. Taking the case of checking UIC1 as an example, using ②+③+④ costs more time (23.48s) than only using ③ (17.78s). Moreover, the result of using ⑤ (50.22s) shows that sometimes using B-Invariants may not help much for reducing solving time. Reason behind this is that the formula to be solved becomes larger when additional formulas (B-Invariants or structure knowledge of STM) are added, and hence the workload of the SMT solving algorithm is increased as well, especially when the added knowledge does not provide much help. Therefore, there is a balance between the number of B-Invariants added and the benefit obtained from them. A kind of heuristics might be necessary for using B-Invariants, which is considered as one of our future work. In addition, how to come up with B-Invariants that are helpful for checking other predicates is also worth

further investigating. For this point, heuristics and experience often used in interactive theorem proving technique (see e.g., [17]) for generating lemmas (to be used for proving a target invariant) might be useful.

6. Related Work

In addition to the SMT-based approach described, we have been developing an explicit-state model checker `Garakabu2`, which implements search algorithms and techniques similar to SPIN [18], but has a friendly user interface, and facilities for graphically tracking execution of counterexamples. However, explicit algorithms sometimes suffer the state explosion problems for a STM design due to the tremendous number of possible interleavings of execution of constituent STMs. SMT-based BMC could attack this problem to some extent by expressing all possible behaviors within a bound as a formula and searching a possible failure. We plan to integrate the prototype of our encoding into `Garakabu2` as a complementary to the explicit approaches.

Our formalization of static and dynamic aspects of STM designs has been much influenced by the work on formalization and verification of hierarchical UML state machines [8], [9], [19], although *hierarchical structure* is not considered in our work so far. Regarding verification, the primary focus of our work, the work in [9] proposed to translate UML models into the model checker SPIN, and the work in [19] proposed a symbolic encoding approach of UML models into the input of NuSMV [20], through which BDD-based [21] and SAT-based [4] model checking could be conducted. In our work, instead of translating STM designs into the input of any particular symbolic model checkers, we encode them into formulas in SMT-LIB language and thus different state-of-the-art SMT solvers could be used in the back-end.

Our encoding approach is partly inspired by the work in [13], in which model programs representing transition systems are encoded into the input of Z3 [10] for conducting SMT-based bounded reachability analysis. The work in [13] proposed the encoding for abstract data types such as sets and maps that are typically used in model programs, and these types, however, are not considered in our work. In addition, an action existing in parts of multiple model programs could execute at one time step, and STMs in a STM design are asynchronously composited and thus only one is allowed to make a progress at one time step. Boolean flag variables are used to control concurrent execution of this same action in model programs, but we use these flag variables for making the counterexample structure clearer and restricting the asynchronous execution.

One primary difference between our work with the above mentioned ones is that our encoding takes particular care over the issue of multiple assignments to the same variable in a normal cell. Numbers starting from $\alpha(x, k)$, instead of merely k and k–1 as the case in those work, are used as indexes (step numbers) to variables. We believe that this support is worthwhile for practical purpose. Actually, borrowing the SMT approach for encoding sequential C programs described in [14], the action language $\mathcal{L}$ used in our work could be extended greatly as well to cover more elements, e.g., `for` and `while` structures. This idea of SMT-based encoding of systems composed of both sequential programs and non-sequential transition structures is one of our main contributions in the paper.

Another primary difference is the proposed idea of introducing additional knowledge into the target formula to be solved, which has been shown via experiments to be very effective for most of the situations. This idea provides a practical way for end-users of SMT solvers, who have little chance or expertise in improving efficiency of SMT solvers from inside. Although further investigation of this idea, e.g., how to come up with useful B-Invariants, is still necessary, we believe that this is another primary contribution of this paper.

7. Conclusions and Future Work

According to a 2008 survey of JASA (Japan Embedded Systems Technology Association, www.jasa.or.jp), 13 percent of the industrial informants use STM as the modeling language for their (embedded) software development (however without supports for formal analysis). We hope that our work could be helpful for them to develop their own formal verification tools. In addition, we hope that the SMT-based encoding approach, especially the way of making counterexamples clearer, the idea of encoding combination of sequential programs and non-sequential transition structure, and the way of accelerating SMT solving, could be useful to some extent for general usage of SMT techniques.

Much work needs to be done in the future, which includes, e.g., (1) Formalize and encode hierarchical STMs, i.e., (the behaviors of) a STM is called in the normal cells of other STMs. An efficient way should be proposed since trivially flat the hierarchical structure often blowups the state space; (2) Encoding STM designs that employ message passing as the meaning of communication, where encoding *message queue* is a key issue; and (3) Encoding properties in temporal logics like LTL, for which the SMT-based encoding approach [22] implemented in NuSMV [20] for LTL properties might be helpful. Currently, we have been working on aspects (2) and (3), and will report our preliminary results in another opportunity.

Acknowledgment

This research is conducted as a program for the "Regional Innovation Cluster (Global Type, the 2nd Stage)" by Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan. We would like to thank Prof. Kokichi Futatsugi and Kazuhiro Ogata of JAIST, Prof. Shin Nakajima of NII, Prof. Shusaku Iida of Senshu University, *et al.* for giving valuable comments to this work. We appreciate all relevant organizations and people for their support. Last, we would also like to express our great appreciation to

the anonymous reviewers and the associated editor for giving us insightful and valuable comments that are very helpful for improving the paper.

