

Formal Analysis of STM Design with SAL Infinite Bounded Model Checker

Weiqiang Kong[†], Tomohiro Shiraishi[†], Yuki Mizushima[†], Noriyuki Katahira[†], Akira Fukuda[‡], and Masahiko Watanabe[§]

[†]*Fukuoka Industry, Science & Technology Foundation (Fukuoka IST), Japan,*

[‡]*Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan*

[§]*CATS Co., Ltd., Japan*

{kong, shiraishi, mizushima, katahira}@lab-ist.jp, fukuda@f.csce.kyushu-u.ac.jp, watanabe@zipc.com

Abstract—State Transition Matrix (STM) is a flexible table-like modeling language that has been frequently used for specifying behavior of distributed systems. In this paper, we first present a formalization of the static and dynamic aspects of a STM design (i.e., design written in STM). Consequentially, based on this formalization, we investigate how a STM design can be formally analyzed using SAL, precisely SAL infinite bounded model checker, through a language translation. Specifically, the formal analysis is conducted focusing on four kinds of safety properties related to: (1) Invalid Cells, (2) Static Constraints, (3) Dynamic Constraints, and (4) Deadlock, respectively, since the fulfillment of these properties is commonly desired by industrial practitioners for a STM design. A simple *Internet Connection Control* system is used as our demonstration example.

Keywords—State Transition Matrix (STM), SAL infinite bounded model checker, formal analysis, safety properties

I. INTRODUCTION

State Transition Matrix (STM) [16] is a table-like modeling language for describing behavior of distributed 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 behavior of the system when the specific event occurs in the specific status. A whole system can be constructed by defining each of its subsystems as a STM, and the subsystem STMs communicate through shared variables or message passing.

The notion of STM has been frequently used in industry for developing design of distributed embedded software, and has been adopted as the modeling language of some model-based CASE tools such as ZIPC [1]. However, assuring functional correctness of a STM design (i.e., having the design satisfy certain desired properties) is a non-trivial task. On the one hand, designers may easily introduce subtle logical errors incautiously into a STM design, especially when the design involves multiple inter-communicating STMs, which makes it difficult for the designers to maintain an overall image for the whole system. On the other hand, there is a lack of mechanized formal verification support for STM design. (e.g., ZIPC provides syntactic check only.) The primary reason for this lack may be, to our understanding, that STM is a semi-formal language and no well-documented formalization for STM is available that can be used as a basis for formal verification.

The subject of this paper is to provide mechanical formal verification support to STM design. To this end, we first formalize the structure of STMs and the dynamic behavior of a STM design. Consequentially, based on the formalization, we propose an approach of translating a STM design into the input language of SAL (Symbolic Analysis Laboratory, see: sal.csl.sri.com) [6], [7] as a means of providing STM with verification support, and hence, with the capability of revealing design errors with model checking [4] in an early design phase.

SAL provides a suite of state-of-the-art model checkers, among which is an infinite bounded model checker – called *infBMC* in this paper – that is based on SMT solving technique [15], [9]. We choose to use *infBMC*, rather than other model checkers such as the BDD-based and SAT-based ones included in the SAL suite, since *infBMC*'s input language supports infinite data types like Integers and Reals, and therefore *infBMC* can be directly used to analyze design that has an *infinite state space*. To use other model checkers of SAL, either finite subranges of the infinite data types should be selected, or certain abstract techniques [4] should be employed beforehand to make the design have finite states. Note however that, as the name indicates, *infBMC* is *primarily*¹ used to analyze the state space upto a given depth, rather than the entire infinite state space. Therefore, our usage of *infBMC* is mainly for falsifying, namely finding logical errors of, a STM design.