References

[1] W. Kong, T. Shiraishi, Y. Mizushima, N. Katahira, A. Fukuda, and M. Watanabe, "An SMT approach to bounded model checking of design in state transition matrix," 10th ICCSA, pp.231–238, IEEE CS, 2010.
[2] M. Watanabe, "Extended hierarchy state transition matrix design method - version 2.0," CATS Technical Report, 1998. A Japanese Version is published as a book by Higashiginza Publisher, Tokyo, 2006.
[3] CATS Co., Ltd., Japan, "ZIPC v9.2," www.zipc.com, 2009.
[4] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, Handbook of Satisfiability, ch. 26, pp.825–885, IOS Press, 2009.
[5] S. Ranise and C. Tinelli, "The SMT-LIB format: An initial proposal," 1st PDPAR, online: http://goedel.cs.uiowa.edu/ smt-lib, 2003.
[6] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for DPLL (T)," 18th CAV, LNCS 4144, pp.81–94, Springer, 2006.
[7] M. Matsumoto, K. Anada, D. Ueshima, M. Watanabe, and A. Fukuda, "Model checking of state transition matrix," 2nd ITSSV, AIST Technical Report, vol.AIST-PS-2005-017, pp.2–11, 2005.
[8] J. Dubrovin, T. Junttila, and K. Heljanko, "Symbolic step encodings for object based communicating state machines," 10th FMOODS, LNCS 5051, pp.96–112, Springer, 2008.
[9] T. Jussila, J. Dubrovin, T. Junttila, T. Latvala, and I. Porres, "Model checking dynamic and hierarchical UML state machines," 3rd MoDeV2a, pp.94–110, 2006.
[10] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," 14th TACAS, LNCS 4963, pp.337–340, Springer, 2008.
[11] C. Barrett and C. Tinelli, "CVC3," 19th CAV, LNCS 4590, pp.298–302, Springer, 2007.
[12] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," 5th TACAS, LNCS 1579, pp.193–207, Springer, 1999.
[13] M. Veanes, N. Bjørner, and A. Raschke, "An SMT approach to bounded reachability analysis of model programs," 28th FORTE, LNCS 5048, pp.53–68, Springer, 2008.
[14] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," 13th SPIN, LNCS 3925, pp.146–162, Springer, 2006.
[15] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," 38th DAC, pp.530–535, ACM, 2001.
[16] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," Commun. ACM, vol.5, no.7, pp.394–397, 1962.
[17] Y. Bertot and P. Castéran, Interactive Theorem Proving and Program Development, Springer, 2004.
[18] G.J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2008.
[19] J. Dubrovin and T. Junttila, "Symbolic model checking of hierarchical UML state machines," Technical Report B23, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2007.
[20] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," 14th CAV, LNCS 2404, pp.359–364, Springer, 2002.
[21] E. Clarke, O. Grumberg, and D. Peled, Model Checking, MIT Press, 1999.
[22] T. Latvala, A. Biere, K. Heljanko, and T.A. Junttila, "Simple bounded LTL model checking," 5th FMCAD, LNCS 3312, pp.186–200, Springer, 2004.

Weiqiang Kong is a researcher at Fukuoka Industry, Science & Technology Foundation (Fukuoka IST), Japan, since 2008. He received his Bachelor and Master degrees in computer science in 2000 and 2003, respectively, from Wuhan University, China. He received his Ph.D. in information science from Japan Advanced Institute of Science and Technology (JAIST) in 2006. He was a post-doc researcher at JAIST from 2006 to 2008. His research interests include software engineering and formal verification.

Tomohiro Shiraishi is currently a researcher at Fukuoka Industry, Science & Technology Foundation (Fukuoka IST), Japan since 2007. He received his Master degree in system electronic engineering from Tokyo University of Technology in 2000. His research interests include software engineering and formal verification (especially design model checking).

Noriyuki Katahira is currently a researcher at Fukuoka Industry, Science & Technology Foundation (Fukuoka IST), Japan since 2009. His research interests include software engineering and formal verification (especially design model checking).

Masahiko Watanabe is a CTO and senior executive vice president at CATS CO., LTD. Japan since 1998. He received his PhD in information science from Kyushu University in 2009. His research interests include software engineering and formal verification.

Tetsuro Katayama received the Ph.D. degree in engineering from Kyushu University, Fukuoka, Japan in 1996. From 1996 to 2000 he has been a Research Associate at the Graduate School of Information Science, Nara Institute of Science and Technology, Japan. Since 2000 he has been an Associate Professor at the Department of Computer Science and Systems Engineering, Faculty of Engineering, University of Miyazaki, Japan. His research interests include software testing, software visualization, and embedded systems. He is a member of the IPSJ and JSSST.

Akira Fukuda received the BEng, MEng, and Ph.D. degrees in computer science and communication engineering from Kyushu University, Japan, in 1977, 1979, and 1985, respectively. From 1977 to 1981, he worked for the Nippon Telegraph and Telephone Corporation, where he engaged in research on performance evaluation of computer systems and the queueing theory. From 1981 to 1991 and from 1991 to 1993, he worked for the Department of Information Systems and the Department of Computer Science and Communication Engineering, Kyushu University, Japan, respectively. In 1994, he joined Nara Institute of Science and Technology, Japan, as a professor. Since 2001 and 2008, he has been a professor of Graduate School of Information Science and Electrical Engineering, and director of System LSI Research Center, Kyushu Univ., Japan, respectively. His research interests include embedded systems, ubiquitous computing, system software (operating systems, compiler, and run-time systems), parallel and distributed systems, and performance evaluation. He received 1990 IPSJ (Information Society of Japan) Research Award, 1993 IPSJ Best Author Award, and IPSJ Fellow. He is a member of the ACM, the IEEE Computer Society, the IPSJ, and the Operations Research Society of Japan.