Based on the translated SAL specification of a STM design, we express four specific kinds of safety properties, which are commonly desired by industrial practitioners for a STM design [3], [14], as Linear Temporal Logic (LTL) formulas, and use *infBMC* to analyze if they are satisfied by the specification. Informally, the properties that we particularly focus on are relate to (1) *Invalid Cells* – a specific event should never occur in a given status, (2) *Static Constraints* – certain correlation between status of different STMs, (3) *Dynamic Constraints* – certain correlation between status-change in one STM with a specific status of another STM, and (4) *Deadlock* – execution of the system should not fall into a situation in which no further process can be made.

¹SAL supports a technique called *k-induction* [8] that allows *infBMC* to iteratively search further into the state space until a threshold is reached and correctness can be concluded. But computing this threshold is often very expensive [5].

ConButton	S	Visible	Invisible
E		0	1
xPressCon	0	Invisible event(Main, Connect)	/
VisCon	1	×	Visible /

DisButton	S	Invisible	Visible
E		0	1
xPressDis	0	/	Invisible event(Main, Disconnect)
VisDis	1	Visible /	×

Main	S	Disconnection	Connection
E		0	1
Connect	0	Connection event(DisButton, VisDis)	/
Disconnect	1	×	Disconnection /

Fig. 1. A simple Internet Connection Control system

We use a simple *Internet Connection Control* system adapted from [3] as our working example to demonstrate the formalization, translation, and analysis. The system consists of three subsystems (components) modeled as three STMs (shown in Figure 1) (1) a button called ConButton, by pressing which, when it is visible, an Internet-connecting instruction is sent, (2) a button called DisButton, by pressing which, when it is visible, an Internet-disconnecting instruction is sent, and (3) a main device called Main that receives the instructions, manages the actual Internet connecting and disconnecting, and after that, sends an instruction back to one of the two buttons for setting it as visible. Note that we intentionally introduced misoperations (design errors) in the system for demonstration purpose.

Organization. Section 2 presents a formalization to STM. Section 3 describes the proposed translation from a STM design into SAL. Section 4 shows how to express the four specific properties in LTL and analyze them with infBMC. Section 5 discusses related work and concludes the paper.

II. STATE TRANSITION MATRIX (STM)

STM is a flexible modeling language. Natural language, programming language, and table notations can be mixed for specifying a system design. In this paper, we consider a subset of STM with the motivation of giving a precise description that can be formally verified efficiently. Adopting a similar approach used in [10], [13] for formally describing UML state machines, we formalize the structure of STMs and the dynamic behavior of a STM design.

A. Structure of STMs

First assume that there exists an 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 an index number denoted by $index(s) \in \mathbb{N}$. 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 $E_{external}$ and internal events $E_{internal}$, where $E_{external} \cap E_{internal} = \emptyset$. An event $e_E \in E_{external}$ is represented by a Boolean variable of $\mathcal{L}$ (whose name is prefixed with a lower-case “x” by convention), and an event $e_I \in E_{internal}$ is represented by a Boolean expression (possibly a Boolean variable) of $\mathcal{L}$. Each event is associated with an index number denoted by $index(e) \in \mathbb{N}$.

- C is a finite set of cells consisting of Normal cells C_{normal} , Ignore cells C_{ignore} , and Invalid cells $C_{invalid}$. Each normal cell $c_N \in C_{normal}$ is a tuple $\langle s, e, a, s' \rangle \in S \times E \times (\mathcal{L} \cup \{/\}) \times S$. We define $source(c_N) = s$, $event(c_N) = e$, $actions(c_N) = a$, 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 Ignore and Invalid cells $c_I \in C_{ignore} \cup C_{invalid}$ as for c_N , but $actions$ and $target$ are not.

A cell c of a STM H , pinpointed by its index numbers ($index(source(c)), index(event(c))$), specifies the behavior of H when event $event(c)$ is dispatched while H is in status $source(c)$ (i.e., $active(H) = source(c)$). If $c \in C_{normal}$, $actions(c)$ will be executed² atomically and after that, H moves to status $target(c)$. If $c \in C_{ignore}$, nothing changes. If $c \in C_{invalid}$, an error occurs. For example, cell (0,0) – $\langle Invisible, xPressDis, / \rangle$ of STM DisButton in Figure 1 is an Ignore cell, meaning intuitively that *pressing the button* (denoted by xPressDis, which is an external event) when the button is *invisible* (denoted by Invisible) changes nothing and is just ignored. Cell (1,1) – $\langle Visible, VisDis, \times \rangle$ is an Invalid cell, meaning intuitively that the event *setting the button as visible* (denoted by VisDis, which is an internal event) should not occur when the button is already *visible* (denoted by Visible).

The action language $\mathcal{L}$ is needed to specify events E and actions of cells C_{normal} . We choose for $\mathcal{L}$ a simple subset of ANSI C language expanded with an additional specific statement, and thus the syntax and semantics basically follow the conventions of the C programming language. Type system of $\mathcal{L}$ consists of Boolean, Integers, and Reals³. Events are expressed as $\mathcal{L}$ expressions of type Boolean and actions are a list of $\mathcal{L}$ statements. Supported *expressions* of $\mathcal{L}$ are (1) Boolean literals TRUE and FALSE, Integer literals and Real literals, (2) Variable identifiers, and (3) infix expressions $leftexpr \text{ op } rightexpr$, where op can be one of +, -, *, /, %, &&, ||, >, <, >=, <=, ==, or !=, with the semantics of ANSI C. Supported *statements* of $\mathcal{L}$ are (1) assignments of the form $lhs = rhs$, (2) If statements of the form `if condition {statement1} else {statement2}`, and (3) an additional *event dispatch* statement of the form `event (H, eventident)`. When an event dispatch statement is executed, the event e denoted by

²The symbol “/” is used to denote *doing nothing*, and in the case $actions(c) = /$, execution of “/” changes nothing.

³The three types can be represented indirectly by the C data types `int` (zero for FALSE and nonzero for TRUE), `int`, and `float`, respectively.

eventident of STM H is dispatched (in this case, e should be an internal event expressed by just a Boolean variable with the name *eventident*). For example, cell (1,0) (denoted here by c) of STM *DisButton* in Figure 1 is a Normal cell, where $source(c) = Visible$, $event(c) = xPressDis$, $actions(c) = event(Main, Disconnect)$, and $target(c) = Invisible$. Intuitive meaning of this cell is that if *pressing the button* when the button is *visible*, the event dispatch statement is executed – namely event *Disconnect* of STM *Main* is dispatched – and after that, the status of *DisButton* is changed to *Invisible*.

B. Dynamic Behavior of a STM Design

A system design D specified in STM generally consists of multiple STMs $H_1, \dots, H_n$ composited asynchronously. We consider, in the paper, that these STMs communicate by using shared variables. The *dynamic behavior* of D is defined based on the notion of *global states*, G .

We fix some notations to be used. For $1 \leq i \leq n$, $Evar(E_i)$ denotes the set of variables involved for expressing events (Boolean expressions of $\mathcal{L}$) of E_i of H_i . Assume that H_i contains m Normal cells, $Avar(H_i) = \bigcup_{j=1}^m Avar(actions(c_j))$ denotes the set of variables involved for expressing actions (lists of statements of $\mathcal{L}$) of H_i , where $c_j \in C_{normal}$ of H_i . We also define $Evar(D) = \bigcup_{i=1}^n Evar(E_i)$ and $Avar(D) = \bigcup_{i=1}^n Avar(H_i)$. Note that the functions $Evar$ and $Avar$ are used freely in an overloading sense, and their signatures could be inferred easily in the contexts. 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 $\langle active(H_1), \dots, active(H_n), value(Evar(D) \cup Avar(D)) \rangle$, where $active(H_i)$ is the current active status of H_i , and $value(Evar(D) \cup Avar(D))$ is the current values of all involved variables in D .

The dynamic behavior of D denoted by $M(D)$ is captured by the tuple $\langle G, g_{init}, \Delta \rangle$, where G is the set of all global states, $g_{init} \in G$ is an initial global state, and $\Delta \subseteq G \times (\bigcup_{i=1}^n H_i.C_{normal}) \times G$ is a transition relation describing how global states change as a result of the execution of normal cells. Note that we use $H_i.C_{normal}$ to denote the set of normal cells C_{normal} of STM H_i . This way applies to sets of events and status of H_i as well.

- In the initial global state g_{init} , the active status of each STM H_i is the status s_i where $index(s_i) = 0$, and all the involved variables of D have their initial (default) values.
- For a cell $c \in H_i.C_{normal}$, c is said to be *enabled* in a global state g if $enabled(c, g) = ((active(H_i) = source(c)) \wedge eval(event(c), g))$ evaluates to TRUE, where $eval(expr, g)$ evaluates the value of an $\mathcal{L}$ expression $expr$ in the context of g . An enabled cell could be executed. Executing the cell c in g moves the system design D to another global state g' , and is denoted by $\langle g, c, g' \rangle \in \Delta$. In g' , $active(H_i) = target(c)$ and the statements of $actions(c)$ are executed atomically as the conventions of C programming language. If $actions(c)$ contains an event dispatch statement $event(H, eventident)$, then

the value of the Boolean variable expressing the event *eventident* of H is set to be TRUE. At each execution step of D , more than one cells might be enabled. In this case, one of them will be selected and executed non-deterministically since $H_1, \dots, H_n$ of D are composited asynchronously.

An external event is dispatched by the environment where the system D resides in, and an internal one is dispatched by the execution of D . An external event $event(c)$ is *consumed* if the cell c is executed in a global state g , and $event(c)$ evaluates to FALSE in the successor global state g' , i.e., $eval(event(c), g') = FALSE$. An internal event is not explicitly consumed in the same way as external ones, i.e., its Boolean value is not directly changed to FALSE in g' , but will just be re-evaluated in g' based on the current values of its involved variables.

III. TRANSLATION OF STM INTO SAL

A. An Overview of SAL Language

The SAL language consists of a type system and four languages for defining expressions, transition relations, modules and contexts. *Types* can be built-in basic types such as Booleans, Natural numbers, Integers and Reals, or user-defined types introduced using uninterpreted type declarations. *Expressions* should be type-correct and consist of constants, variables, arithmetic operators, function, etc. *Transition relations*, expressed as guarded commands, describe value-changes of variables on current and successor states. *Modules*, describing state transition systems, comprise of variables, initialization sections, and transition relations. Modules can be composed both synchronously and asynchronously. *Contexts* groups together definitions of types, functions, modules and module properties.

A complete syntax of SAL language can be found in its language manual [7]. Syntax details relevant to this paper will be introduced later when needed.

B. Translation from STM into SAL Language

1) *STM Data Types, Events, and Status*: The data types of the action language $\mathcal{L}$ for expressing events and actions of STMs have a direct mapping with those basic types included in SAL type system. Therefore, no translation is needed for handling STM data types.

For each external event $e_E \in E_{external}$ of a STM H , a SAL *global* variable of type Boolean is declared. The dispatch (occurrence) of e_E is represented by the variable's value changing from FALSE to TRUE, where the reverse direction represents the consumption of e_E . For each internal event $e_I \in E_{internal}$ of H , a SAL *global* variable is declared for each $\mathcal{L}$ variable $v \in Evar(e_I)$ (involved for expressing e_I) with corresponding types. These variables are declared as global ones since their values could be changed either by the environment (we simulate the environment as a SAL module – to be introduced in Section 3.2.3), or by the execution of other STMs, and SAL language does not allow such changes happen to *local* variables. In addition, for each $\mathcal{L}$ variable

$v \in Avar(H)$ (involved for expressing actions of H), a SAL global variable is declared. Initial values for variables of type Boolean are set by default to FALSE, and initial values for variables of other types are set by default to 0 for Integers, and Reals (SAL's Reals type is a mathematical one rather than a float representation).

To represent the status information of a STM, a local variables of type Integer is declared with initial value 0. A status will then be referred for simplicity by its index number rather than its name.

Let us see the translation of STM DisButton as an example. The variables and their initial values are declared (in a SAL module defining DisButton) as follows:

```
GLOBAL
  xPressDis: BOOLEAN,  VisDis: BOOLEAN,
  Disconnect: BOOLEAN
LOCAL
  DisButtonStatus: INTEGER
INITIALIZATION
  xPressDis = FALSE;    VisDis = FALSE;
  Disconnect = FALSE;  DisButtonStatus = 0
```

2) *STM Cells*: For each Normal cell c , a SAL transition relation in the form of guarded command is generated. A guarded command is a structure: *Label: Guard --> Effect*, where *Label* is the name of the command, *Guard* is a Boolean expression denoting the enable condition of the command, and *Effect* is a list of equalities (assignments) between a left-hand side primed (successor-state) variable and a right-hand side expression on unprimed (current-state) and primed variables. We use the Boolean expression *enabled(c, g)*, denoting the condition for c to be enabled, as *Guard*, and *actions(c)* as the *Effect*, intuitively in the form of *Label: enabled(c, g) --> actions(c)*. The *lhs* variable in each assignment of *actions(c)* is renamed as a primed version. The *rhs* variables referring to values in the successor state are also renamed with primes. For an event dispatch statement in *actions(c)*, we simply translate it into an assignment, whose *lhs* is the primed version of the variable denoting the designated event and *rhs* is the Boolean literal TRUE.

For STM DisButton, two guarded commands are generated for Normal cells (1,0) and (0,1), respectively, which are also contained in the SAL module DisButton:

```
DisButtonT10:
  xPressDis = TRUE AND DisButtonStatus = 1
  -->
  Disconnect' = TRUE; DisButtonStatus' = 0;
  xPressDis' = FALSE;

[] DisButtonT01:
  VisDis = TRUE AND DisButtonStatus = 0
  -->
  DisButtonStatus' = 1;
```

where DisButtonT10 and DisButtonT01 are labels of the commands. For Ignore cell (0,0) and Invalid cell (1,1) of the STM DisButton, no guarded command is generated. This is natural for the Ignore cell since its execution will change nothing. For the Invalid cell, we should, however,

verify whether it is true that event VisDis could not occur in status Visible. We will express this Invalid cell as a safety property and check it in Section 4. The above two guarded commands are composed asynchronously by using the SAL operator [] (put in front of label DisButton01).

Since a module will deadlock if no guard is satisfied, and consequentially, SAL may produce unsound results for a deadlock module, we add a special guarded command:

```
[] DisButtonN: ELSE -->
```

which is satisfied, but changes nothing, whenever all other guards in the module are unsatisfied.

3) *STM (Tables)*: STMs ConButton and Main are translated into two SAL modules in the same way as translating STM DisButton. In addition, we generate an additional SAL module called ExEvent. As the name indicates, our purpose is to use it to generate (or in other words, simulate the occurrence of) external events, here events xPressCon and xPressDis. In the module ExEvent, we declare three guarded commands (also composed by the operator []):

```
ExEventT0:
  xPressCon = FALSE --> xPressCon' = TRUE;

[] ExEventT1:
  xPressDis = FALSE --> xPressDis' = TRUE;

[] ExEventN: ELSE -->
```

The whole system composed asynchronously (by using the SAL operator []) of the above four modules are declared in a SAL module system as follows:

```
system: MODULE =
  ConButton [] DisButton [] Main [] ExEvent;
```

IV. MODEL CHECKING USING SAL

Based on the SAL specifications translated from the STM design, we could express different safety and liveness properties in LTL and model check them using SAL infBMC. In this paper, we focus on four specific kinds of safety properties since the fulfillment of them is commonly desired by industrial practitioners for a STM design [3], [14]. Generally, each of the properties is written as: *propertyName* : THEOREM system | - *property*, where THEOREM is a SAL keyword to declare a property, system is the module describing the design under consideration, and *property* is an LTL expression denoting the property to be checked.

A. Invalid Cells

An Invalid cell indicates that a specific event should never occur during a specific status. When considered by the designer as Invalid, the cell is marked as $\times$ in the STM. However, two potential unexpected scenarios may exist (1) the cell is indeed a Normal cell, but the designer marked it, by mistake, as Invalid, and (2) the cell is indeed an Invalid cell, but the specified event of this cell could, however, occur in the specified status. Both of these two scenarios should be avoided. We could do this by expressing Invalid

cells as LTL properties (representing $\forall c \in \bigcup_{i=1}^n H_i.C_{invalid}$, $\forall g \in G. \neg enabled(c, g)$) and using infBMC to check if any counterexample exists. For Invalid cell (1,1) of DisButton, an LTL property is declared:

```
invalidDisButton: THEOREM system
|-
G(NOT(VisDis = TRUE AND DisButtonStatus = 1));
```

where G is the LTL operator denoting *Globally*, and NOT denotes Boolean negation. infBMC reports a counterexample of length 4. We show an abbreviated version of the counterexample by showing, starting from the initial state, the value-changes of variables <VisDis, disButtonStatus> caused by the execution of guarded commands:

```
<FALSE, 0> == (Tran1: ExEventT0) ==> <FALSE, 0>
== (Tran2: ConButtonT00) ==> <FALSE, 0>
== (Tran3: MainT00) ==> <TRUE, 0>
== (Tran4: DisButtonT01) ==> <TRUE, 1>
```

Intuitively, the counterexample says that: the event xPressCon is first dispatched (Tran1); an Internet-connecting event Connect is then sent (Tran2); after that, the Internet is connected and the event VisDis is sent (Tran3); and last, after setting the DisButton as visible, a counterexample is found, where VisDis is TRUE and active status of DisButton is Visible, namely event VisDis occurs in the status Visible. Reason for this counterexample is not difficult to understand: after execution of cell (0,1) of DisButton, the event VisDis is not consumed. Since this event is an internal event, it is the designer's responsibility to explicitly specify its consumption. If we revise the SAL specification by changing the value of the variable VisDis from TRUE to FALSE in the *Effect* of guarded command DisButtonT01 and model check this property again, no counterexample is found this time when we set a depth, say 50. It should be made clear that the correctness of the property has not been proved yet since only a subset of the state space (within depth 50) is checked. In this paper, we only use infBMC for falsification purpose (i.e., finding counterexamples).

B. Static Constraints

Static constraints demand certain correlation between status of different STMs. We extend *active(H)* as *active(H, g)* to denote active status of STM *H* in global state *g*. Such a correlation can be in the form of, for example, *STM B is in status sb if STM A is in status sa*, i.e., $\forall g \in G: active(H_A, g) = sa \Rightarrow active(H_B, g) = sb$. (more complicated correlation can also be declared.) When a design involves multiple (or large amount) STMs, it is not easy for designers to maintain an overall image of the design and guarantee the satisfaction of it to static constraints (and dynamic constraints as well, to be introduced next).

We could express a static constraint as a LTL property by using the variables representing status information of STMs. An example property is written as follows:

```
staticConstraint: THEOREM system
```

```
|- G(mainStatus = 0 => disButtonStatus = 0);
```

where $\Rightarrow$ is the logical operator *implies*. The LTL formula says that: when the Internet is disconnected (status Disconnection), the button for disconnecting Internet should not be ready for pressing (status Invisible). SAL did not find any counterexample for this property when we set the searching depth as 50.

C. Dynamic Constraints

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, for example, *when the status of STM A has just changed from sa to sa' (where sa' $\neq$ sa), the status of STM B should be sb*, i.e., $\forall g \in G: active(H_A, g) = sa \wedge active(H_A, g') = sa' \Rightarrow active(H_B, g') = sb$, where $\langle g, c, g' \rangle \in \Delta$, and *source(c) = sa, target(c) = sa', sa $\neq$ sa'*.

We could express a dynamic constraint as a LTL property by using the variables representing status information of STMs. An example property is written as follows:

```
dynamicConstraint: THEOREM system
|- G((mainStatus = 1 AND X(mainStatus = 0))
=> X(conButtonStatus = 0));
```

where X is the LTL operator *next*. The LTL property says that: when the Internet has just changed from connection to disconnection, the button for connecting the Internet should be visible (ready for pressing). This property is desirable since if the Internet has changed to disconnection, but the button for connecting the Internet is not visible, then the Internet cannot be connected again. SAL reports a counterexample of length 7 for the property. We will not show the details of the counterexample, but only the sequence of applied guarded commands starting from the initial state to the final state where the counterexample is found:

```
[initialState]
==> ExEventT1 ==> ExEventT0 ==> ConButtonT00
==> MainT00 ==> DisButtonT01 ==> DisButtonT10
==> MainT11 ==> [finalState] <- counterexample
```

From this sequence, we could understand the reason for the counterexample: after the Internet is changed into disconnection and before ConButton executes its cell (1,1), ConButton's status can still be Invisible, and thus breaks the property.

D. Deadlock

For a state transition system, deadlock generally refers to a situation where the system reaches into a (non-terminal) state that has no successors. However, for a STM design, since external events can happen arbitrarily and therefore it is possible that certain transitions could always be executed and successor states may always exist, this general definition is not quite precise for a STM design.

A definition for deadlock of a STM design is given in [3] with the main idea as *a system reaches into a deadlock state if no transitions, except those caused by external events, can*

be executed on this state. We try to follow this definition and directly express deadlock freedom as an LTL property that represents: $\forall g \in G \setminus \{g_{init}\}: \neg \bigwedge \{eval(e, g) = \text{FALSE} \mid e \in \bigcup_{i=1}^n H_i.E_{internal}\}$, where $H_i.E_{internal}$ denotes the set of internal events $E_{internal}$ of H_i . Informal meaning of the formula is that: it is not possible that all the internal events of the design evaluate to FALSE at the same time except for the initial state (since by default they are FALSE initially). We introduce another global variable called *depth* whose initial value is 0, and its value is increased by 1 in the effects of all guarded commands for Normal cells.

```
deadlockFree: THEOREM system
|-
G(NOT(VisCon = FALSE AND VisDis = FALSE AND
      Connect = FALSE AND Disconnect = FALSE
      AND depth > 0));
```

When input this formula into SAL infBMC, a counterexample is reported, which says that the system will fall into a “deadlock” state after some execution steps and have to wait for an external event to restart the system. However, for the Internet Connection Control system, waiting for an external event to connect/disconnect the Internet is a very reasonable behavior. We therefore realize that although the deadlock definition given in [3] can be used for describing deadlock freedom properties for *some* STM design, it may cause model checkers to report unreasonable counterexamples for more *general* cases. Users are therefore required to judge and filter those unreasonable results.

Last, note that SAL tool suite provides a deadlock checker. However, since it can only be used for finite state systems, while a STM design usually have infinite states, we could not use this provided checker.

V. RELATED WORK AND CONCLUSIONS

In addition to SAL, we have also tried to use SPIN [12] as the backend tool to provide STM with formal verification support. We implemented a translator SPIN-Converter [2] to convert STM into *Promela*, SPIN’s input language. The tool is downloadable from <http://www.zipc.com/support/rre/spinconv/>. SPIN is an explicit-state model checker for finite state systems. SAL infBMC is a symbolic one based on SMT solving technique [15], [9]. infBMC can be directly used to check infinite state systems, while to use SPIN, we have to finitize the state space beforehand. However regarding speed, our experience shows that SPIN could generally find counterexamples faster than infBMC. This is a price that one might like to pay for SMT-based techniques to directly handle infinite state systems.

Another effort for model checking STM is the model checker *Garakabu* that we have been developing (an early version of it is reported in [3]). Similar to SPIN, *Garakabu* is an explicit-state model checker equipped with techniques for handling STM design with large state space, such as POR and Bit-State Hashing [11], [12]. However, *Garakabu* is specialized for STM and special design decisions related to STM are reflected in it. *Garakabu* supports model checking of STM

design that uses messaging passing. Counterexample obtained from *Garakabu* can be simulated with the CASE tool ZIPC. We will report the details of *Garakabu* in another opportunity.

Next, we plan to implement a SMT-based searching engine into *Garakabu*. Therefore the work presented in this paper on using SAL infBMC for checking STM design, serving as a starting point and basis, could help us to understand and clarify the *pros and cons* of different model checking techniques. In addition, according to a 2008 survey of JASA (Japan Embedded Systems Technology Association, www.jasa.or.jp/), 13% of the industrial informants use STM as the modeling language for embedded software development. We also hope that our formalization of STM could be helpful for them to start implementing their own formal verification tools, and that the approach of translation and expressing properties as LTL formulas could provide valuable information.

ACKNOWLEDGMENT

This research is conducted as a program for the “Knowledge Cluster Initiative (Stage II)” by Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan. We would like to thank Prof. Kokichi Futatsugi, Kazuhiro Ogata of Japan Advanced Institute of Science and Technology (JAIST), Prof. Shin Nakajima of National Institute of Informatics (NII), *et al.* for giving valuable comments to this work. We also appreciate all relevant organizations and people for their support.

REFERENCES

- [1] CATS Co., Ltd., Japan, *ZIPC Version 9.2*, 2009.
- [2] SPIN Converter, URL: www.zipc.com/support/rre/spinconv/, 2009.
- [3] K. Anada and M. Matsumoto, *The Development of the Model Checking Tool Garakabu*, Embedded Software 2007, pp. 161-169, Nikkei Electronics, 2007.
- [4] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, MIT Press, 1999.
- [5] E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman, *Completeness and complexity of bounded model checking*, In VMCAI 2004, LNCS 2937, pp. 85-96, Springer, 2004.
- [6] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, *SAL 2*, In CAV 2004, LNCS 3114, pp. 496-500, Springer, 2004.
- [7] L. de Moura, S. Owre, and N. Shankar, *The SAL Language Manual*, CSL Technical Report SRI-CSL-01-02 (Rev. 2), 2003.
- [8] L. de Moura, H. Rueß, and M. Sorea, *Bounded Model Checking and Induction: From refutation to verification*, In CAV 2003, LNCS 2392, pp. 14-26, Springer, 2003.
- [9] B. Dutertre and M. Sorea, *Timed Systems in SAL*, SDL Technical Report SRI-SDL-04-03, 2004.
- [10] J. Dubrovin, T. Junttila, K. Heljanko, *Symbolic step encodings for object based communicating state machines*, In FMOODS 2008, LNCS 5051, pp. 96-112, Springer, 2008.
- [11] G. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering, Vol. 23, No. 5, pp. 279-295, 1997.
- [12] G. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, ISBN 0-321-22862-6, Addison-Wesley, 2008.
- [13] T. Jussila, J. Dubrovin, T. Junttila, T. Latvala, I. Porres, *Model checking dynamic and hierarchical UML state machines*, In MoDeV2a 2006, pp. 94-110, 2006.
- [14] M. Matsumoto, K. Anada, D. Ueshima, M. Watanabe, and A. Fukuda, *Model Checking of State Transition Matrix*, In ITSSV 2005, AIST Technical Report AIST-PS-2005-017, pp. 2-11, 2005.
- [15] R. Sebastiani, *Lazy Satisfiability Modulo Theories*, Journal on Satisfiability, Boolean Modeling and Computation, 3(2007): 141-224, 2007.
- [16] M. Watanabe, *Extended Hierarchy State Transition Matrix Design Method -Version 2.0*, CATS Technical Report, 1998.